Parallel Job Scheduling with Overhead: A Benchmark Study

Richard A. Dutton and Weizhen Mao
Department of Computer Science
College of William and Mary
Williamsburg, VA 23187
USA
{radutt,wm}@cs.wm.edu

Jie Chen and William Watson, III
The Scientific Computing Group
Jefferson Lab
Newport News, VA 23606
USA
{chen,watson}@jlab.org

Abstract

We study parallel job scheduling, where each job may be scheduled on any number of available processors in a given parallel system. We propose a mathematical model to estimate a job's execution time when assigned to multiple parallel processors. The model incorporates both the linear computation speedup achieved by having multiple processors to execute a job and the overhead incurred due to communication, synchronization, and management of multiple processors working on the same job. We show that the model is sophisticated enough to reflect the reality in parallel job execution and meanwhile also concise enough to make theoretical analysis possible. In particular, we study the validity of our overhead model by running well-known benchmarks on a parallel system with 1024 processors. We compare our fitting results with the traditional linear model without the overhead. The comparison shows conclusively that our model more accurately reflects the effect of the number of processors on the execution time. We also summarize some theoretical results for a parallel job schedule problem that uses our overhead model to calculate execution times.

1. Introduction

Along with the power of parallelization in supercomputers and massively parallel processors (MPPs) comes the problem of deciding how to utilize this increased processing capability efficiently. In a parallel computing environment, multiple processors are available to execute collectively submitted jobs. In this paper, we assume that jobs are *malleable* in that they can distribute their workload evenly among any number of available processors in a parallel system in order to shorten their execution times.

The ideal execution time of a malleable job with length p is solely defined by the computation time of the job, which is p/k, if the job's execution utilizes k processors. The more processors are assigned to execute the job, the less the execution time of the job. Note that when $k = 1$, the execution time is p. So, if t is the execution time of the job, then

$$t = p/k. \tag{1}$$

We call the above way of computing executions times the *linear model* since it only considers the linear speedup in computation times. It is also the traditional model historically used by theoreticians in the parallel computing community.

However, inherently serial code and parallel processing overhead (from process management, shared memory access and contention, communication, and/or synchronization) often prevent actual execution times from achieving the ideal in the linear model. We define an alternative model which incorporates such overhead. Similar to the linear model, we derive the execution time of a job with length p as a function of k, where k is the number of parallel processors assigned to execute the job. The function takes into account both the speedup in computation time (i.e., p/k) and the slowdown due to overhead. We naturally assume that the overhead (also measured as time) is proportional to k and should be zero when $k = 1$. In addition, to simplify the function for the ease of analysis later on, we assume that the overhead is independent of the job's length, but may be related to the parallel system's architecture. To avoid having to consider different physical features of the parallel system, we use a constant c to reflect the impact of the system on overhead. Specifically, among the k processors assigned to the job, consider one as the master processor that initiates the parallel execution of the job and the other $k - 1$ as the slave processors. We define c, a constant linked to the parallel system, to be the overhead per slave processor. Then, the total overhead incurred when executing the job becomes $(k - 1)c$. So, the execution time of the job is

$$t = p/k + (k - 1)c. \tag{2}$$

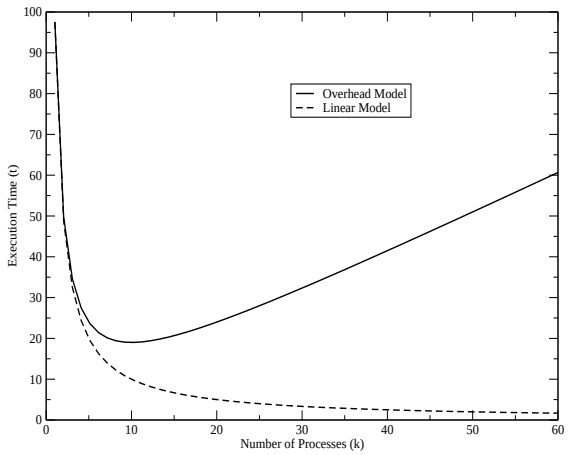

Figure 1. Execution time t of a job with $p = 100$ as a function of k in the linear model ($c = 0$) and the overhead model ($c = 1$). Execution time in the overhead model is minimized when $k = \sqrt{p/c} = 10$.

We call the above way of computing execution times the *overhead model*, as oppose to the traditional linear model that does not have the overhead term, which is the second term in (2).

Figure 1 shows the curves for the execution times for both models. In the linear model, the execution time t decreases monotonically as k increases. In the overhead model, however, as k increases, t first drops drastically until it reaches the minimum at $k = \sqrt{p/c}$ and then climbs up gradually. The value of k, $\sqrt{p/c}$, at which the minimum execution time is achieved, is obtained by first computing the first-order derivative of the execution time function (2) with respect to k and then finding the value of k that makes the derivative to be zero.

There are similar models proposed to incorporate overhead in literature, such as in [26]. However, these models are usually too complicated to allow theoretical analysis. We will show in this paper that our overhead model is not only a model that captures the reality in parallel systems but also one that makes theoretical analysis possible.

We are interested in how well our overhead model reflects the effect of a parallel system on execution times of jobs, especially, whether execution times indeed increase (or decrease at a slower rate than in the linear model) when too many parallel processors are assigned to execute jobs, as our overhead model indicates. Our approach is to take some well-known parallel benchmarks and run each of them on a parallel system using various numbers of processors. For each benchmark, the execution times are recorded and then plotted for all runs. Then a least square fit routine is used to fit both the overhead and the linear models, adjusting model parameters to get the best fit to the experimental data. Finally, the fitting results for both models are compared to validate our overhead model and confirm its superiority over the linear model.

We note that various parallel job scheduling problems may be formulated under our proposed overhead model. For example, we can define a scheduling problem as follows. We have available $m \geq 2$ identical processors in a parallel system and we are given n parallel jobs $J_1, J_2, \ldots, J_n$ to schedule. For each job J_j with length p_j, a scheduling algorithm must assign both $k_j \in \{1, 2, \ldots, m\}$ processors to execute the job and a start time s_j that does not conflict with previously scheduled jobs and is a time instant when there are k_j processors available. A job must be executed simultaneously on all assigned processors and may not be preempted. Using our overhead model to calculate the execution time t_j for each scheduled job J_j, we wish to minimize the makespan of the schedule, defined to be $C = \max_j C_j$, where $C_j = s_j + t_j$ is the completion time of job J_j. We choose the makespan in this formulation since it measures the utilization of resources and is most often used in past research.

A scheduling algorithm is said to be *online* if jobs have to be scheduled in the order that they appear in the job sequence, i.e., $J_1, J_2, \ldots, J_n$. In other words, when an online algorithm makes a scheduling decision (which includes choosing k_j and s_j) on job J_j, it has no knowledge about $J_{j+1}, \ldots, J_n$, such as their lengths. On the other hand, an *offline* scheduling algorithm receives the entire job sequence in advance and thus can usually make better decisions based on the complete knowledge on jobs.

We organize the rest of this paper as follows. In Section 2, we discuss the related work. In Section 3, we give our experimental results for the benchmark study of our overhead model. In Section 4, we summarize theoretical results for job scheduling under the overhead model. In Section 5, we give our conclusions and final remarks.

2. Related Work

Job scheduling has been a fruitful area of research for many decades. A survey on sequential job scheduling can be found in [14]. The study of parallel job scheduling has recently drawn a lot of attention from researchers with the rapid development of parallel systems [9]. In fact, the general problem of parallel job scheduling which allows multiple processors to execute a job simultaneously, especially, the case of malleable jobs, was first proposed in [7]. Since it was proved that scheduling malleable jobs is NP-hard [7], offline approximation algorithms were designed and then analyzed by the means of worst-case performance ra-

tio study [5, 15, 18, 20, 27]. Variations of malleable job scheduling, such as with preemption of jobs [4] and with precedence constraints among jobs [13], were also studied.

Online algorithms [16] are typically used to solve scheduling problems, where jobs are considered and scheduled in the order given in the input job sequence or in the order of arrival times. Such algorithms are usually analyzed using competitive analysis, which compares the quality of the schedule produced by an online algorithm with that of the optimal schedule for the same input. Online scheduling of both sequencial and parallel jobs was surveyed in [24, 25, 23].

The overhead we add in our overhead model is actually a type of *setup cost*, a term sometimes used in the scheduling community [1]. In the related literature, setup cost is typically related to job-specific preparation that is necessary for the execution of a sequential job. In our model, the setup cost is only related to overhead arising from parallel processing. Job scheduling with communication in a network of computers was studied in [21]. Our work differs from this work in that we simplify the model by eliminating the network graphs and adding the overhead constant c.

Models similar to ours were proposed in [26]. For example, a model defined in [26] used a different overhead constant for each job. Attempts were made to evaluate various models for estimating execution times of malleable jobs in [17]. But the experiments were conducted using published execution time data that were obtained by running benchmarks on no more than 4 processors, with the exception of one benchmark, which was run on up to 11 processors. Our work differs from [26] and [17] in several ways. First, we use a start-of-art parallel system with 1024 processors and run various well-known parallel benchmarks [2, 8, 29] on various numbers of processors, up to 512. As a result, the data of execution times are more widely collected to reflect the massive degree of parallelism. Second, our theoretical work [6, 11, 12, 19] focuses on the design of efficient online algorithms for parallel job scheduling in the overhead model and the competitive analysis of these algorithms. The detailed mathematical analysis was not done in the parallel scheduling community before.

3. Experiments and Fitting Results

3.1. Method of Assessment

We design our experiments with the goal to evaluate our proposed overhead model, especially in how well in reality the model reflects the effect of the number of processors assigned to execute a job on the execution of the job. We are interested in whether the execution time indeed increases (or decreases at a slower rate than in the linear model) when too many processors are executing a job.

We follow the method of assessment given below:

- Select a group of parallel benchmarks (jobs), which are parallel implementations of algorithms for various scientific applications.
- Select a parallel system (cluster) with a large number of processors to use.
- For each benchmark, run it on various numbers of processors in the parallel system and record the corresponding execution time for each run.
- For each benchmark, plot the numbers of processors (x-axis) versus the corresponding execution times (y-axis) and use Least-Square-Fit to fit the plot with both the formula in the overhead model and the formula in the linear model.
- Evaluate the fitting results for both models for all benchmarks.

Note that the overhead constant c is one of the parameters to be determined in fitting. Since it is associated with the parallel system, it stays the same for all benchmarks and in all runs. Also note that since our parallel system uses message passing for communication, the overhead mainly comes from the delay due to communication between processors. For this reason, the term overhead is sometimes replaced by the term communication in the discussion in this section.

3.2. Parallel Benchmarks

We choose the NAS Parallel Benchmark (NPB) [2] and the xhpl benchmark [29] for our study. The NAS benchmarks were developed at the NASA Ames Research Center to evaluate the performance of parallel and distributed systems. The xhpl benchmark is a part of the HPC Challenge Benchmark [28] and it measures the floating point rate of execution for solving a linear system of equations on parallel computers.

NPB is a set of 8 benchmark programs. It has five kernels: EP, MG, CG, FT, and IS, and three simulated computational fluid dynamics (CFD) programs: BT, SP, and LU. In this paper, we use the Message Passing Interface (MPI) implementation of the original NPB benchmarks with release version 3.2. The performance data are collected for the fixed problem size 128^3, which is designated as class C for the NPB benchmarks.

The four benchmarks selected from NPB are CG, FT, IS and BT. The Conjugate Gradient (CG) benchmark is used to calculate an approximation to the smallest eigenvalue of a large, sparse, symmetric positive definite matrix. The kernel

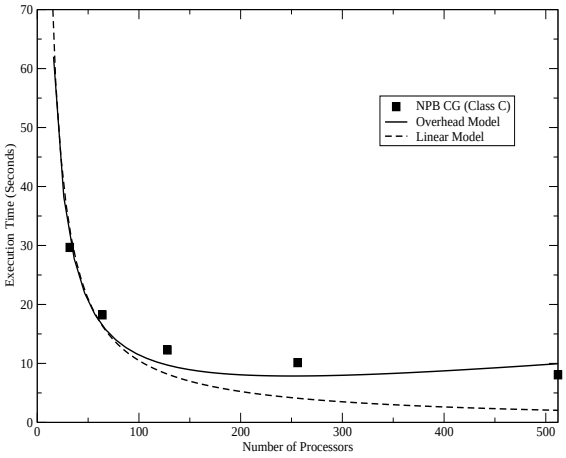

Figure 2. Comparing the linear ($c = 0$) and overhead ($c > 0$) models using the Conjugate Gradient (CG) benchmark.

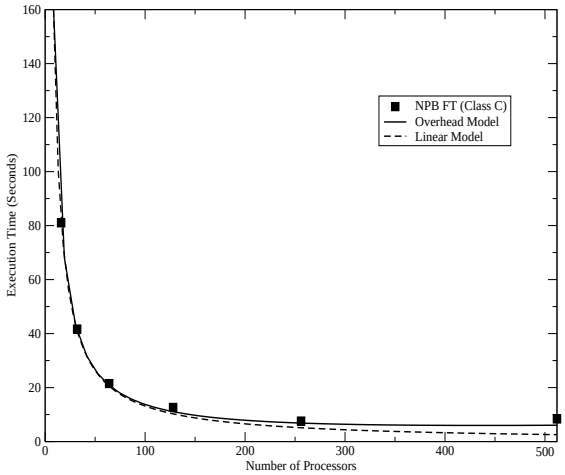

Figure 3. Comparing the linear ($c = 0$) and overhead ($c > 0$) models using the Fourier Transform (FT) benchmark.

features unstructured and irregular long distance communication, which posts a challenge to any parallel system. The Fourier Transform (FT) benchmark is a 3-D Partial Differential Equation (PDF) solver using the fast Fourier transformations. This kernel performs the essence of many "spectral" codes. It is a very good test of long-distance communication performance. The Integer Sort (IS) benchmark performs a sorting operation that is important in the "particle cell method" applications [10], which require sorting operations to assign and reassign particles to appropriate locations. This benchmark reveals both integer computation and communication performance. The Block Tridiagonal (BT) benchmark solves block tridiagonal equations with 5×5 block size.

We do not choose the remaining four benchmarks (EP, MG, LU, and SP) in NPB. In the Embarrassingly Parallel (EP) benchmark, two-dimensional statistics are accumulated from a large number of Gaussian pseudo-random numbers. This problem is typical of many Monte-Carlo simulations. Since it requires no communication, thus little overhead, we do not consider it in our study. The Multi-Grid (MG) benchmark solves a 3-D Poisson Partial Differential Equation. The communication patterns of this program are highly structured and simple, which reveals little overhead of a system. Therefore the benchmark is not studied in this paper, either. The Lower-Upper (LU) diagonal benchmark does not perform the LU factorization [22] but instead solves a regular-sparse, block (5×5) lower and upper triangular system using the Successive Over-Relaxation (SSOR) method [3]. This benchmark requires limited amount of parallelism, thus is not of our interest. The Scalar Pentadiagonal (SP) benchmark solves scalar pentadiagonal equations. Since it is similar in communication characteristics to BT [8], which is already selected for our study, we omit SP.

The last benchmark used in our study is not from NPB. The xhpl is a portable implementation of the high performance Linpack benchmark [30] for distributed memory computers using MPI. It uses the two dimensional block-cyclic data distribution and the right-looking variant of the LU factorization with row partial pivoting to solve large randomly generated dense linear equations. The benchmark has been used to classify the top 500 fastest computers [32]. The benchmark reveals not only floating point computation but also network communication performance. The performance data of xhpl are obtained for a fixed problem size of 20000×20000.

3.3. System Environment

All benchmark results are collected on a cluster consisting of more than 128 nodes (or 1024 processors) located at Jefferson Lab, a national lab affiliated with the United States Department of Energy. Each node has two quad-core AMD Opteron processors running at 1.9 GHz, and nodes are connected with the Infiniband DDR networks that provide 20Gb per second point-to-point bandwidth. Each node has 4GB memory and is running Fedora Core 7 Linux distribution with Linux kernel 2.6.22. The compiler used to compile the benchmark programs is gcc 4.1.2. The MPI library is the MVAPICH version 0.9.9 [31] from Ohio State University.

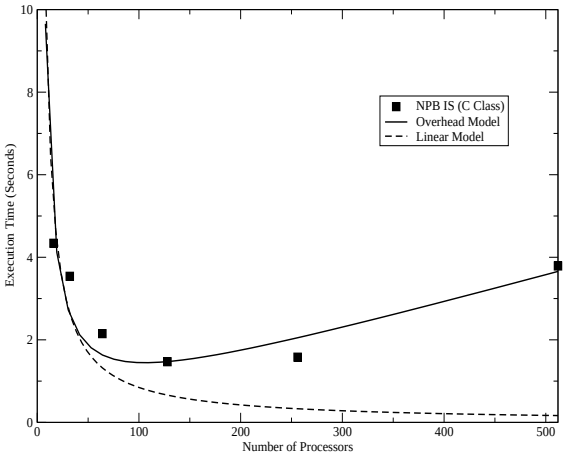

Figure 4. Comparing the linear ($c = 0$) and overhead ($c > 0$) models using the Integer Sort (IS) benchmark.

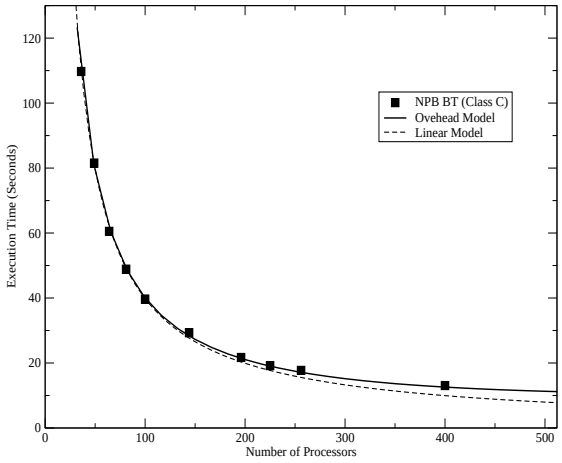

Figure 5. Comparing the linear ($c = 0$) and overhead ($c > 0$) models using the Block Tridiagonal (BT) benchmark.

3.4. Results

The fitting results for the five benchmarks (CG, FT, IS, BT, and xhpl) are given in Figures 2, 3, 4, 5, 6. The solid square dots are data points collected from running benchmarks on the parallel system. The solid curves are the fitting results using the overhead model with $c > 0$ while the dashed curves are the fitting results using the linear model with $c = 0$.

First, we notice that in all five figures that show fitting results, the solids curves are closer to the solid dots than the dashed curves. This indicates that the formula in the overhead model is more accurate to reflect the experimental data than the formula in the linear model.

Second, for one benchmark, IS, the execution time measured indeed starts to increase as the number of processors passes over 250, as shown in Figure 4. For other four benchmarks, the execution time measured decreases in the range of processor counts tested, which is up to 512, as shown in Figures 2, 3, 5, 6. This indicates that they probably have not reached the minimums for $k \leq 512$. However, the rate of decrease, for all four benchmarks, is slower than the rate in the linear model.

To analyze the difference in the fitting results between the two models mathematically, we follow the steps below to generate a relative error histogram.

- Let P be the set of data points collected during the experiments for all benchmarks, where a data point is the execution time of a benchmark in a certain run.
- For each data point $t \in P$, let t' be the corresponding value from fitting using a chosen model. Compute the relative error of the fitting result with respect to the experimental result as $r = (t' - t)/t$.
- Compute $count_i$, the number of data points with relative error in the interval $[0.1i, 0.1i + 0.1)$, where i is an integer.
- Normalize $count_i$ by computing $f_i = count_i/|P|$, called frequency.

Once we get the frequency value f_i for all is at which $f_i > 0$, we can create a two-dimensional histogram, where the x-axis is the index i and the y-axis is the frequency. For easily recognizing the interval of relative errors that each i represents, we replace i along the x-axis with $0.1i$, the lower bound of the interval of length 0.1. We then obtain the histogram as shown in Figure 7, where black bars are for the overhead model and grey bars are for the linear model.

From Figure 7, we observe that the black bars, corresponding to the number of fitting values generated from the overhead model, are mostly clustered in intervals of smaller relative errors. For example, for the overhead model, the tallest black bar indicates that the fitting values of roughly 32% of the data points have the relative error of between $[-0.1, 0)$, and the leftmost black bar indicates that the largest relative error (in absolute value) of the fitting values for the model is at most 0.5. In addition, we observe that the grey bars, corresponding to the number of fitting values generated from the linear model, are more widely spread out in relative errors. For example, for the linear model, there are less than 18% of data points with the fitting values in

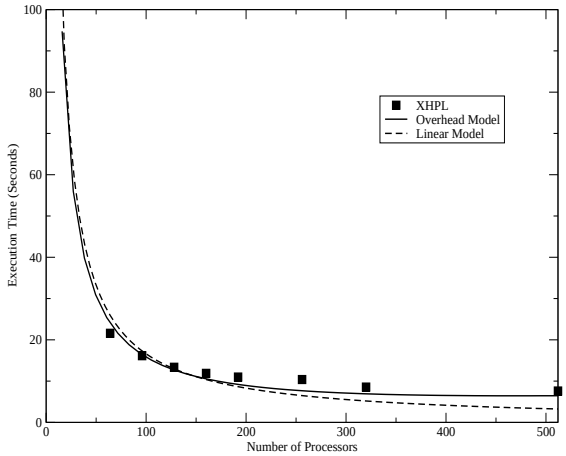

Figure 6. Comparing the linear ($c = 0$) and overhead ($c > 0$) models using the high performance Linpack (xhpl) benchmark.

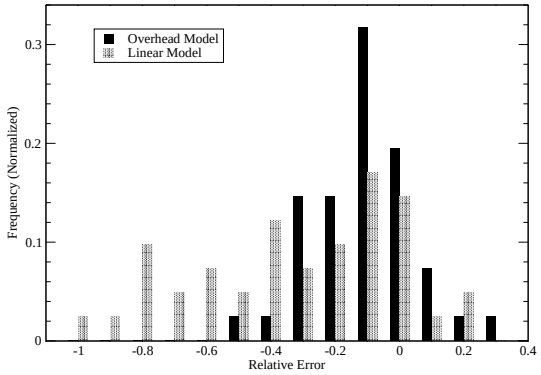

Figure 7. The relative error histogram for comparing the linear and overhead models.

the $[-0.1, 0)$ interval of relative errors, and there are fitting values with relative error as bad as -1.

In summary, we conclude from our experiments that our overhead model indeed describes the relation between the number of assigned processors and the execution time more accurately than the linear model. We point out that although more sophisticated models to estimate execution times are available [26], our overhead model is one that is sophisticated enough to beat the widely used linear model, and meanwhile, is simple enough for us to conduct mathematical analysis of algorithms for scheduling problems defined using the model, which will be clear in the next section.

4. A Scheduling Problem and Its Algorithms

We recall the parallel job scheduling problem defined in Section 1, where n malleable jobs $J_1, \ldots, J_n$ are given as the input sequence and the goal is to schedule the jobs in a parallel system with m processors to minimize the makespan of the schedule, using the overhead model to calculate the execution times of jobs. Here, we summarize the theoretical results obtained for the problem, focusing on the design of online algorithms for the problem and the competitive analysis of the algorithms. By doing so, we show that the mathematical formula in the overhead model in calculating execution times makes it possible to design some intuitive but nontrivial algorithms and to perform challenging but doable analysis.

We focus on online algorithms, in which jobs are scheduled in the order given in the input sequence. To schedule the next job J_j (where j starts from 1), an algorithm must make the following two decisions for the job:

- Decide k_j, the number of processors to assign to the job according to the system utilization at the time, and
- Decide s_j, the start time of the jobs such that there will be k_j processors available at s_j.

Two online algorithms have been studied. They are the Shortest-Execution-Time algorithm (SET) and the Earliest-Completion-Time algorithm (ECT).

SET decides k_j and s_j separately for each J_j. For each job J_j (starting from J_1 and following the order in the input job sequence thereafter), the algorithm computes k_j so that it minimizes the execution time function $t = p_j/k + (k-1)c$ and then schedules the job on any k_j processors that result in the earliest start time s_j. Note that although $k = \sqrt{p_j/c}$ minimizes the execution time function t, since k_j is an integer in $[1, m]$, the actual value of k_j is set to be either $\min\{m, \lfloor\sqrt{p_j/c}\rfloor\}$ or $\min\{m, \lceil\sqrt{p_j/c}\rceil\}$, depending on which of the two gives the smaller execution time.

ECT combines the selection of k_j and s_j into one step. For each job J_j (starting from J_1 and following the order in the input job sequence thereafter), the algorithm chooses $k_j \in [1, m]$ such that there are k_j processors at some time to execute the job, resulting in the earliest time that J_j can possibly be completed, i.e., the earliest completion time. This can be expressed as choosing k_j such that the completion time of J_j

$$C_j = \min_{1 \le k \le m} \{s_j + p_j/k + (k-1)c\}.$$

Here s_j is implicitly determined by the value of k.

Both algorithms are intuitive in concept and efficient in time complexity. Simulation study of their performance has shown that ECT often gives schedules with makespans shorter (better) than the makespans of the schedules given by SET [12]. Competitive analysis has been conducted to study mathematically the performance of these online algorithms. Before we state the results of competitive analysis, we define the term of competitive ratio, which is used to quantitatively measure the quality of solutions produced by an online algorithm when compared with the optimal solutions.

We say that an online algorithm A is *r-competitive* if, for all instances I, $C(I) \leq r \cdot C^*(I) + b$, where $C(I)$ and $C^*(I)$ are the makespan of the schedule constructed by A for instance I and the optimal makespan for instance I, respectively, and b is constant with respect to the job sequence (but may depend on m, which is constant for a given parallel system). The competitive ratio of A, $R[A]$, is the infimum (minimum) over all such values r.

We observe that the competitive ratio of an online algorithm reveals how close the solution given by an online algorithm is to the optimal solution in the worst case for all inputs. If algorithm A has a competitive ratio of $R[A]$, then we may say that for any input job sequence, the makespan of the schedule constructed by algorithm A is no more than $R[A]$ times that of the optimal schedule for the same input sequence (plus a constant b). We also see that the smaller $R[A]$ is, the better the worst-case performance of A is.

The following is the results on competitive ratios for SET and ECT.

- For SET, $R[SET]$ is $4(m-1)/m$ for even $m \geq 2$ and $4m/(m+1)$ for odd $m \geq 3$, where m is the total number of processors in the parallel system [11, 19]. Or roughly speaking, $R[A]$ is around 4, for large m.
- For ECT, $R[ECT] \geq 30/13$ for all m and $R[ECT] = 30/13$ for $m = 2, 3, 4$. See [6].

The proof of the upper bound for ECT for $m \geq 5$ is not done, although we suspect that it matches the establish lower bound of $30/13$.

5. Conclusions

In this paper, we study the validity of our overhead model that is used to estimate the execution time of a malleable job when assigned to multiple processors and incorporates both the computation speedup and the slowdown due to overhead caused by parallel processing. Experiments conducted by running well-known parallel benchmarks produce real-time execution times of the benchmarks running on various numbers of processors in a state-of-art parallel system established in the Jefferson Lab. These experimental data are then fitted using the overhead model and the traditional linear model. The comparison of the fitting results for the two models shows that our overhead model has advantages over the linear model in that the former more closely predicts the behavior of the experimental data than the latter. We also summarize the theoretical results for online algorithms designed for a parallel job scheduling problem defined using the overhead model.

Future research may focus on studying variations of the overhead model, such as using a different overhead constant c_j to replace c for each job J_j. On the algorithm side, new ideas for scheduling algorithms may be desired and a tight bound for the competitive ratio for ECT is needed.

Acknowledgment

This work was supported in part by the United States Department of Energy, Contract DE-AC05-84ER40150.

We also thank the anonymous referees for their helpful comments.

References

[1] A. Allahverdi, C. T. Ng, T. C. E. Cheng, M. Y. Kovalyov, A survey of scheduling problems with setup times or costs, *European Journal of Operational Research*, to appear in 2008.

[2] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagnum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, The NAS parallel benchmarks, *International Journal of Supercomputer Applications*, 5, 63-73, 1994.

[3] R. Barrett, M. Barry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, H. Vosrt, *Templates for the solution of linear system: Building blocks for iterative methods*, 2nd edition, SIAM publications, 1996.

[4] J. Blazewicz, M. Kovalyov, M. Machowiak, D. Trystram, and J. Weglarz, Preemptable malleable task scheduling problem, *IEEE Transactions on Computers*, 55, 486-490, 2006.

[5] J. Blazewicz, M. Machowiak, G. Mounie, and D. Trystam, Approximation algorithms for scheduling independent malleable tasks, *Proceedings of the European Conference on Parallel and Distributed*, 191-197, 2001.

[6] R. Dutton and W. Mao, Online scheduling of malleable parallel jobs, *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems*, 1-6, 2007.

[7] J. Du and J.T. Leung, Complexity of scheduling parallel task systems, *SIAM Journal on Discrete Mathematics*, 2, 473-487, 1989.

[8] A. Faraj and X. Yuan, Communication characteristics in the NAS parallel benchmarks, *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems*, 729-734, 2002.

[9] D. Feitelson, L. Rudolph, U. Schwiegelshohn, K. Sevcik, and P. Wong, Theory and practice in parallel job scheduling, *Proceedings of the Workshop for Job Scheduling Strategies for Parallel Processing*, 1-34, 1997.

[10] Y. Grigoryev, V. Vshivkov, and M. Fedoruk, *Numerical Particle-in-Cell Methods: Theory and Applications*, Brill Academic Publishers, 2002.

[11] J. Havill and W. Mao, Competitive online scheduling of perfectly malleable jobs with setup times, *European Journal of Operational Research*, to appear in 2008.

[12] J. Havill, W. Mao, and V. Dimitrov, Improved parallel job scheduling with overhead, *Proceedings of the Seventh Joint Conference on Information Sciences*, 393-396, 2003.

[13] K. Jansen and H. Zhang, Scheduling malleable tasks with precedence constraints, *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, 86-95, 2005.

[14] 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, Vol. 4: Logistics of Production and Inventory*, edited by So. C. Graves, A. H. Rinnooy Kan, and P. Zipkin, North-Holland, 1990.

[15] B. Johannes, Scheduling parallel jobs to minimize the makespan, *Journal of Scheduling*, 9, 433-452, 2006.

[16] R. Karp, On-line algorithms versus off-line algorithms: How much is it worth to know the future? *Technical Report TR-92-044*, International Computer Science Institute at Berkeley, 1992.

[17] A. Laroy, Parallel run time models for real machines, *Masters Research Project: The College of William and Mary*, 2001.

[18] W. Ludwig and P. Tiwari, Scheduling malleable and nonmalleable parallel tasks, *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, 167-176, 1994.

[19] W. Mao, J. Chen, and W. Watson, On-line algorithms for parallel job scheduling problem, *Proceedings of the IASTED International Conference on Parallel and Distributed Computing Systems*, 753-757, 1999.

[20] G. Mounie, C. Rapine, and D. Trystam, Efficient approximation algorithms for scheduling malleable tasks, *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, 23-32, 1999.

[21] C. Phillips, C. Stein, and J. Wein, Task scheduling in networks, *Algorithm Theory–SWAT 94*, edited by E. Schmidt and S. Skyum, Lecture Notes in Computer Science, Springer Verlag, 1994.

[22] Y. Saad, Communication Complexity of the Gaussian Elimination Algorithm on Multiprocessors, In *Linear Algebra and Its Applications*, 77, 315-340, 1986.

[23] D. Shmoys, J. Wein, and D. Williamson, Scheduling parallel machines on-line, *SIAM Journal on Computing*, 24, 1313–1333, 1995.

[24] J. Sgall, On-line scheduling of parallel jobs, *Proceedings of the International Symposium on Mathematical Foundations of Computer Science*, LNCS 841, 159–176, 1994.

[25] J. Sgall, On-line scheduling – a survey, in *On-line Algorithms*, edited by A. Fiat and G. Woeginger, Springer-Verlag, 1997.

[26] K. Sevcik, Application scheduling processor allocation in multiprogrammed parallel processing systems, *Performance Evaluation*, 19, 107-140, 1994.

[27] J. Turek, J. Wolf, and P. Yu, Approximate algorithms for scheduling parallelizable tasks, *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, 323-332, 1992.

[28] http://icl.cs.utk.edu/hpcc/, HPC challenge.

[29] http://www.netlib.org/benchmark/hpl/, HPL: A portable implementation of the high-performance Linpack benchmark for distributed-memory computers.

[30] http://www.netlib.org/linpack/, LINPACK.

[31] http://mvapich.cse.ohio-state.edu/, MVAPICH: MPI over Infiniband and iWARP.

[32] http://www.top500.org/, Top 500 supercomputing sites.