Proceedings of the IASTED International Conference
Parallel and Distributed Computing and Systems
November 3-6, 1999 in Cambridge Massachusetts, USA

On-line Algorithms for a Parallel Job Scheduling Problem

Weizhen Mao*
Department of Computer Science
The College of William and Mary
P. O. Box 8795
Williamsburg, VA 23187-8795
USA
wm@cs.wm.edu

Jie Chen and William Watson III
High Performance Computing Group
Thomas Jefferson National Accelerator Facility
12000, Jefferson Avenue
Newport News, VA 23606
USA
{*chen,watson*}*@jlab.org*

Abstract

In this paper, we consider a simple model for parallel job scheduling with computation and communication costs. For each job J_j with processing time p_j, if k_j processors are assigned to execute the job then the actual execution time of the job is $p_j/k_j + ck_j$, where the constant c is the communication cost (or in general the overhead cost) associated with each processor. We define a scheduling problem based on the model to minimize the makespan. We give a study of similarity and difference between our problem and the well-studied two-dimensional packing problem. We design an on-line algorithm with competitive ratio of 2 for the dual-processor case. We present simulation results on the performance of an algorithm for the multiprocessor case.

Keywords: Parallel job scheduling, on-line algorithm, competitive ratio, simulation.

1 Introduction

In the parallel computing environment, multiple processors are available to execute collectively an in-coming job. Although it is sometimes necessary to first partition a job into parallel tasks before allocating processors to the job for the execution of the parallel tasks, the two issues, job partitioning and job scheduling, are usually treated separately. In this paper, we focus on the issue of job scheduling and assume that a job may be executed by any number of processors without worrying about how the partition is done. Intuitively, the more processors are assigned to a job, the less execution (computation) time is needed for the job. However, past research often ignores the additional cost (time) incurred when processors assigned to the same job communicate with each other. In other words, we believe that the more processors are assigned to a job, the more time is spent on communications among the processors. This motivates us to propose the following parallel job scheduling model which incorporates both computation and communication in a simple mathematical way.

In a computer system with m identical processors, n independent jobs $J_1, \ldots, J_n$ are to be executed. Job J_j has length p_j. It is assumed that in a schedule J_j may be executed by any k_j processors, where $1 \leq k_j \leq m$. Let c, a positive number, be the inherent communication (or in general overhead) cost associated with each processor. We assume that $c \leq p_j$ for all j since in the high-performance computing environment, jobs are often large while processors are fast. We then define the execution time of J_j to be $p_j/k_j + ck_j$ if k_j processors are assigned to execute J_j in a schedule. Obviously, The first term in the sum, p_j/k_j, reflects the computation cost and it decreases as k_j increases. The second term, ck_j, is the communication cost and it increases as k_j increases. One may argue that it may be more reasonable to use $c(k_j - 1)$ for the communication cost since when $k_j = 1$ there should be no communication cost for the job. We certainly agree. We choose to use ck_j because we believe that in reality the constant c includes not only communication cost but also any overhead expense associated with, for example, shared memory access and synchronization among processes and so it is more reasonable to link c to each processor. Throughout the paper, we use communication cost to describe c although the reader should be aware that the equivalence may be generalized to a broader scope. We next formulate an optimization problem based on the model with the goal to minimize the makespan (the maximum completion time among all jobs) of a schedule. Note that there are other optimality cri-

*Author for correspondence.

teria to consider, however, we choose the makespan since it measures the utilization of the resources and is most often used in research.

In this paper, we are interested in design and analysis of on-line algorithms for the problem defined above. Assume that the jobs are given in an order, say, $J_1, \ldots, J_n$. An on-line algorithm must schedule a job J_j without the knowledge of the following jobs, $J_{j+1}, \ldots, J_n$. The quality of an on-line algorithm can be measured by its competitive ratio, which is the maximum ratio, for all instances, between the makespan of the schedule obtained by the algorithm and the makespan of the optimal schedule. Clearly, such ratio is always greater than 1. We observe that the closer the competitive ratio is to 1, the better performance the algorithm gives.

The model of parallel job scheduling which allows multiple processors to execute a job simultaneously was first proposed by Du and Leung [5]. Interesting theoretical results have been developed since, such as preemptive scheduling by Deng *et al.* [4], on-line scheduling by Shmoys *et al.* [11] and dynamic scheduling by Feldmann *et al.* [7]. Job scheduling with communications in a network of computers was first studied by Phillips *et al.* [10]. Our work differs in that we simplify the model by eliminating the network graphs and adding the communication cost parameter. The theoretical and practical issues of parallel job scheduling were surveyed by Feitelson *et al.* [6].

We organize this paper as follows. In Section 2, we give a comparison between our parallel job scheduling problem and the two-dimensional packing problem. In Section 3, we present an on-line algorithm for the dual-processor case and prove the tight competitive ratio for the algorithm. In Section 4, we define an on-line algorithm for the multi-processor case of our problem and study it performance by simulation experiments. Finally, in Section 5, we make our conclusions and discuss directions for future research.

2 Relation to two-dimensional packing

According to Azar and Epstein [1], the two-dimensional packing problem can be defined as follows: Given a bin with a fixed width and an unbounded height, and also a set of rectangles, each with a width and a height. The goal is to place the rectangles into the bin to minimize the height of the occupied space in the bin such that (1) the bottoms (widths) of the rectangles are parallel to the bottom (width) of the bin (no rotation is allowed) and (2) the spaces occupied by different rectangles do not overlap. The original two-dimensional packing problem was introduced by Baker *et al.* [2]. They showed the NP-completeness of the problem and designed an off-line algorithm with a performance ratio of 3. Coffman *et al.* [3] gave a more complex off-line algorithm with a performance ratio of 1.5. As for the on-line approach, Azar and Epstein [1] proved that there is an algorithm with a logarithmic competitive ratio.

In the scheduling problem we have defined in Section 1, each job J_j can be considered as a rectangle with width k_j and height $p_j/k_j + ck_j$ if k_j processors are assigned to execute the job by an algorithm. Of course we observe that since k_j is not given as input this equivalence between a job and a rectangle must be coupled with a specific algorithm which does the processor assignments. Suppose that there are m processors in our scheduling problem. These processors can be considered as a bin with width m and an unbounded height. Our goal to minimize the makespan is then equivalent to placing the rectangles that represent the jobs into the bin to minimize the height of the occupied space.

In addition to the complication caused by k_j, which makes our scheduling problem more general and consequently harder than two-dimensional packing, we also notice another difference between the two problems. In two-dimensional packing, a rectangle cannot be partitioned into strips widthwise, i.e., it must be placed into some unoccupied space of the same rectangular shape as the rectangle itself. However, in our scheduling problem, a job may be executed by processors that are not physically adjacent to each other in the bin. This is to say that a job (rectangle) can be sliced into unit-width strips and then be assigned to processors, one strip to one processor, as long as the starting times of these job strips are the same.

From the discussion above, we conclude that our job scheduling problem share similarity to the two-dimensional packing problem to some degree. However, the difference between the two, namely, the specification of k_j and the flexibility of jobs to be partitioned into strips, is too significant for us to apply any of the results developed for two-dimensional packing to our scheduling problem.

3 Analysis of an algorithm for the dual-processor case

When there are two processors ($m = 2$), a job J_j may be executed by one processor, taking time $p_j + c$, or it may be executed by two processors, taking time $p_j/2 + 2c$. It is easy to verify that if $p_j < 2c$ then $p_j + c < p_j/2 + 2c$ and if $p_j \geq 2c$ then $p_j + c \geq p_j/2 + 2c$. Based on this fact, an intuitive algorithm can be designed as follows: For any job J_j, if $p_j < 2c$ then one processor will be used to execute the job, i.e. $k_j = 1$, otherwise, two processors will be used, i.e., $k_j = 2$. The algorithm is on-line in the sense that jobs are processed in the order of $J_1, J_2, \ldots, J_n$. (Note that the algorithm can be also implemented off-line, where all the jobs requiring two processors will be scheduled first followed by jobs using one processor.) Once k_j is determined for each job, jobs will then be scheduled at the earliest possible starting time as in the List Scheduling algorithm first proposed by Graham [8] and later cited by many researchers including Lawler *et al.* [9].

THEOREM 1 *The algorithm for the dual-processor case has a tight competitive ratio of 2.*

Proof. Given any instance, apply the algorithm to construct a schedule. Let S_1 be the set of jobs executed by one processor and S_2 be the set of jobs executed by two processors in the schedule. Let C_{max} be the makespan (maximum completion time) of the schedule. Let I be the total processor idle time before C_{max} in the schedule. We then have

$$\begin{aligned} C_{max} &= \sum_{J_j \in S_2} (p_j/2 + 2c) + \frac{1}{2} \sum_{J_j \in S_1} (p_j + c) + \frac{1}{2} I \\ &= \frac{1}{2} \sum_{\forall J_j} p_j + \frac{1}{2} c|S_1| + 2c|S_2| + \frac{1}{2} I. \end{aligned}$$

Now for the optimal schedule of the same instance, let C^*_{max} be its makespan. Let S^*_1 be the set of jobs executed by one processor and S^*_2 be the set of jobs executed by two processors in the optimal schedule. Let I^* be the total processor idle time before C^*_{max} in the optimal schedule. Then we have

$$\begin{aligned} C^*_{max} &= \sum_{J_j \in S^*_2} (p_j/2 + 2c) + \frac{1}{2} \sum_{J_j \in S^*_1} (p_j + c) + \frac{1}{2} I^* \\ &= \frac{1}{2} \sum_{\forall J_j} p_j + \frac{1}{2} c|S^*_1| + 2c|S^*_2| + \frac{1}{2} I^*. \end{aligned}$$

Now Consider the ratio between C_{max} and C^*_{max}.

$$\begin{aligned} & C_{max}/C^*_{max} \\ = & \frac{\frac{1}{2}\sum_{\forall J_j} p_j + \frac{1}{2}c|S_1| + 2c|S_2| + \frac{1}{2}I}{\frac{1}{2}\sum_{\forall J_j} p_j + \frac{1}{2}c|S^*_1| + 2c|S^*_2| + \frac{1}{2}I^*} \\ \leq & \frac{\frac{1}{2}\sum_{\forall J_j} p_j + \frac{1}{2}c|S_1| + 2c|S_2| + \frac{1}{2}\sum_{J_j \in S_1}(p_j + c)}{\frac{1}{2}\sum_{\forall J_j} p_j + \frac{1}{2}c|S^*_1| + 2c|S^*_2|} \\ & \quad (\text{since } I \leq \sum_{J_j \in S_1} (p_j + c) \text{ and } I^* \geq 0) \\ = & \frac{\frac{1}{2}\sum_{\forall J_j} p_j + cn + c|S_2| + \frac{1}{2}\sum_{J_j \in S_1} p_j}{\frac{1}{2}\sum_{\forall J_j} p_j + \frac{1}{2}cn + \frac{3}{2}c|S^*_2|} \\ & \quad (\text{since } |S_1| + |S_2| = |S^*_1| + |S^*_2| = n) \\ \leq & \frac{\frac{1}{2}\sum_{\forall J_j} p_j + cn + c|S_2| + \frac{1}{2}\sum_{J_j \in S_1} p_j}{\frac{1}{2}\sum_{\forall J_j} p_j + \frac{1}{2}cn} \\ & \quad (\text{since } |S^*_2| \geq 0) \\ \leq & \frac{\frac{1}{2}\sum_{\forall J_j} p_j + cn + \frac{1}{2}\sum_{J_j \in S_2} p_j + \frac{1}{2}\sum_{J_j \in S_1} p_j}{\frac{1}{2}\sum_{\forall J_j} p_j + \frac{1}{2}cn} \\ & \quad (\text{since } p_j \geq 2c \text{ for } J_j \in S_2) \\ = & \frac{\sum_{\forall J_j} p_j + cn}{\frac{1}{2}\sum_{\forall J_j} p_j + \frac{1}{2}cn} \\ = & 2. \end{aligned}$$

To prove the above bound is tight we need to show that the ratio $C_{max}/C^*_{max} \geq 2$ for some instance. Consider an instance with $n = 2k$ jobs. Let $p_j = c$ for odd indices $j = 1, 3, \ldots, n-1$ and $p_j = 2c$ for even indices $j = 2, 4, \ldots, n$. The on-line algorithm will schedule J_1 on one processor taking time $c + c = 2c$, then J_2 on two processors taking time $2c/2 + 2c = 3c$, followed by J_3 on one processor taking time $2c$, then J_4 on two processors taking time $3c$, and so on. So we have

$$C_{max} = 2c \cdot \frac{1}{2}n + 3c \cdot \frac{1}{2}n = \frac{5}{2}cn.$$

The optimal algorithm, on the other hand, will give a schedule no worse than the one that has all jobs on one processor, taking time $c + c = 2c$ for odd-indexed jobs and $2c + c = 3c$ for even-indexed jobs. Since two equal-length jobs can be executed in parallel by two processors, we have

$$C^*_{max} \leq 2c \cdot \frac{1}{4}n + 3c \cdot \frac{1}{4}n = \frac{5}{4}cn.$$

Therefore the ratio for the instance is

$$C_{max}/C^*_{max} \geq \frac{\frac{5}{2}cn}{\frac{5}{4}cn} = 2.$$

4 Analysis of an algorithm for the multi-processor case

4.1 Algorithm description

When there are more than two processors to execute the jobs, algorithm design becomes much harder. First, a scheduling algorithm has to decide how many processors to use to execute each job, i.e., k_j for each J_j, which is not just a choice of 1 or 2 as in the case studied in the previous section since k_j is now an integer between 1 and m. Second, the algorithm has to decide which k_j processors to use and when the execution of J_j starts. A good algorithm should make these two decisions simultaneously since one decision may affect the other. However, such an algorithm is hard to design and analyze. An alternative is to separate the two decision stages and try to make the best possible (greedy) choice for each. This is the kind of algorithms (two-stage algorithms) we focus on in this section.

At the first stage to decide k_j, the number of processors to execute J_j, it is natural and intuitive to choose an integer which minimizes the execution time of the job. The problem then becomes "Determine k_j between 1 and m to minimize $f(k_j) = p_j/k_j + k_j c$". By forcing the derivative of $f(k_j)$ to be zero, we get $k_j = \sqrt{p_j/c}$, which is in fact the value that minimizes $f(k_j)$. Since k_j has to be an integer between 1 and m, we have

$$k_j = \begin{cases} m & \text{if } m \leq \lfloor\sqrt{p_j/c}\rfloor \\ \lfloor\sqrt{p_j/c}\rfloor & \text{if } m \geq \lceil\sqrt{p_j/c}\rceil \\ & \text{and } f(\lfloor\sqrt{p_j/c}\rfloor) \leq f(\lceil\sqrt{p_j/c}\rceil) \\ \lceil\sqrt{p_j/c}\rceil & \text{if } m \geq \lceil\sqrt{p_j/c}\rceil \\ & \text{and } f(\lfloor\sqrt{p_j/c}\rfloor) \geq f(\lceil\sqrt{p_j/c}\rceil). \end{cases}$$

Once k_j is computed for each J_j using the formulas above, we may employ a greedy strategy similar to List Scheduling to construct a schedule at the second stage. The algorithm processes the job in the order given in the on-line fashion. For each job J_j, it allocates the k_j processors such that the job has the earliest starting time.

To summarize, our algorithm for the multi-processor case (which is in fact a generalization of the algorithm for the dual-processor case) can be described as follows:

- Stage one: For each job J_j, compute k_j, the number of processors that will be used to execute the job at stage two, using the formulas given;
- Stage two: For each job J_j (in the input order), choose k_j (whose value has just been computed at stage one) processors such that J_j has the earliest starting time and then schedule the job.

Clearly, the algorithm is time-efficient. In fact, it can be implemented in time $O(mn)$, for m processors and n jobs. In the following subsection, we will present our simulation results on the quality of the solutions produced by the algorithm.

4.2 Simulation results

The purpose of the simulation experiments is to compare the makespan of the schedule obtained by our algorithm proposed in the previous subsection to the makespan of the optimal schedule for a variety of problem inputs. The comparison is mostly done in the form of calculating the ratio between the two makespans. Because a true optimal scheduling algorithm for the problem has to check all possible assignments for k_j (there are m^n such assignments) and all possible job orders (there are $n!$ such orders), resulting a total time complexity of $O(m^n n!)$, the execution of the optimal algorithm is extremely time costly even for inputs with small n and m. We therefore define our inputs for the experiments as follows. Let the machine environment be similar to the Sun HPC Server Series with the number of processors $m = 4, 8, 14, 30, 64$. Let the length of each job p_j be randomly chosen from the range between 20 and 200. Let the communication cost c be 1, 5, 10, and 20. Note that c must be no larger than any p_j according to our assumption. Finally, let n be 10. (This assumption is only used in our first two groups of experiments not our third.)

Even with the above assumptions about the inputs, running the optimal algorithm is still virtually impossible. We then design our simulation to evaluate the two stages of our algorithm separately. The first group of experiments are designed to study the performance of stage one of our algorithm, i.e., whether it makes sense to choose k_j to minimize the execution time of a job. We compute the optimal makespan C^*_{max} (approximately) by using the same job order, i.e., $J_1, \ldots, J_n$, as our on-line algorithm yet checking all possible assignments for k_j (each can be any integer between 1 and m). We call this the first-stage OPT. We then compute C_{max} by applying our on-line algorithm to the same input. The numbers in Table 1 are the ratios between C_{max} and C^*_{max}.

C_{max}/C^*_{max}	$c = 1$	$c = 5$	$c = 10$	$c = 20$
$m = 4$	1.09	1.46	1.81	1.69
$m = 8$	1.46	1.89	1.64	1.64
$m = 14$	2.04	2.08	1.60	1.77
$m = 30$	2.28	1.83	1.21	1.00
$m = 64$	1.71	1.00	1.00	1.00

Table 1. First-stage OPT: Same job order but all possible k_j assignments.

The second group of experiments are designed to study the performance of stage two of our algorithm, i.e., whether it makes sense to schedule a job as early as possible. We compute the optimal makespan C^*_{max} (approximately) by using the same k_j's as in our on-line algorithm yet checking all possible job orders. We call this the second-stage OPT. We then compute C_{max} by applying our on-line algorithm to the same input. The numbers in Table 2 are the ratios between C_{max} and C^*_{max}.

C_{max}/C^*_{max}	$c = 1$	$c = 5$	$c = 10$	$c = 20$
$m = 4$	1.00	1.10	1.00	1.13
$m = 8$	1.00	1.13	1.15	1.22
$m = 14$	1.00	1.27	1.25	1.32
$m = 30$	1.15	1.32	1.21	1.00
$m = 64$	1.15	1.00	1.00	1.00

Table 2. Second-stage OPT: Same k_j assignment but all possible job orders.

In the previous two groups of experiments, the number of jobs is fixed to be 10. To see how our algorithm behaves for large batches of jobs, we design our third group of experiments. Let $m = 30$ and $n = 200$. Let $c = 1, 5, 10, 20$. Since it is impossible to compute the true C^*_{max} by checking all k_j assignments and all job orders, we only look at some random k_j assignments and random job orders. The C^*_{max} used in the experiments is the best among all such combinations. This algorithm, which we call the random OPT, is then forced to terminate after 30 hours of execution. Table 3 gives the makespans of our algorithms and those of the random OPT.

c	1	5	10	20
C_{max}	916	820	823	847
C^*_{max}	742	1806	3156	5876

Table 3. Random OPT: Random k_j assignments and random job orders.

The numbers in all three tables indicate that our algorithm for the multi-processor case performance reasonably

well in practice, particularly in the machine environment and for the job characteristic studied in the simulation.

5 Conclusions

In this paper, we have proposed a model for parallel job scheduling. The model is simple, yet it includes the consideration of communication costs (or in general overhead costs) among processors working on a job simultaneously. We have defined a scheduling problem based on the model to minimize the makespan of a schedule. We have made comparisons between our scheduling problem and the two-dimensional packing problem and pointed out their similarity and difference. For the dual-processor case, we have given an on-line algorithm with a tight competitive ratio of 2. For the multi-processor case, we have presented an algorithm and used simulation to show that the algorithm performs reasonably well in practice.

There are several directions for future research. One, we are interested in other models which incorporate communication (overhead) costs. Two, we wish to mathematically analyze the performance of our algorithm for the multi-processor case. Third, we are interested in better ways to obtain optimal schedules to make the simulation more accurate.

Acknowledgement

We wish to thank Jessen Havill for pointing out an error in the proof of Theorem 1 and the two anonymous referees for their helpful comments.

References

[1] Y. Azar and L. Epstein, On two dimensional packing, *Journal of Algorithms* **25**, 290–310 (1997).

[2] B. S. Baker, E. G. Coffman, Jr., and R. L. Rivest, Orthogonal packings in two dimensions, *SIAM Journal on Computing*, 846–855 (1980).

[3] E. G. Coffman, Jr., M. R. Garey, D. S. Johnson, and R. E. Tarjan, Performance bounds for level oriented two-dimensional packing algorithms, *SIAM Journal on Computing*, 808–826 (1980).

[4] X. Deng, N. Gu, T. Brecht, and K. Lu, Preemptive scheduling of parallel jobs on multiprocessors, *Proceedings of the seventh ACM-SIAM Symposium on Discrete Algorithms*, 159–167 (1996).

[5] J. Du and J. Y-H. Leung, Complexity of scheduling parallel task systems, *SIAM Journal on Discrete Mathematics* 2, 473–487 (1989).

[6] D. G. Feitelson, L. Rudolph, U. Schwiegelshohn, K. Sevcik, and P. Wong, Theory and practice in parallel job scheduling, *Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph (eds.), Lecture Notes in Computer Science, Springer Verlag (1997).

[7] A. Feldmann, J. Sgall, and S-H. Teng, Dynamic scheduling on parallel machines, *Theoretical Computer Science* 130, 49–72 (1994).

[8] R. L. Graham, Bounds for multiprocessing timing anomalies, *SIAM J. Appl. Math.* **17**, 416–429 (1969).

[9] E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys, Sequencing and scheduling: Algorithms and complexity, *Handbooks in Operations Research and Management Science, Volume 4: Logistics of Production and Inventory*, edited by So. C. Graves, A. H. Rinnooy Kan, and P. Zipkin, North-Holland (1990).

[10] C. Phillips, C. Stein, and J. Wein, Task scheduling in networks, *Algorithm Theory–SWAT 94*, E. M. Schmidt and S. Skyum (eds.), Lecture Notes in Computer Science, Springer Verlag (1994).

[11] D. Shmoys, J. Wein, and D. Williamson, Scheduling parallel machines on-line, *SIAM Journal on Computing* 24, 1313–1331 (1995).