Approximation of Repeated Scheduling Chains of Independent Jobs of Unknown Length Based on Historical Data

RICHÁRD KÁPOLNAI, IMRE SZEBERÉNYI, BALÁZS GOLDSCHMIDT
Budapest University of Technology and Economics
Department of Control Engineering and Information Technology
Magyar tudósok körútja 2., 1117 Budapest
HUNGARY
kapolnai@iit.bme.hu

Abstract: We consider the problem of minimizing the makespan when scheduling a *Parameter Sweep Application (PSA, set of independent jobs)* on identical machines of a parallel computational infrastructure. However, there is no *a priori* information on the job lengths, and the user intends to execute the whole PSA *multiple times* and is able to measure individual machine completion times. We also require that any set of jobs assigned to a machine has to be briefly described so an arbitrary schedule may not be suitable.
This paper proposes an iterative framework which repeats computing an approximation schedule, executing the PSA and updating the historical database according to the machine completion times. After each iteration, the approximation algorithm further improves some upper bound of the makespan until a 2-approximation is reached. The scheduling algorithm always assigns consecutive jobs called *chains* to machines keeping the historical database and the machine assignment descriptions brief.

Key–Words: PSA, scheduling, workload balancing, chain partitioning, approximation, historical data, uncertainty

1 Introduction

We investigate the problem of scheduling a *Parameter Sweep Application (PSA)* on a parallel computational infrastructure. In a PSA, the same computation has to be executed for many different values of a parameter, which is common in engineering or scientific computations. To each parameter value, we assign a non-interruptible *job*. We call the domain of all possible parameter values the *parameter space*, corresponding to the set of jobs. Every job requires a *processing time* to complete called its *length*, and a machine's *completion time* is the total length of jobs assigned to it. We consider the cost of a schedule the *makespan*, i.e. the maximal completion time. Hence the underlying problem is the widely known NP-hard optimization problem of scheduling independent jobs to identical parallel machines while minimizing the makespan, often denoted as $P||C_{\max}$, except that we have no *a priori* knowledge of the job lengths. To maintain tractability, we focus on a 2-approximation.

Our inspiration was a supposedly common usage scenario. When a user develops a PSA for a project, sometimes executes it several times during the project, as observed by others [8], because of different reasons: testing, fixing bugs, adding new features such as more detailed output, improved precision etc. To exploit a parallel infrastructure, the user has to partition somehow the parameter space by creating a *schedule*, i.e. a mapping from the parameter space to the set of machines. We call the work of a machine, which is the set parameters assigned to it, an *assignment*. In many PSAs, the processing times of the parameters may be significantly different and hard to predict, so the first partition may result in a makespan far from optimal. So after the execution, assuming the measured machine completion times are available, the user may notice that some assignments took extremely short or long time to complete. To minimize the makespan, the user should adjust the partition intuitively or systematically in order to shorten the longest assignments at the next execution. If the makespan still turns out to be too high, further adjustments are made until the makespan meets the requirements.

Our most essential assumption is that most users tend to repeat executing their PSAs. Naturally not to optimize the makespan, but for various reasons such as executing later a revised version of PSA. This kind of user behaviour is assumed throughout this paper:

Conjecture 1. *The user submits for execution a PSA multiple times.*

It also is important to notice that the process above is validated by another strong assumption: the parameter space and processing times of the parameters remain constant between the executions, nevertheless, the user may do changes to the PSA (e.g. fix a program bug).

The main goal of this paper is proposing a generic, iterative framework to help automating this process. In each iteration the framework updates its historical database with the measured completion times of the execution of the previously computed schedule. If it is not clear whether the approximation ratio of 2 has already been reached, then splits the long assignments into parts estimating proportionally their lengths, contracts the small ones, and re-computes the schedule for the next execution. We state that, under all assumptions above, after a finite number of iterations the approximation ratio of 2 is always reached for any PSA. For a demonstration, we implemented a preliminary version of the framework as a Saleve client [9]. The demonstration includes a simple synthetic example as well as a real, previously published PSA. We believe this framework can be incorporated into existing PSA execution supporting tools or even scheduling techniques.

According to Conjecture 1, the PSA is not repeated because our framework needs more iteration. It is repeated regardless of the framework because this is the wish of the user. So the framework does not cause additional execution costs, instead it gradually decreases the execution costs.

To keep in sight easy implementation and small historical database, let us suppose the parameter space is an ordered set. Our scheduling algorithm only assigns a set of consequtive jobs called a *chain* to any machine, which keeps the machine assignment descriptions brief so the historical database. So in a nutshell, based on historical data, the framework splits the long chains and contracts the small ones in each iteration.

The approximation ratio of 2 cannot be further improved unless we give up our restriction of scheduling only chains of jobs. Hence we say, inspired by the "price of anarchy" [20], the *price of chaining* is 2.

Due to space limitations, the proofs of our statements have been omitted from this paper.

2 Related Work

The classical scheduling problem $P||C_{\max}$ has received much attention through the past decades. There are simple 2-approximation algorithms, e.g. list scheduling [11] or rounding a fractional schedule [1]. However, these algorithms cannot be used in the iterations of the framework as shown in Section 4, so we present yet another 2-approximation for $P||C_{\max}$.

Scheduling chains is usually referred to as *one dimensional array partitioning* or *chain partitioning*. In our setting, the optimal chain partitioning can be found in polynomial time [4], and exact solutions are surveyed in e.g. [21]. Despite the efficiency of exact solutions, we are not interested in them because even an optimal chain partitioning is just another 2-approximation for the original problem $P||C_{\max}$.

There are a number scheduling techniques when uncertainty arises. Some surveys characterize techniques as proactive (more robust to unplanned changes) and reactive (less committed to existing plan) [13, 2]. For instance the *online* list scheduling, also known as self-scheduling is reactive. While self-scheduling is a good approximation in theory, obviously effects an enormous overhead in a system. Guided self-scheduling and factoring algorithms offer a compromise: they form chunks of jobs (similar to our chains) to reduce the overhead. Initally these algorithms dispatch large chunks, then the chunk size will gradually decrease to improve balancing [15].

However, reactive behaviour usually implies the algorithm intervenes *during* the execution of the schedule based on some quasi real-time feedback of the system state, aiming to improve *one* schedule. Instead, our long-term goal is simpler: we aim to collect historical data to improve the next schedules. We note that getting real-time feedback may be very expensive in distributed systems. Our framework uses only a historical database which is not real-time, thus we believe it is more generally applicable and easier to implement.

A notable idea to deal with uncertainty in jobs is requiring the users to provide the processing times [17]. AppLeS [3] and GrADS [6] require the user to provide explicit performance model of the application to predict processing times. Probabilistic estimation was also studied, assuming the lengths are independent random variables [7].

In work of others, the application properties are learnt by building a historical database from measurements from previous executions. Probabilistic estimation could be backed up by a database to estimate the average of the distribution [5]. A database can support machine learning techniques to automatically develop performance models from the application source code [23], or to predict processing times using records of *similar* jobs where similarity is calculated from job attributes e.g. user and job id, requested resources [19].

Some work focus on collecting processing times of the whole application [22, 12] in order to optimize resource performance such as utilization. Others, as well as this work, measure the parts of an application to optimize application running time [10, 14, 24, 18]. Some of them model the processing time to be depending on the input data size so a time can be predicted by projecting a previously measured time on the same resource for another data set (e.g. [18, 12]). In our case, we cannot use this prediction system for the chain lengths because then every job (or parameter) would have uniform processing time, contradicting to our setting. In [14] and [24], the goal is determining the optimal number of machines to achieve the desired speed-up and efficiency. MARS [10] collects both application-specific and system-specific information to balance loads, but it was designed for more general parallel applications than a PSA, and it does not exploits that the tasks may be divisible as our chains.

3 The Proposed Framework

In this section we formalize our objective and present the framework.

We wish to schedule n non-interruptible jobs of length $p_1, \ldots, p_n$ onto m identical machines. A schedule f is a mapping $f : [1 \mathrel{..} n] \mapsto [1 \mathrel{..} m]$, and we say f is a chain partitioning iff either $f(j+1) = f(j)$ or $f(j+1) = f(j)+1$ holds for all $1 \leq j < n$, so each assignment is a chain. A chain c is a subinterval within $[1 \mathrel{..} n]$, its cardinality is its *size*, its processing time is its *length*, denoted by $\ell(c)$. The completion time of machine i is the total length of its assignment: $C_i(f) = \sum_{f(j)=i} p_j$. A chain containing only one job is a *singleton*, its size is 1. Let $C^*_{\max}$ denote the makespan of the optimal schedule (NP-hard to compute), and $C^{1D*}_{\max}$ the makespan of the optimal chain partitioning (computable in polynomial time). We call the *price of chaining* the supremum of the quotient $C^{1D*}_{\max}/C^*_{\max}$ over all valid scheduling problems. To demonstrate the price of chaining is 2, we have

Example 1. *Let $n = 2m$, and the processing times $p_1 = \ldots = p_m = m$, $p_{m+1} = \ldots = p_{2m} = 1$.*

Clearly, for this input $C^*_{\max} = m+1$ and $C^{1D*}_{\max} = 2m$, so if the price of chaining exists, it is at least 2. On the other hand, as the 2-approximation framework presented by this paper is a chain partitioning as well, we have $C^{1D*}_{\max} \leq 2 \cdot C^*_{\max}$. We note that [1] also admits a 2-approximation chain partitioning.

According to our assumption on uncertainty, after executing a schedule f, all measured completion times $C(f)$ become known and can be used to optimize future schedules. $C(f)$ stands for the vector $C_1(f), \ldots, C_m(f)$. It is easy to see that any chain partition f can be unambiguously described in $O(m \log n)$ space.

The working of the framework is outlined in Algorithm 1, details are elaborated in this section. Given m, n, the historical database contains initially an arbitrary initial schedule f_0, and its measured, accurate completion times $C(f_0)$. f_0 can be obtained e.g. by splitting the parameter space into equally sized parts. In the q^{th} iteration, starting with $q = 1$, the schedule f_q has to be computed based on the database. After the execution of f_q, the measurements $C(f_q)$ are merged into the database.

Algorithm 1 *Iterative framework*

```
 1: procedure FRAMEWORK($m, n, f_0, C(f_0)$)
 2:     $q \leftarrow 0$
 3:     while $C_{\max}(f_q) > 2\max\{T_{\text{LB}}, s_{\max}\}$ do
 4:         $q \leftarrow q + 1$
 5:         $P$ ← a synthetic scheduling problem of
            the chains of the database and their length
 6:         for all chain $c$ in $P$ s.t. $\ell(c) > 2T_{\text{LB}}$ do
 7:             Split chain $c$ in half
 8:             Estimate the length of the half chains
 9:             Replace $c$ with the half chains in $P$
10:         $f_q \leftarrow$ FRUGALLYSCHEDULE($P$, $T_{\text{LB}}$)
11:         Execute schedule $f_q$, measure $C(f_q)$
12:         Merge $C(f_q)$ into the database
13:         Recalculate $T_{\text{LB}}$ and $s_{\max}$
```

We define *the chains of the database after merging $C(f_q)$* in line 12 as follows, which we rely in lines 5 and 13 on. A chain assignment c of f_q may or may not contain a chain c^* of estimated length as a subset. If it does not, then we say c is a chain of the database with is its measured length (some completion time). Otherwise it is easy to verify that $c \setminus c^*$ is also a chain. In addition, its length $\ell(c \setminus c^*)$ was already known as stated later by Proposition 2 (previously measured or calculated), so finally after executing f_q the *exact* length of c^* can be calculated from measurements: $\ell(c^*) = \ell(c) - \ell(c \setminus c^*)$. So in this case we say c^* and $c \setminus c^*$ are chains of the database. Hence for each assignment c, we have either one or two chains in the database, so the database consists of less than $2m$ chains.

The goal of the framework is to assure that $C_{\max} \leq 2C^*_{\max}$ after a finite number of iterations. For this purpose, two lower bounds on the optimal makespan $C^*_{\max}$ are determined in each iteration: the average completion time $T_{\text{LB}} := \sum_{j=1}^{n} p_j/m$ and the maximal length $s_{\max}$ of the singleton chains of the database, if there is any, otherwise $s_{\max} := 0$. The framework stops when $C_{\max}(f_q) \leq 2\max\{T_{\text{LB}}, s_{\max}\}$ is guaranteed. We note that we could omit the recalculation of T_{LB} in every iteration (line 13), but it provides the framework some adaptivity: if the processing times change because of some alteration in the PSA, the recalculation of T_{LB} restores the convergence.

In each iteration, a synthetic scheduling problem is prepared from the chains of the database after an adjustment: the longest chains (longer than $2T_{\mathrm{LB}}$) are split in half. The length of the half chains are unknown, so they are estimated temporarily as the half of the original length. Each chain corresponds to a synthetic job of the same length, where a synthetic length is either provided by the database or estimated as above. The synthetic jobs are scheduled by the algorithm called FRUGALLYSCHEDULE, presented in Algorithm 2, which is a simple array partitioning algorithm for jobs of known length, with important features though. FRUGALLYSCHEDULE returns a schedule f' which can be trivially interpreted also as a schedule f of the original jobs of unknown length. We mention that line 5 of Algorithm 2 is optional and does not improve the bounds of the framework stated in Section 4. Still, we keep it for illustration purposes because without it the algorithm would radically contract machine assignments and use much less machine than m in our examples.

Algorithm 2 *Subroutine: schedules frugally*

```
Require: $P$: jobs of known length $p'_1, \dots, p'_{n'}$
1: function FRUGALLYSCHEDULE($P$, $T$)
2:     $i \leftarrow 1$
3:     for all job $j$ do
4:         if $C_i > 0$ and $C_i + p'_j > 2T$ then $i \leftarrow i+1$
5:         if $C_i > T$ then $i \leftarrow i + 1$        ▷ optional
6:         if $i > m$ then ABORT! $T$ is too small
7:         Assign job $j$ to machine $i$, i.e. $f'(j) := i$
8:     return the computed schedule $f'$
```

The next section formally states that the framework meets the requirements, while Section 5 strengthens it via an example.

4 Analysis of the Framework

Although the framework and the scheduling subroutine look primitive, together they successfully manage some nontrivial issues. Most important one is the convergence (finite number of iteration), achieved by the framework by keeping decreasing the sizes of the longest chains in the database until every non-singleton chain is under $2T_{\mathrm{LB}}$. To assist in that, the scheduling subroutine should not map more than one synthetic job of estimated size ("half chain") to a machine. If each estimation is specified with exact measurements by the end of the iteration, long chains will keep getting smaller and shorter. Even an optimal chained partitioning may map more than one estimated chain to a machine as well as other approximations such as the one in [1], so the existence of FRUGALLYSCHEDULE (Algorithm 2) is justified.

Thus the framework and the subroutine has to interact to succeed. We begin the discussion with the properties of FRUGALLYSCHEDULE.

Proposition 1. *Given the number of machines m and n' jobs of length $p'_1, \dots, p'_{n'}$ by the problem P, let $p'_{\max} := \max_{1 \le j \le n'}\{p'_j\}$. For the schedule f' returned by* FRUGALLYSCHEDULE(P, T) *in Algorithm 2, we have the following:*

(i) if $T \ge T_{LB}$, then the algorithm does not abort and f is a valid chain partitioning,

(ii) if the algorithm does not abort, then $C_{\max}(f') \le \max\{2T, p'_{\max}\}$,

(iii) if the algorithm does not abort, no machine gets two jobs longer than T, even if $C_{\max}(f') > 2T$.

According to part (ii), FRUGALLYSCHEDULE is not an ordinary 2-approximation. It is of importance, because while $p'_{\max}$ is a lower bound on the optimal makespan of the synthetic scheduling problem P, it is usually not a lower bound for the original scheduling problem, as the longest synthetic job may not correspond to a singleton chain. The algorithm is frugal because it does not allow the makespan to reach $2p'_{\max}$, as an ordinary approximation would.

Proposition 1 enables us to finish the analysis. Let $x(q)$ be the maximum size of the chains in the database of length more than $2T_{\mathrm{LB}}$ at the end of the q^{th} iteration, and 0, if every length is under $2T_{\mathrm{LB}}$.

Proposition 2.

(i) In any schedule executed by the framework, the assignment of a machine can contain at most one half chain of estimated length, so the length of the complement of the estimated chain is known.

(ii) $x(q+1) \le \lceil x(q)/2 \rceil$ for all q.

(iii) The framework stops when $x(q)$ is 1 or 0. If $x(q) = 1$, then $C_{\max}(f_q) = s_{\max}$, if $x(q) = 0$, then $C_{\max}(f_q) \le 2T_{LB}$.

Finally, we summarize the results in

Theorem 1. *The framework presented in Algorithm 1 yields a makespan at most $2C^*_{\max}$ using a database of size $O(m \log n)$, after at most $\lceil \log n \rceil$ iterations, starting from any initial schedule f_0.*

5 Demonstration of the Framework

In this section we present a graph generation PSA (Example 2) to test the framework with.

Example 2 (Graph generation PSA [16])**.** *Shown on Fig. 1 (top), $n = 49566$, and let $m = 12$.*

The whole process is illustrated on Fig. 1. The first diagram (on top) visualizes the processing times. The second diagram of Fig. 1 shows the initial schedule f_0 and its completion times $C(f_0)$. The initial schedule assigns the uniform-sized chains to the 12 machines. The assignment intervals are separated by vertical lines with machine indices displayed. There are 12 rectangles on the second diagram, and the width of the i^{th} rectangle corresponds to the *size* (number of jobs) of the assignment of machine i, while its height corresponds to the *length* of this assignment. Obviously the makespan of f_0 is the height of the highest rectangle: $C_{\max}(f_0) = 760177$.

The average completion time is $T_{\text{LB}} = 194851$, so the first iteration splits the first and the second chains, estimate their sizes (not shown on the figure) and computes the next schedule f_1, executes it and measures its completion times. f_1 and $C(f_1)$ are shown on the third diagram. In this iteration, FRUGALLYSCHEDULE did not assign any jobs to machines 10,11 and 12 so we have 9 rectangles. The makespan is still $C_{\max} = 433898$.

The second iteration splits only the third chain of f_1, computes and executes the schedule f_2, updates the database with $C(f_2)$, shown on the fourth, last diagram of Fig. 1. The only difference from the previous iteration is that the one chain split in half is dispatched to two distinct machines. Finally, the makespan is $C_{\max} = 368580$, so the framework stops with the satisfying schedule f_2.

Acknowledgements. The work reported in the paper has been developed in the framework of the project “Talent care and cultivation in the scientific workshops of BME” project. This project is supported by the grant TÁMOP - 4.2.2.B-10/1–2010-0009.

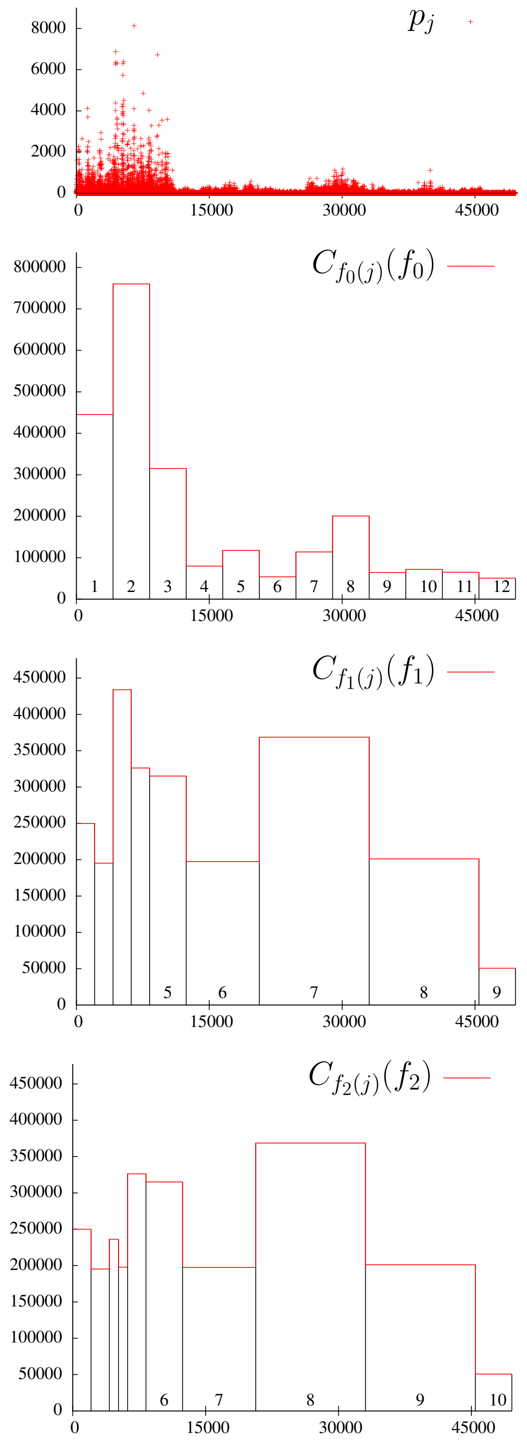

Figure 1: Graph generation PSA processing times p_j and machine completion times $C(f_q)$ of iterations $q = 0, 1, 2$. All horizontal axes correspond to the job number j. Vertical axes denote time.

References:

[1] A. Archer and E. Tardos. Truthful mechanisms for one-parameter agents. In *IEEE Symposium on Foundations of Computer Science*, pages 482–491, 2001.

[2] H. Aytug, M. A. Lawley, K. McKay, S. Mohan, and R. Uzsoy. Executing production schedules in the face of uncertainties: A review and some future directions. *European J. Operational Research*, 161(1):86–110, 2005.

[3] F. Berman, R. Wolski, H. Casanova, W. Cirne, H. Dail, M. Faerman, S. Figueira, J. Hayes, G. Obertelli, J. Schopf, G. Shao, S. Smallen, N. Spring, A. Su, and D. Zagorodnov. Adaptive computing on the grid using AppLeS. *IEEE Transactions on Parallel and Distributed Systems*, 14(4):369–382, 2003.

[4] S. Bokhari. Partitioning problems in parallel, pipeline, and distributed computing. *IEEE Transactions on Computers*, 37(1):48–57, 1988.

[5] M. Chtepen, B. Dhoedt, F. De Turck, P. Demeester, F. Claeys, and P. Vanrolleghem. Adaptive checkpointing in dynamic grids for uncertain job durations. In *Int. Conference on Information Technology Interfaces*, pages 585–590, 2009.

[6] H. Dail, F. Berman, and H. Casanova. A decoupled scheduling approach for grid application development environments. *J. Parallel and Distributed Computing*, 63(5):505–524, 2003.

[7] R. L. Daniels and J. E. Carrillo. β-robust scheduling for single-machine systems with uncertain processing times. *IIE Transactions*, 29:977–985, 1997.

[8] A. B. Downey and D. G. Feitelson. The elusive goal of workload characterization. *SIGMETRICS Performance Evaluation Review*, 26(4):14–29, 1999.

[9] P. Dóbé, R. Kápolnai, A. Sipos, and I. Szeberényi. Applying the improved Saleve framework for modeling abrasion of pebbles. In *Int. Conference on Large-Scale Scientific Computing*, volume 5910 of *LNCS*, pages 467–474. Springer, 2010.

[10] J. Gehring and A. Reinefeld. MARS—A framework for minimizing the job execution time in a metacomputing environment. *Future Generation Computer Systems*, 12(1):87–99, 1996.

[11] R. L. Graham. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal*, 45(9):1563–1581, 1966.

[12] C. Gregg, M. Boyer, K. Hazelwood, and K. Skadron. Dynamic heterogeneous scheduling decisions using historical runtime data. In *Workshop on Applications for Multi- and Many-Core Processors*, San Jose, CA, 2011.

[13] W. Herroelen and R. Leus. Project scheduling under uncertainty: Survey and research potentials. *European J. Operational Research*, 165(2):289–306, 2005.

[14] E. Heymann, M. Senar, E. Luque, and M. Livny. Adaptive scheduling for master-worker applications on the computational grid. In *IEEE/ACM Int. Workshop on Grid Computing*, volume 1971 of *LNCS*, pages 214–227. Springer, 2000.

[15] S. F. Hummel, J. Schmidt, R. N. Uma, and J. Wein. Load-sharing in heterogeneous systems via weighted factoring. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 318–328, New York, NY, USA, 1996. ACM.

[16] R. Kápolnai, G. Domokos, and T. Szabó. Generating spherical multiquadrangulations by restricted vertex splittings and the reducibility of equilibrium classes. Manuscript.

[17] D. A. Lifka. The ANL/IBM SP scheduling system. In *IPPS '95 Workshop on Job Scheduling Strategies for Parallel Processing*, volume 949 of *LNCS*, pages 295–303. Springer, 1995.

[18] C.-K. Luk, S. Hong, and H. Kim. Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *IEEE/ACM Int. Symposium on Microarchitecture*, pages 45–55. ACM, 2009.

[19] T. N. Minh and L. Wolters. Using historical data to predict application runtimes on backfilling parallel systems. In *Euromicro Conference on Parallel, Distributed and Network-based Processing*, pages 246–252. IEEE, 2010.

[20] C. Papadimitriou. Algorithms, games, and the internet. In *ACM Symposium on Theory of Computing*, pages 749–753. ACM, 2001.

[21] A. Pınar and C. Aykanat. Fast optimal load balancing algorithms for 1D partitioning. *J. Parallel and Distributed Computing*, 64(8):974–996, 2004.

[22] D. Tsafrir, Y. Etsion, and D. Feitelson. Backfilling using system-generated predictions rather than user runtime estimates. *IEEE Transactions on Parallel and Distributed Systems*, 18(6):789–803, 2007.

[23] Z. Wang and M. F. O'Boyle. Mapping parallelism to multi-cores: a machine learning based approach. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 75–84. ACM, 2009.

[24] G. Wrzesinska, J. Maassen, and H. E. Bal. Self-adaptive applications on the grid. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 121–129. ACM, 2007.