Job Scheduling to Improve Running Time on a Set of Parallel Processors

Jared Bass

under the direction of
Vahab Mirrokni
Massachusetts Institute of Technology

Research Science Institute
August 1, 2002

Abstract

When more than one processor is available to run a program, we can decrease the total computation time by running certain jobs in parallel. Given a set of jobs and their inter-dependencies, the general problem is to minimize the total computation time by correctly scheduling these jobs on processors. In all but a few cases this problem is NP-complete. We develop polynomial-time heuristic algorithms to approximate the optimal solution within a certain factor. We suggest several such algorithms here involving *mintime* priority functions.

1 Introduction

Given a multiprocessor system, it is often possible to schedule a program to run in less time than it would run on a single processor. In a simple case, if a program is divided into two roughly equal independent halves then each half could be executed on one processor and the results from each half could then be combined at the end if necessary. This would divide the computing time by two. When more processors are available, more improvements are possible; thus, given a program and a multiprocessor computer, the problem is to schedule the program so that it finishes as quickly as possible.

Unlike in the example above, however, many large programs have a huge number of interconnected parts, some computers have large networks of processors, and in general it is never that simple. In fact, the instances of this problem that are most important to real life and have the most applications are NP-complete.

Many polynomial time heuristic algorithms have been developed to approximate the optimal solution to the general scheduling problem. Once an estimate is made, it is very important to know how good an approximation it is for a general case. It is desirable for the length of the schedule obtained to be within a certain (constant) factor of the length of the optimal solution. If ω is the running time of the schedule found by an algorithm, ω_0 is the optimal running time of the program, and $\frac{\omega}{\omega_0} \leq k$ for all possible inputs, then k is an *approximation ratio* of the algorithm (we can also call this a k approximation algorithm). If there exists a case in which $\frac{\omega}{\omega_0} = k$, this bound is *tight*.

1.1 Problem Statement

A program is separated into a series of jobs, which may be divided among the processors and which also have a set of precedence constraints: output from one job may be needed for another, so the receiving job must wait for the first job to finish. These constraints

are represented by a directed acyclic graph (DAG) in which vertices are jobs and edges are directed from one job to any others which require information following its completion. Lengths of time are measured in computational cycles unless otherwise stated.

Each job v_i has associated characteristics. p_i is the length or execution time of v_i. $delay(v_i, v_j)$ is the time delay necessary between the completion of job v_i for information to be sent to job v_j. This accounts for pipelining in the program, in which one processor is able to implement parallel processing within itself. After the first job of a pipelined series finishes, a second part of the processor may finish the series while the part necessary to begin a new job is free. Information from the completion of v_i may only be used after all separate computations in v_i have been completed. If v_j requires information from v_i in order to be run, then v_i is a predecessor or parent of v_j, and v_j is a child of job v_i. These terms are taken from the graph-theoretical representation of the job priorities.

The set of processors, numbered $1, 2, \ldots, m$, has an associated set of communication delays $com(i, j)$, the amount of time necessary to transfer information from a completed job on processor i to processor j. In an actual set of processors information does not transfer instantaneously but must be passed along some channel connecting the processors, so usually $com(i, j) > 0$.

There is a standard representation given to general scheduling problems. In problem $\alpha|\beta|\gamma$, α represents the setup of the processors (generally, P_m is used to represent m identical parallel processors and in some cases $m = \infty$), β gives a list of constraints (e.g. precedence constraints, delays, etc.), and γ is the variable to be optimized. In our applications, γ is C_{max}, or the makespan (total running time) of the program.

2 Related Works

In many cases, p_i is taken to be 1 for all i. This schedules jobs on the most basic level, as it is impossible for a job to take less than one cycle of computing time. Another assumption made in this field at large, although generally less accurate, is that the processors form a complete network with uniform communication delays. Algorithms may be viable for a wider variety of situations than this, and may possibly work more efficiently, but the maximum communication delay must be taken into account when obtaining a bound on the approximation ratio of the algorithm (e.g. [6]). Thus the best obtainable bound on the performance of an algorithm generally assumes a complete network of processors. The formal problem statement for this situation is $P_m|prec., p_i = 1, com(i,j) = \rho|C_{max}$[1]. Heuristics (strategies to quickly approximate solutions to more difficult problems) generally perform better for small communication delays (i.e. $com(i,j) \approx p_i$) than for larger communication delays.

There is no known constant factor k approximation algorithm (i.e., k does not depend on any parameter of the specific program to be scheduled) for the general problem mentioned above, but many specific cases have been shown to have constant approximation factors. When the precedence graph is constrained to an inforest or outforest there are constant bounds, and better bounds are possible when $p_i = 1$. Bounded communication delays result in better approximation factors, and when there are no communication delays there are algorithms with approximation factors less than 2.

One general problem in this area which has been shown to not be NP-complete is $P_m|prec. = tree, p_i = 1, com(i,j) \leq \rho(\text{constant})|C_{max}$, as Karger, 2001 [8] solves it in polynomial time. Generalizing the above problem further results in an NP-complete problem.

A more specific problem that has been studied in depth is $P_3|prec.|C_{max}$, in which there

[1] ρ is generally used as the ratio of the maximum communication delay to the minimum job length.

are no communication delays. Various $\frac{4}{3}$ approximation algorithms have been obtained for this problem, including those mentioned in Lam, 1977 [2]. This paper generalized for the Coffman-Graham algorithm on $P_m|prec.|C_{max}$ It has also been shown that any list-scheduling algorithm for this problem has an approximation factor of at most $\frac{5}{3}$ ([2]). This problem is NP-complete, but for decades it has not been known whether $P_3|prec., p_i = 1|C_{max}$ is NP-complete. The best known approximation factor is the one mentioned above.

A commonly used model for heuristics is list-scheduling, in which each job is scheduled as soon as possible in an empty time slot. If more than one job is available, the job with the highest priority is chosen first. One example of a priority function is the height of a job, or the longest path in the precedence graph to a job with no children. Jobs with longer paths get to start earlier: this decreases a lower bound on the time remaining at each step of the schedule.

3 Approximations Using *mintime*

3.1 Simple *mintime*

The *mintime* function (*mintime* definitions come from Mirrokni, 2002[7]) is an attempt to gain a close approximation of the minimum time remaining after a certain job is executed for all of its descendants to be executed. It is at least as good an approximation as the height of a job, and in many cases is closer to the actual minimum execution time (both the height and *mintime* are at most the optimal execution time). This will be used as a priority function in list-scheduling for the general problem $P_m|prec., p_i, delay(v_i, v_j), com(i, j)|C_{max}$.

For each job v (of length p_v) with children $v_1, v_2, \ldots, v_k$ in decreasing order of $delay(v, v_i)+$ $mintime(v_i)$, $mintime(v)$ is computed as follows:

- If v has no children then $mintime(v) = p_v$

- Otherwise, let $c_1, c_2, \ldots, c_m$ be the communication delays to all other processors[2]. Repeat the following for $i = 1, 2, \ldots, k$:

  1. Find t such that c_t is minimum.
  2. Set $O_i = p_v + delay(v, v_i) + c_t + mintime(v_i)$.
  3. Increase c_t by p_i.

- $mintime(v) = \max_i O_i$

Within this function *mintime* is used as a priority: jobs that will take the longest to finish after v will be scheduled with the greatest precedence. A drawback to this *mintime* is that it assumes all processors are free beginning from the completion of job v, as are all descendants of the job. A simple example to show a fault in this is the three-job program with two of the jobs as parents of the third. The first two jobs will be scheduled on different processors, say 1 and 2, at the same time and their child will have to wait for $com(1, 2)$ before it can be scheduled. In the case of intree precedence constraints (i.e. each job has at most one child) and $\rho = 1$, this simple algorithm has an additive error of $\lceil \frac{m-2}{2} \rceil$ (Varvarigou, 1996[4]).

When *mintime* list-scheduling is applied to $P_3|prec.|C_{max}$ the performance seems to rival that of some $\frac{4}{3}$ approximation algorithms. The example presented in Lam, 1977 [2] to prove the tightness of the $\frac{4}{3}$ bound for the Coffman-Graham algorithm is scheduled optimally by the *mintime* algorithm. No programs with $p_i = 1$ for all v_i have yet been found to give non-optimal solutions to this problem, but no bound better than $\frac{5}{3}$ has yet been proved.

3.2 Simple Bound

Any list-scheduling algorithm for the problem $P_\infty|prec., p_i = 1, com(i, j) = \rho|C_{max}$[3] has an approximation ratio of at most $\rho + 1$. This is obtained if after each job the next job is

[2]Since this is a complete network, all times are ρ except for one which is the processor of v and has $c_t = 0$.
[3]It is not necessary that $m = \infty$, only that m is very large.

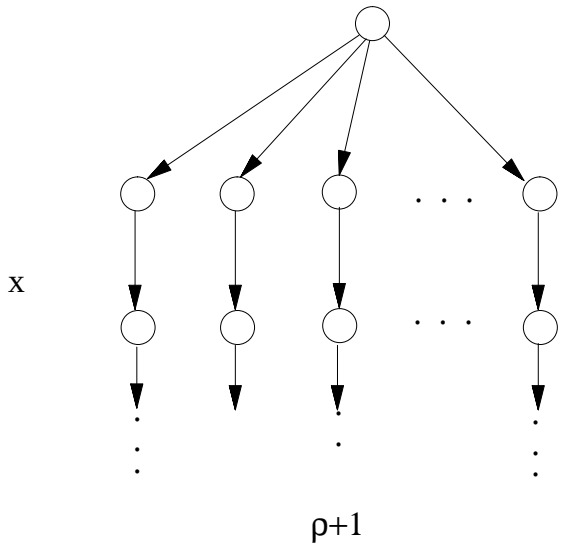

Figure 1: Schedule of jobs which leads to $\rho + 1$ approximation

scheduled as soon as possible but on a different processor. The intuitive view of this is that putting ρ empty cycles after each job allows all jobs to be scheduled without any further delays. This bound is tight for this simple *mintime* function.

To prove the tightness of the bound[4], take a precedence graph shown in Figure 1: an outtree (i.e. each job has at most one parent) with one root node (job) and $\rho + 1$ linear branches of x nodes each with $x >> \rho$. If the root node is scheduled on the first processor, the next ρ jobs, children of the root, will be scheduled in the next ρ time slots. The final child of the first job will also be scheduled on this processor: although there has been a ρ delay so that it could be scheduled anywhere, the order in which processors are checked would result in this placement. Similarly, if the branches were checked in a particular order this pattern would repeat and eventually all jobs would be scheduled on a single processor. The optimal schedule puts one branch on each processor, beginning each as soon as possible. The running time of the *mintime* approximation is $(\rho + 1)x + 1$ while the optimal running

[4]The only assumption made is that precedence of jobs with equal *mintime* is chosen by the order in which they are checked, as is the processor chosen to do a job if more than one is free. This is a very reasonable assumption to make.

time is $1 + \rho + x$, giving a ratio of $\frac{(\rho+1)x+1}{1+\rho+x} \to \rho + 1$ as $x \to \infty$.

3.3 Improved *mintime* and More Time Bounds

In order to improve the *mintime* bound, the existing state of scheduled jobs must be considered. The function $mintime(v, p)$ is a better estimate on the minimum execution time of an available job on a certain processor p given the current set of scheduled jobs. This involves trying all possible schedules of children of v and determining which yields the fastest execution time. It is much more computationally intensive than the simple *mintime*, but still runs in polynomial time if the number of children of each job and the number of processors are constant. By running this process in reverse it is possible to obtain the earliest possible time a certain job may be run, or its availability time, $avtime(v, p)$.

To calculate $mintime(v, q)$ (processor number p has been replaced with q for clarity):

- If v is a leaf, then for all q, $mintime(v, q)=p_v$.
- Otherwise, consider every possible case of assignments of children of v to processors. For each assignment:
  1. For each processor r, let $v_1, v_2, \ldots$ be the children of v assigned to r.
  2. Schedule these jobs such that (for t_i the starting time of job v_i) $\max_i(t_i+mintime(v_i, r))$ is minimum, and let this minimum number be O_r.
  3. Let the *value* of this assignment be $\max_{\text{all processors } r}(O_r)$.
- $mintime(v, q)$ is the minimum value of any assignment.

In order for the running time to be polynomial, the number of possible assignments must be limited. If the number of children of any node and the number of processors are bounded by constants then the running time is polynomial. Otherwise, it is be non-polynomial in these numbers.

To incorporate these functions into a heuristic algorithm, we first define *convenience*. A job v is *convenient* for a processor q, given the existing state of the program, if $avtime(v, q) + mintime(v, q)$ is the minimum over all processors. The scheduling algorithm is slightly modified from list-scheduling, as the priority function changes as the algorithm proceeds. This second heuristic follows:

- Update *avtime* for all available jobs.
- For each processor q, take the set of jobs convenient for q: from this set, schedule the job v with the greatest $mintime(v, q)$ at $avtime(v, q)$ on processor q.

This algorithm takes into account more situations than the original *mintime*-based list-scheduling algorithm, but the most basic case of one job with two predecessors is scheduled "badly" (as it was with the first *mintime* definition) when large communication delays are present. These algorithms neglect the fact that although one processor may allow for earlier execution time, it may be better to wait until later for a more convenient processor to become available. To this end, we define $estimate(v, q)$ for an available job by the following:

1. Assign job v to processor q at the earliest time possible.
2. Compute the availability time for all instructions.
3. For all instructions u, let $value(u)$ be $\min_{\text{all processors } p} avtime(u, p) + mintime(u, p)$.
4. $estimate(v, q) = \max_{\text{all jobs } u} value(u)$.

Redefine convenience so that a job v is convenient for a processor q if $avtime(v, q) + estimate(v, q)$ is minimum over all processors. The scheduling algorithm is the same as the one mentioned above, except now *estimate* has replaced *mintime*.

4 Partitioning the Program

We now try to obtain a constant factor k approximation algorithm for the general scheduling problem. Let the precedence graph of a program be G. Define G_ρ to be the set of jobs in G which have *avtime* at most ρ. Recursively, schedule the graph G as follows

1. Schedule G_ρ using a good approximation, for example the algorithm involving *estimate* above.

2. After the completion of the last job in G_ρ leave all of the next ρ time slots empty.

3. Schedule G/G_ρ.

This adds many empty cycles to the program, but communication delays are less important within each G_ρ. Jobs each have *avtime* at most ρ, and so will be scheduled in approximately ρ time, faster than it would take for two processors to communicate. It is in general much easier to schedule short programs quickly (running times of both the scheduling algorithm and the final program should be fast) than long programs.

If each G_ρ were able to be scheduled in at most $\alpha\frac{|G_\rho|}{m} + \beta\rho + \gamma h(G_\rho)$ (where $h(G)$ is the height of a graph G) then an $\alpha+\beta+\gamma+1$ approximation factor is obtainable for this algorithm. If α, β, and γ are constants then this would lead to a constant factor approximation of the general scheduling problem. Let k be the maximum number of separate G_ρ segments, so $\lceil\frac{h(G)}{\rho}\rceil \leq k \leq \frac{h(G)}{\rho} + 1$ and $k\rho \leq h(G) + \rho$ since there are at least ρ jobs per G_ρ. Adding up these approximation factors over all G_ρ, the final result is that

$$\text{total running time} = \omega \leq \alpha\frac{|G|}{m} + k\beta\rho + \gamma h(G) + (k-1)\rho \leq$$
$$\alpha\frac{|G|}{m} + (\gamma+\beta+1)h(G) + \beta\rho \leq (\alpha+\beta+\gamma+1)\omega_0 + \beta\rho$$

(where ω_0 is the optimal running time) and the $\beta\rho$ term becomes negligible as ω_0 becomes large.

If all jobs in G_ρ form an inforest (i.e. a group of intrees), then $\alpha = 1, \beta = 0$, and $\gamma = 1$ because all jobs may be divided evenly among the processors (each intree in the inforest is scheduled on one processor) and a set of jobs of at most length $h(G_\rho)$ may be scheduled after the rest of the jobs are finished. This leads to a 3 approximation algorithm.

5 Conclusions and Future Work

The *mintime*(v) list-scheduling algorithm is an attempt for a good approximation on a general system of processors. It was improved upon by instead calculating *mintime*(v, p), *avtime*(v, p), and finally *estimate*(v, p) to gain more intelligent approximations of the optimal solution. Future work will include finding bounds on the improved *mintime* heuristics, first in the specific case of $P_3|prec.|C_{max}$.

By partitioning the program into sets of G_ρ which run in relatively short amounts of time, it is possible to obtain a good approximation algorithm after only calculating a bound for a set of jobs with small maximum *avtime*. The running time of the algorithm to schedule these jobs is short, as only very short precedence graphs need to be scheduled. If any constant values of α, β, and γ (as in Section 4) are found, then the partitioning algorithm described will have a constant approximation factor.

6 Acknowledgements

I could not have done this without the help of RSI students and alumni readers: the former for making me feel comfortable enough to delve into my research and the latter for helping me correct my paper.

I would like to thank my tutor, Lisa Powell, who always made sure I was on the right track. Thanks also belong to my mentor, Vahab Mirrokni, who provided me with many insightful ideas to explore and left me thinking after every one of our meetings. He helped me formulate my research and guided me along its exploration. Finally, I would like to thank the Center for Excellence in Education for giving me the opportunity to participate in a program such as RSI and meet so many helpful and interesting people.

References

[1] Hu, T. C.: *Parallel Sequencing and Assembly Line Problems.* Yorktown: IBM Research Center: May 5, 1961.

[2] Lam, Shui and Ravi Sethi: "Worst Case Analysis of Two Scheduling Algorithms." *Siam J. Comput* Vol. **6**, No. **3**, September 1977.

[3] Lenstra, Jan, et. al.: "The Complexity of Scheduling Trees with Communication Delays." *Journal of Algorithms* **20**, 1996: 157-173.

[4] Varvarigou, Theodora, et. al.: "Scheduling In and Out Forests in the Presence of Communication Delays." *IEEE Transactiond on Parallel and Distributed Systems*, Vol. **7**, No. **10**, October 1996.

[5] Guinand, F., et. al. "Worst Case Analysis of Lawler's Algorithm for Scheduling Trees with Communication Delays." *IEEE Transactiond on Parallel and Distributed Systems*, Vol. **8**, No. **10**, October 1997.

[6] Munier, A. *Approximation Algorithms for Scheduling Trees with General Communication Delays.* Paris: Universite P. et M. Curie, September 22, 1996.

[7] Mirrokni, Vahab, et. al. "A Theoretical and Practical Approach to Instruction Scheduling on Spatial Architectures." Unpublished manuscript, 2002.

[8] Karger, D. R., et. al. "Parallel Processor Scheduling with Delay Constraints." ACM-SIAM Symposium on Discrete Algorithms (SODA), January 2001.