Job Scheduling Techniques for Distributed Systems with Heterogeneous Processor Cardinality

Hung-Jui Chang
Department of Computer Science and Information Engineering
National Taiwan University
Taipei, Taiwan

Jan-Jan Wu
Institute of Information Science
Academia Sinica
Nankang, Taiwan

Pangfeng Liu
Department of Computer Science and Information Engineering
Graduate Institute of Networking and Multimedia
National Taiwan University
Taipei, Taiwan

Abstract

This paper proposes scheduling algorithms for assigning jobs with different release time and execution time, to machines with heterogeneous processor cardinality. We show that this scheduling problem is NP-complete, and propose dynamic programming to find the optimal schedules. Since the dynamic programming is time-consuming we propose techniques that improve the efficiency of the dynamic programming. We also propose heuristic algorithms for this scheduling problem. Experimental results suggest that some of the heuristics not only compute the answer efficiently but also provide good solution.

1 Introduction

As grid systems become increasingly popular, the vast amount of computing and storage resources allow us to dispatch large scale resource-demanding problems to remote servers so that the final solution can be found within a reasonable amount of time. Grid systems are widely adopted by scientists in various research areas. For example, the Worldwide Large Hadron Collider Computing Grid [9] builds and maintains a data storage and analysis infrastructure for the entire high energy physics community. Other organizations like the Biomedical Informatics Research Network [2] provides a platform for biomedical science sharing data and computing resources.

Grid systems often consist of different computing sites with heterogeneous resources, which make it difficult to schedule tasks to run efficiently. Due to the heterogeneous nature, grid systems are often built across the wide area network. As a result, the communication latencies between sites in grid systems are usually high. High communication latencies have significant impact to the efficiency of data transfer. As a result job must be assigned to only one machine instead of run across many machines to reduce the transfer latency.

In this paper, we are interested in scheduling jobs that require a given number of processors. We are given a set of tasks to schedule so that a job will only be assigned to a machine where the number of available processors is enough to run the job. The objective is to develop algorithms that minimizes the total makespan of job execution.

The rest of the paper is organized as follows. Section 2 reviews related works on scheduling jobs on multiprocessors. Section 3 describes the system model of scheduling. Section 4 introduces the dynamic programing, and propose efficient heuristic algorithms. Section 5 examines the experimental results from our heuristics and compare them with the optimal solution found by the dynamic programing program. Section 6 concludes and and discusses future works.

2 Related Works

Scheduling is a key issue on parallel and distributed systems. In order to improve system performance we need to schedule jobs to run at the right place at the right time. In particular, when job compete for resources that are not preemptive, it is very import to interleave their usage on these resources so that system utilization and quality of service are maintained at the same time.

Scheduling is a very difficult problem in parallel and dis-

tributed systems. General scheduling problems on multiprocessor system has been showed to be NP-complete [12]. Even if there are only two processors with identical computing capacity and jobs do not share data, the problem is still NP-complete because it is a special case of 2-Partition problem [5].

There are many different objective functions in scheduling. Some researches focus on *average completion time* of all jobs [3, 8]. Some researchers focus on *total completion time* [3, 7]. However, most of the research [1, 4, 6, 10] focus on the makespan – the completion time of the last job. As a result we will focus on the makespan in this paper.

There have been many research results in minimizing makespan for multi-processors. Garey and Graham [6] give an approximation algorithm with competitive ratio $2 - \frac{1}{m}$ in an environment where jobs are independent and require only one processor. Albers improves the competitive ratio to 1.923 in [1]. Naroska and Schwiegelshohn [10] show that the same bound in [1] holds when the jobs need more than one processor. Dell'Amico et al. [4] summarize lower and upper bounds on scheduling jobs on multiprocessors. Schwiegelshohn et al. [11] suggest a model in which there are many machines, and each machine has a number of processors. They show that if jobs cannot run across machine, then to find an approximation algorithm with competitive ratio less than 2 is also NP-complete. This result implies that it is unlikely one can find approximation algorithm for the scheduling problem on multi-processors.

3 Model and Problem Definition

A grid system M has $|M| = m$ machines and each machine $M_i \in M$ has p_i processors. We denote P_i as the set of processors in M_i and $P_{i,j}$ as the j-th processor of machine M_i. Without lose of generality the machines are ordered so that $m_{i-1} \leq m_i$ for all $M_i \in M$. All processors in the system are identical.

A job set J has n independent jobs. A job J_j has three attributes – a processing time t_j, a release time r_j, and a degree of parallelism s_j. The processing time is the amount of time for the job to finish. The release time is the time for a job to be ready for execution. The degree of parallelism is the number of processors that must be exclusively allocated to the job during its processing. Note that we do not allow multi-site scheduling or preemption, therefore a job J_j must be executed on s_j processors on one machine without interruption.

A schedule for a grid system M to run a job set J contains two attributes for each job – the machine a job is assigned to and the starting time of the job. Formally for a given schedule S we use $a_{S,j}$ to denote the machine job J_j is assigned to, and $r_{S,j}$ denotes the starting time of job J_j.

A schedule S is feasible if all following conditions hold. The first condition is that a job cannot start before its release time. The second condition is that at any time t and on any machine M_i, the total number of processors required by those jobs running on M_i. Those jobs J_j that are assigned to machine M_i ($a_{S,j} = M_i$) and are running at time t ($r_{S,j} \leq t < r_{S,j} + t_j$), use at most p_i processors, where p_i is the number of processors in machine M_i.

1. $r_{S,j} \geq r_j$, for all $J_j \in J$.
2. $\sum_{\{J_j | a_{S,j}=M_i, r_{S,j} \leq t < r_{S,j}+t_j\}} s_j \leq p_i$, for all $M_i \in M$, $J_j \in J$, $t \geq 0$.

For a given grid system M and a job set J, we want to find a schedule S to minimize the makespan. The makespan of a schedule S, denoted by $C(S)$, is the maximum completion time over all job $J_j \in J$ in schedule S.

4 Algorithms

It is easy to see that the job scheduling problem mentioned earlier is NP-complete. We can have two machines, each has one processor. The parallelism of every job is 1, so that every job can run on either machine. The goal is to divide the jobs into two groups so that total execution times of the two groups are the same. This is exactly the 2-partition problem. Since 2-partition is a special case of the job scheduling problem, the scheduling problem is NP-complete.

Since the problem is NP-complete, it is unlikely to find an efficient algorithm that finds the optimal schedule. Instead we will use a dynamic programing to find the optimal schedule. Also in order to enhance the efficiency of the dynamic programming, we propose techniques to speed up the dynamic programming.

4.1 Schedule

Our algorithm constructs the job schedule in phases, and in each phase the algorithm schedules a job and adds it into the current "partial" schedule. We will use the *current schedule* to denote the partial schedule up to this point, and we use J_S to denote the set of jobs that have already been scheduled by this current (and partial) schedule S.

The *ready* time of a processor p is the time that a new job J_2 can run on p under the current schedule S. If job J_1 is last running job on processor p according to S, J_2 has to wait until J_1 completes, and the ready time of J_2 will be the completion time of J_1. Initially the ready time of every processor p is 0 since the initial schedule is empty and there is no job running on p.

Let $R_{S,i,j}$ to denote the ready time of the j-th processor in machine M_i, i.e., $P_{i,j}$. under current schedule S. Again for ease of notation we will drop the S subscript when the

context clearly indicate the schedule S. As a result the *ready time sequence* $R(S)$ of a schedule S is defined as $R(S) = ((R_{1,1}, \ldots, R_{1,p_1}), \ldots, (R_{m,1}, \ldots, R_{m,p_m}))$. Note that the ready time sequence has m vectors and each of them is for a machine, and the i-th vector has p_i components, i.e., the number of processors in machine M_i.

4.2 Dynamic Programing

We now define the table element in our dynamic programming. Let R be a ready time sequence for machines M, and J' be a subset of J. We define $E(R, J')$ to be the *minimum* makespan when the ready time sequence is R and we have J' to schedule. Therefore the optimal makespan of a given job set J for a machine M will be $E(((0, \ldots, 0), \ldots, (0, \ldots, 0)), J)$, where the ready time of every processor is 0, and we have the entire job set J to schedule.

For ease of description we will define a function that gives all possible ways to pick a given number of processors from a machine. For example, If the number of processors required by J_j (s_j) is two, and we would like to run it on the i-th machine with $p_i = 4$ processors, there will $\binom{4}{2} = 6$ different ways to choose processors from M_i. We can choose from one of the following – $\{P_{i,1}, P_{i,2}\}$, $\{P_{i,1}, P_{i,3}\}$, $\{P_{i,1}, P_{i,4}\}$, $\{P_{i,2}, P_{i,3}\}$, $\{P_{i,2}, P_{i,4}\}$, and $\{P_{i,3}, P_{i,4}\}$. Formally we use $A(M_i, J_j)$ to denote the set of all possible processors sets from M_i where each subset has s_j processors. Note that $A(M_i, J_j)$ is empty if the number of requested processors s_j is larger than the number of processors in machine M_i.

$$A(M_i, J_j) = \begin{cases} \{p | p \subseteq P_i, |p| = s_j\}, & \text{if } p_i \geq s_j \\ \varnothing, & \text{if } p_i < s_j \end{cases} \quad (1)$$

We now describe how a given ready time sequence R could change if we schedule a new job J_j to a machine M_i. For ease of notation let R_i be the i-th vector of R, i.e. the ready time of processors in machine M_i, and $R_{i,j}$ be j-th component of R_i, i.e., the ready time of $P_{i,j}$.

If we assign J_j to a machine M_i using one of the assignment A from $A(M_i, J_j)$, then only the processor in A will be affected. The starting time of J_j will be $\max(r_j, \max_{k \in A} R_{i,k})$, i.e., the maximum of the release time of job J_j (r_j), and the maximum of the ready time of processors in A, the set of processor in M_i that we choose to run J_j. Consequently the completion time of J_j is $\max(r_j, \max_{k \in A} R_{i,k}) + t_j$, which will become the new ready time for processors in A.

We use $T(R, J_j, M_i, A) = R'$ to denote the result of assigning J_j to a machine M_i using one of the assignment A from $A(M_i, J_j)$ based on the ready time sequence R. That is, R and R' are the ready time sequence before and after the scheduling respectively. The definition of R' is as follows. Note that if a processor is not in the assignment A its ready time will not be affected.

$$R'_{i,j} = \begin{cases} R_{i,j}, & \text{if } P_{i,j} \notin A \\ \max(r_j, \max_{P_{i,k} \in A} R_{i,k}) + t_j, & \text{if } P_{i,j} \in A \end{cases} \quad (2)$$

Now we are ready to describe the recursive formula for the optimal makespan function $E(R, J)$ in Equation 3. We consider all jobs J_j in J and all machines M_i in M, and all possible processor assignments for running J_j on M_i, and choose one that will in term has the minimum makespan.

$$E(R, J) = \min_{J_j \in J} \min_{M_i \in M} \min_{A \in A(M_i, J_j)} E(T(R, J_j, M_i, A), J - J_j) \quad (3)$$

The terminal condition for $E(R, J)$ is when J is empty, i.e., we have nothing to schedule. Thus the value of $E(R, \emptyset)$ is the maximum ready time within the ready sequence R.

4.3 Improvements

The dynamic programing we proposed considers all job scheduling orders and the ways to choose processors within a machine, therefore it is very time consuming. In order to speed up the dynamic programming we derive efficiency improving techniques that are based on simple observations on the scheduling problem.

For the ease of notation we sort all elements in each vector R_i of the ready time sequence $R(S)$ in decreasing order, that is, $R_{i,j} \leq R_{i,j+1}$ for all $R_{i,j} \in R_i$. Since we assume that all processors are homogeneous, there will be no difference if the ready time vector of a machine is $(3, 4, 1)$ or $(1, 3, 4)$. We also use $P_{S,i,j}$ to denote the corresponding processor with ready time $R_{S,i,j}$, i.e., the processor with j-th smallest ready time in M_i. The purpose of this "normalization" of ready time is to compare two ready time vectors as in the following definition.

Definition 1. *A schedule S_1 is* better *than another schedule S_2 if each element in the ready time sequence under S_1 ($R(S_1)$) is no more than the corresponding element of ready time sequence $R(S_2)$. That is, $R_{S_1,i,j} \leq R_{S_2,i,j}$, for all $1 \leq i \leq m$, and $1 \leq j \leq p_i$.*

We observe the following if a a schedule S_1 is *better* than another schedule S_2.

Observation 1. *First the current makespan of S_1 will not be larger than the current makespan of S_2. Second if we schedule the same set of remaining jobs in the same order for both schedule S_1 and S_2, it is always possible to schedule the remaining jobs so that the makespan from S_1 will be no more than the makespan from S_2.*

Now we can describe our first technique to speed up the dynamic programming. In the original dynamic programming program, when we assign a job required s processors into a machine of p processors, we need to consider *all* $\binom{p}{s}$ ways to chose processors. In the following Lemma 1 shows that we only need to consider $p - s + 1$ ways that choose processors with consecutive index.

Lemma 1. *Let S be a schedule S, $\mathcal{R}$ be the set of all possible $\binom{p_i}{s_j}$ ready time sequences that start with S and then schedule J_j to machine M_i, and $\mathcal{R}'$ be the set of all possible $p_i - s_j + 1$ ready time sequences that start with S and then schedule J_j to machine M_i with* consecutive *processor index according to increasing ready time. The minimum makespan of schedules that follow $\mathcal{R}'$ will not be longer than the minimum makespan of schedules that follow $\mathcal{R}$.*

Proof. We consider any assignment A among $\binom{p_i}{s_j}$ possibilities within $\mathcal{R}$, and argue that there must be another assignment A' in $\mathcal{R}'$ that is as good as A. Suppose the assignment A choose s_j processors as $(P_{i,a_1}, \ldots, P_{i,a_{s_j}})$ in non-decreasing ready time order, then the assignment A' will choose $(P_{i,a_{s_j}-s_j+1}, \ldots, P_{i,a_{s_j}})$ instead. For example, if s_j is 3 and $a_1 = 3$, $a_2 = 4$ and $a_3 = 7$, then A' will choose $(P_{i,5}, P_{i,6}, P_{i,7})$ instead.

We will make a series of transformations to convert A into A' without delaying the schedule. Consider the last "hole" in the a_k sequence, that is, the largest k such that there exists an integer s such that $a_k > s > a_{k-1}$. Now we adjust the sequence so that new a_{k-1} is now $a_k - 1$. For example in our previous case when a is $(3, 4, 7)$ we now make it $(3, 6, 7)$. The ready time of P_{i,a_k-1} increases from R_{i,a_k-1} to $R_{i,a_k} + t_j$. Recall that t_j is the execution time of job J_j. However the ready time of $P_{i,a_{k-1}}$ decreases from o $R_{i,a_k} + t_j$ to $R_{i,a_{k-1}}$. Since the ready time $R_{i,a_{k-1}}$ will never be later than R_{i,a_k-1} because of the "hole", and the fact that the ready time is sorted. We conclude that changing a_{k-1} to $a_k - 1$ will not delay anything. We can repeatedly perform this switching until there is no "hole" within the series a_k and finally changing A into A' without delaying anything.

For any assignment A we can always find a A' so that the schedule is better. i.e., without delaying the ready time of any processor. From Observation 1 we conclude that by using only $\mathcal{R}'$, i.e., considering only processors that have consecutive processor ids according to ready time, we will still be able to find optimal schedule that starts from S and assigns J_i to M_i. □

With Lemma 1 in place we only need to consider processors with consecutive processor index while selecting processors for a job. This will significantly improve the efficiency of dynamic programming since $\binom{p}{s}$ is much larger than $p - s + 1$.

The second possible improvement is how to order jobs that require the same number of processors. Lemma 2 establishes that if all the processors' ready time are greater than the jobs' release time, we only need to consider scheduling those jobs with the increasing order of processing time.

Lemma 2. *Given a schedule S, if J_1 and J_2 are two jobs that require the same number of processors but not yet scheduled, the release time of both J_1 and J_2 is earlier than the ready time of all processors, and the processing time of J_1 is at most the processing time of J_2, i.e., $t_1 \leq t_2$. Then the minimum makespan of starting from S, then scheduling J_1, then scheduling J_2, is no longer than the minimum makespan of starting from S, then scheduling J_2, then scheduling J_1.*

Proof. For a given schedule S and two unscheduled jobs J_1 and J_2 with $s = s_1 = s_2$, $t_1 \leq t_2$ and $\max(r_1, r_2) \leq R_{S,i,j}$, for all machine $M_i \in M$, and processor $P_{S,i,j} \in M_i$, we want to show that for every schedule S_{21} from the set of all schedules that start from S, then assign job J_2 then J_1, there exists another schedule S_{12} from the set of schedules that start from S, then assign J_1, then J_2, and S_{12} is better than S_{21}.

We observe that we need only consider the case when J_1 and J_2 share processors at a machine M_i. If J_1 and J_2 were assigned to different machines, or to the same machine but different processors, then it does not make any difference which job is scheduled first. Figure 1 illustrates the case when J_1 and J_2 share processors at a machine M_k.

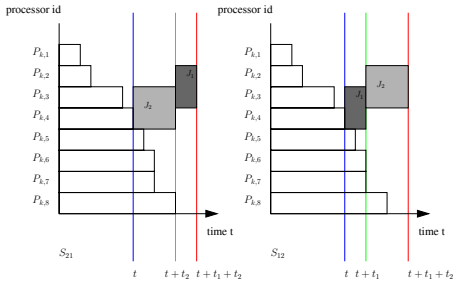

Figure 1. S_{21} **and** S_{12}

From Figure 1 we can easily see that schedule S_{12} is better than S_{21}. For those processors that are not allocated to either J_1 or J_2 the ready time will not change. Let P denote the set of processors that are allocated to either J_1 or J_2. There will be s processors in P now have new ready time $t + t_1 + t_2$, where t is the maximum ready time of processors in P. The other $|P| - s$ processors now have new ready time $t + t_2$ in S_{21}, and $t + t_1$ in S_{12}. Since $t_1 \leq t_2$, S_{12} is better than S_{21}. The lemma follows. □

We can easily generalize Lemma 2 to cases when the

number of unscheduled jobs with the same number of processors is greater than two. The proof is similar and we change the schedule one job at a time.

4.4 Heuristic Algorithms

We propose a earliest-completion-first greedy heuristic to schedule jobs. The algorithm always chooses the job with the minimum completion time. Let J_u be the set of unscheduled job. Initially J_u has all the jobs. We compute the minimum possible completion time of each unscheduled job $J_j \in J_u$, and schedule the one with smallest minimum possible completion time. The chosen job is assigned to the processors according to its minimum possible completion time, and then is remove from J_u. The algorithm finishes when J_u becomes empty.

If there are more than one job that has the earliest completion time, or one job has more than one processor assignment that has the earliest completion time, the algorithm picks the job that requires the maximum/minimum number of processors, and the machine that has the maximum/minimum number of processors. These four combinations will be referred to as *min-min*, *min-max*, *max-min*, and *max-min* heuristics.

5 Experimental Results

The experimental environment is as follow. There are 3 machine in the system – machine M_1 has a processor, machine M_2 has two processors, and machine M_3 has four processors. The number of jobs is from 1 to 17. The processing time of a job is from 1 to 30, and the release time is from 0 to 20. Each job requires 1 to 4 processors. For every parameter combination we run the simulation for 100 times and calculate the average.

We use *relative makespan* to measure the performance of heuristic algorithms. The *relative makespan* of a heuristic algorithm is the makespan of the heuristic algorithm divided by the optimal makespan for the problem instance. We are able to determine the optimal makespan because of our dynamic programming. The heuristic algorithm performs best when its relative makespan is close to 1.

All-Zero-Release-Time We first consider a case when all jobs have release time 0. We will refer to this case as *all-zero-release-time* case. In the all-zero-release-time model all jobs can be scheduled right from the beginning. Also note that when a large set of of computation-intensive jobs are submitted into the system, the jobs will be waiting for the previous jobs to finish before it can start. In that case the schedule will be similar to the all-zero-release-time model since the jobs will be waiting in queue, and will be ready as soon as processors become available.

Figure 2 shows the relative makespan when all jobs requires only one processor in the all-zero-release-time case. We consider three cases –the job processing time is from 1 to 10, 10 to 20, and 20 to 30. Notice that when every job requires one processor, the four heuristic algorithms will be the same. The relative makespan is about 1.3 for the case of processing time from 1 to 10, 1.1 to 1.35 for the case of processing time from 10 to 20, and 1.05 to 1.17 for the case of processing time from 20 to 30.

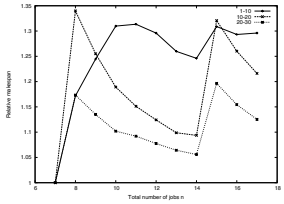

Figure 2. Relative makespan in all-zero-release-time model

When jobs require different number of processors the four heuristics behave differently. Figure 3 (a) compares the relative makespan of the four heuristics when jobs require 1 to 2 processors, and (b) compares the relative makespan when jobs require 1 to 4 processors. The processing time of jobs in both experiments is from 10 to 20. In both two sets of experiments the max-min heuristic performs best.

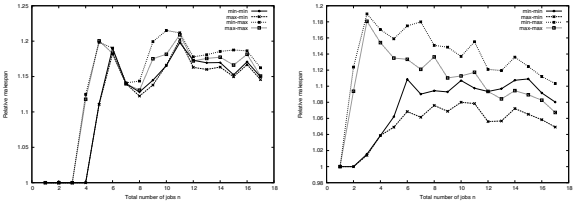

Figure 3. All-zero-release-time model when jobs require (a) 1 to 2 processors (b) 1 to 4 processors

Non-Zero-Release-Time We now consider the case when the release time of jobs are not zero. Since jobs are submitted into the system only when they are released, therefore the heuristic algorithms will know the information of job only after it has been released. We will refer to this model as *non-zero-release-time* model.

Figure 4 compares the relative makespan between all-zero-release-time and non-zero-release-time. The setting is that all jobs require one processor only, the processing time of jobs is from 10 to 20, and the release time of jobs is from 0 to 10. We observe that the relative makespan is more sta-

ble in non-zero-release-time model than in all-zero-release-time model. When the number of jobs is slightly more than the number of processors, usually the optimal schedule is to place shortest jobs into the *same* processor. However, since the heuristics use earliest-completion-first criteria, they will place the shorter jobs into *different* processors. This misjudgement will become more often when *more* shortest jobs are available to schedule. Unfortunately this is the case when all jobs have release time 0, i.e., all-zero-release-time model. That is, in all-zero-release-time model the scheduler is more likely to place shortest jobs into different processor by ECF, since they are available right at the beginning.

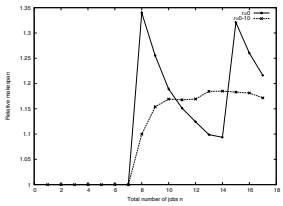

Figure 4. Comparison of all/non-zero-release-time models

Figure 5 illustrate the relative makespan under different release time, where (a) has release time from 0 to 10 and (b) has release time from 0 to 20. The number of processors per job is from 1 to 4 and the processing time of jobs is from 10 to 20. In both sets of release time the max-min heuristics still performs best. We also observe that the relative makespan decreases when the release time becomes later. The reason is that when jobs have later release time, some processors will start their first job later and the makespan will increase. Since the optimal makespan increases, the relative makespan becomes smaller when the release time becomes later.

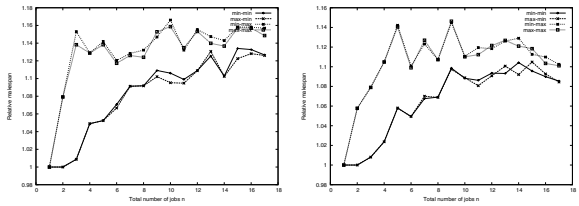

Figure 5. Non-zero-release-time model when release time is (a) from 0 to 10 (b) from 0 to 20

6 Conclusion

This paper proposes scheduling algorithms for assigning jobs with different release time and execution time, to machines with heterogeneous processor cardinality. We show that this scheduling problem is NP-complete, and propose dynamic programming to find the optimal schedules. Since the dynamic programming is time-consuming we propose techniques that improve the efficiency of the dynamic programming. We also propose heuristic algorithms for this scheduling problem. Experimental results suggest that some of the heuristics not only compute the answer efficiently but also provide good solution.

We will continue the investigation on efficient scheduling algorithms for this problem. One of the ongoing work is to examine the possibility of finding good algorithms when the numbers of processors in every machine are special numbers, e.g., a powers of 2. It seems more likely to find efficient solution if we further assume that the number of processors for every job is also a power of 2. Also we would like to consider the case if the execution time of many jobs are the same. We might be able to take advantage of the constraint on the input and derive efficient algorithms.

References

[1] S. Albers. Better bounds for online scheduling. In *STOC*, pages 130–139, 1997.

[2] BIRN: The Biomedical Informatics Research Network. http://www.nbirn.net/.

[3] C. Chekuri, R. Motwani, B. Natarajan, and C. Stien. Approximation techniques for average completion time scheduling. In *SODA*, pages 609–618, 1997.

[4] M. Dell'Amico, M. Iori, and S. Martello. Heuristic algorithms and scatter search for the cardinality constrained p||c_{max} problem. *Journal of Heuristics*, 10(2):169–204, 2004.

[5] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. WH Freeman and Co. New York, NY, USA., 1979.

[6] M. R. Garey and R. L. Graham. Bounds for multiprocessor scheduling with resource constraints. *SIAM Journal on Computing*, 4(2):187–200, 1975.

[7] Y. Guan, W. Xiao, R. K. Cheung, and C. Li. A multiprocessor task scheduling model for berth allocation: heuristic and worst-case analysis. *Operations Research Letters*, 30(5):343–350, 2002.

[8] L. A. Hall, D. B. Shmoys, and J. Wein. Scheduling to minimize average completion time: Off-line and on-line algorithms. In *SODA*, pages 142–151, 1996.

[9] LCG: LHC Computing Grid. http://lcg.web.cern.ch/LCG/.

[10] E. Naroska and U. Schwiegelshohn. On an on-line scheduling problem for parallel jobs. *Information Processing Letters*, 81(6):297–304, 2002.

[11] R. Y. U. Schwiegelshohn, A. Tchernykh. Online scheduling in grids. In *22nd IEEE International Symposium on Parallel and Distributed Processing*, pages 1–10, 2008.

[12] J. D. Ullman. Np-complete scheduling problems. *Journal of Computer and System Sciences*, 10(3):384–393, 1975.