Channel Based Scheduling of Parallelizable Tasks

Jason Glasgow
CenterLine Software
Cambridge, MA, USA
glasgow@centerline.com*

Hadas Shachnai
Deprtment of Computer Science
The Technion, Haifa, Israel
hadas@cs.technion.ac.il

Abstract

In this paper we consider the problem of scheduling a set of tasks on a parallel machine of identical processors. The tasks are parallelizable and can be run simultaneously on several processors, in which case the runtime is decreased. Our goal is to minimize the finish time (or Makespan) of the entire schedule. This problem is known to be NP-Hard.

We propose a new approach to scheduling, based on partitioning the available processors into a fixed number of computation channels. *We assign tasks to channels based on their execution times, and their speed-up function, assuming that these parameters are available prior to the execution of the task sequence.*

The channel approach is shown to be advantageous whenever the overall work needed to execute tasks does not decrease as a function of the number of processors assigned to it, i.e. in most common scenarios. For cases in which this function is a constant *(and, therefore, the overall runtime per task decreases linearly with the number of processors executing it), we present a new scheduling heuristic called the* Partition and Assignment (PA) *algorithm. PA is shown to achieve a worst case bound of 2 to the optimal schedule. It runs in linear time, $O(n+m)$, where m is the number of processors, and n is the number of tasks. For the case of non-linear speedup, we introduce a generalized version of PA (GPA), which achieves a bound of 2 to the optimum, and runs in time $O(m \log a + n)$, where $a = \min(n, m)$.*

1. Introduction

1.1. Model Formulation

The problem of scheduling tasks on machines is a classic resource allocation problem. Scheduling algorithms take as an input an arbitrary number of tasks, each with an execution time, and produce as output a mapping from tasks to processors and start times so as to minimize either the average waiting time, or the finish time of the entire schedule.

In a well studied version of this problem, each task runs on a single processor, until it completes execution. In this work we consider a recent trend in multiprocessors scheduling, where tasks may be scheduled to run on more than one processor, thus decreasing their runtime.
We distinguish between two cases:

- Linear speedup – In some applications the runtime of a task decreases linearly with the number of processors assigned to it.
- In other scenarios, when the overhead inherent in splitting the task among several processors becomes a significant factor, the decrease in runtime is only sub-linear.

Each of these cases corresponds to a class of problems solved on parallel computers. Linear speedup models the many applications that can be trivially parallelized, such as database searches, where the interprocess communication involved is negligible. Circuit simulation using waveform relaxation is another example which obtains linear speedup as long as the circuit can be partitioned into as many pieces as there are processors. Thus we expect a speedup function that decreases linearly until a certain point, and then levels off.

In analyzing the non-linear case, we refer to the notion of *work*, defined as the sum of the running times of the processors allocated to a specific task. The *work function* describes the dependence of the work needed to execute a task upon the number of processors allocated to it. This dependence upon the number of processors stems from the communication overhead of the processors engaged in the execution of a split task. In most common scenarios this communication increases with the number of involved processors. We, therefore, focus on work functions that are non-decreasing in the number of processors. (e.g., see [2]).

*Part of this research was done while the author was at the Department of Electrical Engineering, The Technion, Haifa, Israel.

Our performance measure is the makespan of the schedule, i.e., we seek a schedule that minimizes the overall completion time of the tasks. Indeed, minimizing the makespan corresponds to maximizing the throughput of the system. In [4] it is shown that the problem of finding an optimal schedule with respect to the makespan, for parallel task systems, is strongly NP-hard already for five processors. This motivates our search for an efficient algorithm that will guarantee a reasonable approximation to the optimum.

Assuming a set of identical processors, on which the tasks are to be scheduled, we develop algorithms which are non-preemptive, meaning that once the execution of a task begins, the processor allocation to this task cannot be changed, and the task cannot be interrupted and resumed at a later time. Just the same, the performance guaranteed is within a factor of 2 from the optimum, within the set of *all* possible schedules (including preemptive schedules, and schedules that allow reconfigurations).

1.2 The Channel approach

A typical question in the design of the operating system for a multiprocessor is whether to maintain a central queue for all the processors, or a local queue for each individual processor. We propose an intermediate solution, that is efficient in maintaining a *small* number of queues. Each of the queues is served by a subset of the processors. By maintaining these local queues (in shared memory multiprocessors and similar architectures), we reduce the contention for global locks, which in turn decreases the overhead of scheduling.

We propose a new approach to scheduling parallelizable tasks, which we call the *channel approach*. The channel approach is based on the idea of partitioning the available processors into computation channels. Each channel is composed of a subset of processors, and tasks are assigned to channels based on their *degree of parallelism*, which is determined upon the arrival of the tasks to the system. Thus, a main advantage of the channel approach is in reducing the algorithmic complexity of the scheduler. It also enables a fixed allocation of resources to subsets of the tasks.

By using computation channels, we reduce the problem of scheduling parallelizable tasks on a set of identical processors, to the problem of finding an optimal schedule for non-parallelizable tasks running on a system of *unrelated machines* [6]. Channel based algorithms may be viewed as involving two stages: First determining a partition of the processors to channels (that is, the set of 'unrelated machines'), and then assigning tasks to the channels. For the second stage, we present assignment algorithms, which run in time linear in the number of tasks, and thus suggest simpler scheduling heuristics for the unrelated machines problem.

1.3 Related Work and Outline of Results

Scheduling to minimize the makespan for *non-parallelizable* tasks (i.e. each task running on a single processor) has been extensively studied. Recent works consider the problem of minimizing the makespan for *parallelizable* tasks. Wang and Cheng present in [9] the *Earliest Completion Time (ECT)* algorithm, for which they prove a worst case bound of $3 - 2/m$ on the makespan generated relative to the optimum, with m the number of processors. [3] describes two heuristic algorithms for scheduling tasks assuming non-decreasing work functions. Their worst case bound with respect to the optimum is $2/(1 + 1/m)$ with complexity of $O((n + m) \log m)$. In the papers [7, 8] heuristic algorithms are presented, that solve the problem of minimizing the makespan given a set of independent tasks, and any speedup function. The algorithm of [7] works by applying a non-malleable scheduling (NMS) algorithm to a family of processor allocations, and obtains a bound on the worst case behavior of 2.5 if processor addresses need to be contiguous (non-fragmentable), and a bound of 2 for fragmentable systems. The complexity of these algorithms is $O(mn)$ times the complexity of a NMS scheduling algorithm. Typically, this results in complexity $O(mn^2 \log n)$.

The algorithms presented in this paper improve the worst case bound achieved for the ECT, and for linear speedup tasks yield the same bound as in [3] with *lower* computational complexity. Our results are valid for both fragmentable and nonfragmentable systems.

The rest of the paper is organized as follows. In Section 2 we present an $O(m + n)$ algorithm, which we call the *Partition and Assignment (PA)* algorithm, for scheduling a set of n tasks on m processors, with linear speedup. We prove a bound of 2 on its worst case makespan relative to the optimal schedule. In Section 3 we modify the algorithm and proofs in order to show similar results for tasks with non-decreasing work loads. The *Generalized Partition and Assignment* (GPA) algorithm is shown to achieve a worst case bound of 2 to the optimal makespan, in complexity $O(m \log a + n)$, where $a = \min(n, m)$. We conclude in Section 4 with experimental results that characterize the behavior of the channel algorithms for various distributions on task sizes and levels of parallelism, subject to nondecreasing work functions.

2. Tasks with Linear Speedup

Under the constraint that all tasks have a linear speedup function, i.e. the runtime for task T_i when schedule on p processors is t_i/p, the problem of finding the optimal makespan has a trivial solution. Assigning m processors to each task, and arbitrarily ordering the tasks, will yield an optimal schedule. However a more realistic model is

obtained if we assume that each task has a maximum level of parallelism defined by δ_i, $1 \leq \delta_i \leq m$. Scheduling T_i on more than δ_i processors will not result in runtime less than t_i/δ_i.

First let us define a lower bound on the optimal schedule. The finish time, w^*, of the the optimal schedule is greater than both the total amount of work divided by the number of available processors and the maximum completion time for any task when the task is allocated as many processors as it can use.

$$w^* \geq \max\left(\frac{\sum_i t_i}{m}, \max_i \frac{t_i}{\delta_i}\right) \tag{1}$$

We denote the average finish time for the processors by

$$\bar{w} = \frac{\sum_i t_i}{m}. \tag{2}$$

In most cases our bounds will relate to $\bar{w}$ and not to w^* (the actual optimum) since evaluating w^* from the input set is not possible in polynomial time.

We describe a partition of the m processors into computation channels by C, the number of computation channels, and the sequence S_k, $1 \leq k \leq C$, the number of processors in the k-th channel. In order to assure that all processors belong to a channel we require that $\sum_{i=1}^{C} S_k = m$.

Given the set of tasks $\{T_1, T_2, \cdots, T_n\}$, let $U_{S_k} = \{i \mid \delta_i \geq S_k\}$ be the set of indexes of tasks that can exploit a channel of width S_k without idling any of the processors. Denote by

$$S'_k = \sum_{S_j \geq S_k} S_j$$

the total channel capacity of channels of size S_k or larger, and let

$$W_k = \sum_{i \in U_{S_k}} t_i.$$

W_k is the sum of work available for a channel of size S_k, such that the channel will be used at full efficiency (no idle processors). It should be noted that $T_i \in U_{S_k}$ does not imply that T_i is scheduled on channel k. It could possibly be scheduled on a channel k', $S_{k'} \geq S_k$.

We present below the *Partition and Assignment* (PA) Algorithm which creates a partition of the processors into computation channels and assigns the tasks to channels. Our first definition describes constraints on the partition that will allow the scheduling algorithm to guarantee a bound of 2 on the makespan.

Definition 1 (Proper Channel Capacity) *We have proper channel capacity, if*

$$\frac{\sum_{i \in U_{S_j}} t_i}{S'_j + S_j} \leq \bar{w} \leq \frac{\sum_{i \in U_{S_j}} t_i}{S'_j} \tag{3}$$

for all channels j, $1 \leq j \leq C$ where $\bar{w}$ is given in equation (2).

The constraint in (3) ensures that there is at least enough work on the average to keep each channel busy until after $\bar{w}$, and that if for any j we added another channel of size S_j we would not have enough work.

Algorithm 1 (Partition) *Given a set of tasks, T_i, the following algorithm will create a partition of the processors into computation channels.*

```
j ← 1
S ← 0
C ← 1
for(k = m downto 1)
    while( W_k/(S+k) ≥ w̄ ) do
      begin
        S_j = k
        S ← S + k
        C ← C + 1
        j ← j + 1
      end
endfor
```

Claim 1 *The Partition algorithm creates a partition. That is $S = \sum S_j = m$.*

The inequality, $S = \sum S_j \leq m$ follows directly from the fact that for the last channel created by the algorithm (of width l, $1 \leq l \leq m$)

$$\frac{W_l}{S} \geq \bar{w} \;\Rightarrow\; \frac{W_l}{S} \geq \frac{\sum t_i}{m} \;\Rightarrow\; \frac{W_l}{\sum t_i} m \geq S$$

and $W_l \leq \sum t_i$.

In order to show equality, we look at the last step of the algorithm, $k = 1$. As long as $W_k/(S+k) \geq \bar{w}$ we will create another channel of size 1, and thus increment S. This continues until $W_k/(S+k) < \bar{w}$. (The algorithm may stop earlier.) Thus S and m being integers, S must equal m after the last step. ■

Claim 2 *The partition created by algorithm 1 satisfies the proper channel capacity constraint.*

The proof is immediate from the algorithm, and is thus omitted.

Algorithm 2 (Task Assignment Algorithm) *Place the task T_i on the largest channel S_k such that $S_k \leq \delta_i$. If all the channel capacities are unique integers, then the mapping from tasks to channels is well defined. If there is more than one channel of a given size then assign tasks to each channel (among the set of identical width channels) until the channel completion time exceeds $\bar{w}$.*

Theorem 1 *Given a set of tasks, and a partition that satisfies the proper channel capacity constraint, Algorithm 2 achieves a makespan w satisfying $w \leq 2w^*$.*

The proof uses backward induction. We give it in the full version of the paper.

Recall that the *Partition and Assignment* (PA) algorithm is the combination of algorithm 1 and 2. It is easy to verify that the complexity of the Algorithm PA is O($n + m$).

Corollary 1 *For any set of tasks, $T_1, \ldots, T_n$, the Algorithm PA achieves a makespan $w \leq 2w^*$. Moreover, if there is at least one task, T_i, such that $t_i/\delta_i \geq 2\bar{w}$ then PA achieves the optimal makespan, i.e. $w = w^*$.*

Corollary 2 *The bounds presented in corollary 1 apply also to the preemtible and reconfigurable versions of the makespan problem.*

The generalization in corollary 2 follows immediately from the fact that the bounds are based on comparisons with the minimal amount of work done under an optimal schedule, which would not decrease if preemptions and reconfigurations were allowed.

3. Minimizing Makespan for Tasks with Non-Linear Speedup

When we generalize the problem of minimizing the makespan to include tasks that have non-linear speedup functions, we have to modify our scheduling algorithm. The required changes increase the computational complexity of the algorithms, but still maintain a worst case bound on the makespan of less than two times the optimal. We now restate the problem:

Given a set of n tasks and n functions $f_1(p), \ldots, f_n(p)$, $1 \leq p \leq n$ which specify the runtime of the i-th task when scheduled on p processors, find a mapping of tasks to processors that will minimize the finish time of the schedule. A reasonable assumption is that $t_i(\delta) \geq t_i(\delta + 1)$, i.e. the runtimes are non-increasing. The results below apply to any model where the amount of work done, $t_i(\delta) \cdot \delta$ is non-decreasing, that is, $t_i(\delta) \cdot \delta \leq t_i(\delta + 1) \cdot (\delta + 1)$, for $1 \leq \delta \leq m$. We assume that each task can be allocated any number of processors in the range [1,m].

First we design an algorithm that will allocate a set number of processors to each task. Denote by δ_i the number of processors allocated to task T_i, and let $\vec{\delta} = (\delta_1, \ldots, \delta_n)$ be the current allocation. Then the makespan of the optimal schedule for this fixed allocation denoted by $w^*(\vec{\delta})$ satisfies

$$w^*(\vec{\delta}) \geq \max\left(\frac{\sum_i t_i(\delta_i)\delta_i}{m}, \max_i t_i(\delta_i)\right) \qquad (4)$$

$w^*(\vec{\delta})$ is the lower bound for the non-malleable multiprocessor scheduling problem (NMS). Non-malleable scheduling is a term coined by [8] to describe the problem of scheduling n tasks on m processors when each task must be assigned to a predetermined number of processors. In contrast, malleable scheduling refers to the problem we discuss, in which tasks can be assigned to a variable number of processors. From the above equation we can form a lower bound on the malleable scheduling problem by taking the minimum over all allocations $\vec{\delta}$.

$$w^* \geq \min_{\vec{\delta}} w^*(\vec{\delta})$$

We base our tentative allocation of processors to tasks on equation (4). We initially allocate $\delta_i = 1$ processors to each task. We compute the average processor completion time,

$$\bar{w}(\vec{\delta}) = \frac{\sum t_i(\delta_i)\delta_i}{m}. \qquad (5)$$

Then we compare $\bar{w}(\vec{\delta})$ with the execution time of the longest task. As long as $\bar{w}(\vec{\delta})$ is less than the execution time of the longest task, we increase δ_i for that task. Clearly that task previously defined the lower bound on the makespan. We update $\bar{w}(\vec{\delta})$ and repeat the previous step until we can no longer decrease the lower bound on the makespan. After we have determined a tentative allocation of processors to tasks, $\vec{\delta}$, we can compute $W_k = \sum_{T_i:\delta_i \geq k} t_i$ and $\bar{w} = \bar{w}(\vec{\delta})$. Using these values we apply the algorithm PA to partition the processors into channels and to assign the tasks to channels. We call the combination of algorithm 3 with the PA algorithm the *Generalized Partition and Assignment* (GPA) algorithm.

Algorithm 3 (Generate δ_i) *Given a set of tasks, $T_1, \ldots, T_n$, the following algorithm will compute a tentative maximum allocation of processors, δ_i, for each task T_i, $1 \leq i \leq n$.*

for i :=1 to n do $\delta_i \leftarrow 1$
$\bar{w} \leftarrow \sum_{i=1}^{n} t_i(\delta_i)\delta_i/m$
$A \leftarrow \{T_i | t_i > \bar{w}\}$
$B \leftarrow \{T_i | t_i \leq \bar{w}\}$
`while`($A \neq \emptyset$)
  $i = \arg\max_{i:T_i \in A} t_i(\delta_i)$
  if ($t_i(\delta_i) > \bar{w}$) then
    `begin`
      $\bar{w} \leftarrow \bar{w} - t_i(\delta_i)\delta_i/m + t_i(\delta_i + 1)(\delta_i + 1)$/m
      $\delta_i \leftarrow \delta_i + 1$
      $j \leftarrow i$
      if ($t_i(\delta_i) \leq \bar{w}$) then $B \leftarrow B \cup T_i$; $A \leftarrow A - T_i$
    `end`
  `else`

```
    begin
(*)     if (w̄ > t_j(δ_j − 1)) then δ_j ← δ_j − 1
        STOP
    end
end
```

Theorem 2 *For the vector, $\vec{\delta} = (\delta_1, \ldots, \delta_n)$ defined by algorithm 3, $\bar{w}(\vec{\delta}) \leq w^*$*

Proof: The lower bound on the makespan is defined as either the average processor completion time caused by the workload, or the longest task. Since we increase the workload only when it will decrease the lower bound, it should be clear that the average processor load, $\bar{w}$ will be less than, or equal to, the makespan generated by the optimal algorithm. Step (*) of the algorithm checks to makes sure that the last task to receive an increased allocation did not cause the lower bound to increase. If it did cause the lower bound to increase, then this last allocation is reversed. ■

Lemma 1 *Let $|A|$ denote the initial number of tasks whose run time is greater than $\bar{w} = \bar{w}(\vec{\delta})$ where $\vec{\delta} = (1, \ldots, 1)$, then $|A| \leq \min(n, m)$.*

Proof: Clearly there cannot be more tasks with runtimes greater than $\bar{w}$ than there are tasks. Thus $|A| \leq n$. If $n > m$, let T_m be the m-th largest task. Looking at $T_1, \ldots, T_m$ it is clear that $\bar{w} \geq t_m$, and thus there are at most $m-1$ tasks that have running times greater than $\bar{w}$.■

Theorem 3 *The complexity of algorithm 3 is $O(m \log a + n)$, where $a = \min(n, m)$.*

In order to prove a worst case bound on the makespan produced by the algorithm GPA, we must reformulate the definition of the proper channel capacity.

Definition 2 (Proper Channel Capacity II) *Let C be the number of unique sized channels, S_k be the k-th channel width, and N_k the number of channels of size S_k. Let $S'_k = \sum_{j \geq k} N_j \cdot S_j$ be the capacity of all channels larger or equal to S_k. Given a set of tasks with non-linear speedup functions, and processor allocations $\delta_1, \ldots, \delta_n$, we have proper channel capacity if $\forall k, 1 \leq k \leq C$:*

$$\frac{\sum_{\delta_i \geq S_k} t_i(\delta_i) \cdot \delta_i}{S'_k + S_k} \leq \bar{w} \leq \frac{\sum_{\delta_i \geq S_k} t_i(\delta_i) \cdot \delta_i}{S'_k}$$

Generalized Partition and Assignment Algorithm (GPA)

Input: a set of n tasks, $T_1, \ldots, T_n$, n speedup functions, $t_1(\cdot), \ldots, t_n(\cdot)$, and m processors.

Output: a partition of the processors into C channels of size $S_1, \ldots, S_C$, and a mapping of the tasks to channels.

1. Generate a tentative allocation, $\vec{\delta}$, of processors to tasks using Algorithm 3.
2. Compute $\bar{w} = \bar{w}(\vec{\delta})$ and W_k, $1 \leq k \leq m$.
3. Apply algorithm 1 in order to partition the processors into channels using $\bar{w}$ and W_k, $1 \leq k \leq m$.
4. Assign the tasks to channels with algorithm 2.

Theorem 4 *Given a set of tasks $T_1, \ldots, T_n$, the algorithm GPA achieves a makespan which is less than 2 times the optimal.*

We give the proof in the full version of the paper.

4. Experimental Results

In the previous sections we focused on analyzing the worst case performance of channel based scheduling algorithms. In this section we present the average case behavior of these algorithms obtained by simulations.

We simulated tasks with two different types of speedup functions and where $m = 32$. We compiled results from runs with various numbers of tasks n between 10 and 120. In all cases the execution time of the tasks was chosen to be an exponentially distributed random variable with mean 1.

For linear speedup tasks, we chose the maximum level of parallelism to be a uniformly distributed random variable between 1 and 32 (the number of processors). For each given number of tasks, we ran the PA algorithm on 200 different input sets. For each input set (trial) we computed the ratio of the resultant makespan to the lower bound of the optimal. The average ratio over the 200 trials is plotted in figure 1 versus the number of tasks.

We also ran simulations for a non-linear speedup function based on Amdahl's law[1] which is a typical instance of a non-decreasing work function. Let $C_i = C'_i/t_i$ be the ratio of sequential to parallel code for task T_i. Amdahl's law asserts that the execution time on of the task on p processors will be $t_i(p) = t_i \frac{1+pC_i}{p}$. In the simulations, a value of C was chosen for each task distributed uniformly on the interval $[0.05, 0.1]$. This allows each task to have slightly different behavior when parallelized. The choice of C_i represents tasks that contain less than 10% non-parallelizable code.

Figure 2 shows the results of two versions of GPA for the tasks with Amdahl speedup functions. *List Average* is the average ratio of the GPA makespan to the optimum when tasks are scheduled on identically sized channels using the PA Algorithm. If though, the tasks are sorted and scheduled on identical channels based on *Longest Processing Time* (LPT) [5], we see a significant improvement in the average case behavior. These results

are labeled *LPT Average*. Both versions of the algorithm maintain the same worst case bound.

Similar results were obtained when the GPA Algorithm was applied tasks with the non-linear speedup function $t_i(p) = \frac{t_i}{p^{\alpha_i}}$ where α_i is a uniformly distributed random variable in the interval $[0.8, 0.9]$. This is another typical non-linear speedup function, but in contrast to Amdahl's law the runtime of a task decreases without approaching an asymptotic minimum. Figure 3 shows the ratio of the average makespan to the lower bound of the optimal for tasks that have a speedup function using both the standard list scheduling version of GPA and the improved LPT based version.

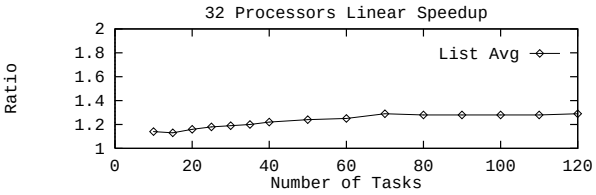

Figure 1. Average ratio of PA makespan to optimum for linear speedup tasks

Acknowledgment

We would like to thank Shai Ben-David for his helpful comments and suggestions regarding the presentation of the paper.

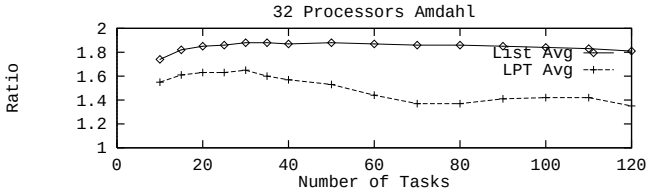

Figure 2. Average ratio of GPA makespan to optimum for Amdahl speedup tasks

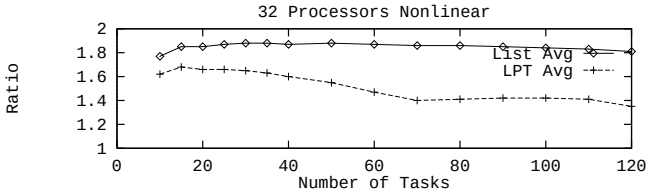

Figure 3. Average ratio of GPA makespan to optimum for Non-linear speedup tasks

References

[1] G. M. Amdahl. Validity of single processor approach to achieving large-scale computing capabilities. In *AFIPS Conference Proceedings 30*, pages 483–485. AFIPS Press, 1967.

[2] Amir Averbuch, Eran Gabber, Boaz Gordissky, and Yoav Medan. A parallel FFT on a MIMD machine. *Parallel Computing*, 15:61–74, 1990.

[3] K. P. Belkhale and P. Banerjee. Approximate algorithms for the partitionable independent task scheduling problem. In *1990 International Conference on Parallel Processing*, pages 72–75, 1990.

[4] J. Du and J. Y-T. Leung. Complexity of scheduling parallel task systems. *SIAM Journal on Discrete Mathematics*, pages 473–487, November 1989.

[5] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, March 1969.

[6] A. M. A. Hariri and C. N. Potts. Heuristics for scheduling unrelated parallel machines. *Computers in Operations Research*, 18(3):323–331, 1991.

[7] J. Turek, J. L. Wolf, K. R. Pattipati, and P. S. Yu. Scheduling parallelizable tasks: Putting it all on the shelf. *Performance Evaluation Review*, 20(1):225–236, June 1992.

[8] J. Turek, J. L. Wolf, and P. S. Yu. Approximate algorithms for scheduling parallelizable tasks. In *Proceedings of the Fourth ACM Symposium on Parallel Algorithmns and Architectures*, pages 323–332, 1992.

[9] Q. Wang and K. H. Cheng. A heuristic of scheduling parallel tasks and its analysis. *SIAM Journal on Computing*, 21(2):281–294, April 1992.