Considerations on Distributed Load Balancing for Fully Heterogeneous Machines: Two Particular Cases

Nathanaël Cheriere
Department of Computer Science
ENS Rennes
Rennes, France
nathanael.cheriere@ens-rennes.fr

Erik Saule
Department of Computer Science
University of North Carolina at Charlotte
Charlotte, USA
esaule@uncc.edu

***Abstract*—When the size of parallel systems increases, centralized algorithms to schedule tasks on the system can induce a significant overhead. This is why decentralized scheduling algorithms have been developed. The most popular one certainly is work-stealing because of its interesting theoretical guarantees. Parallel systems have evolved from homogeneous clusters to fully heterogeneous ones such as GPU-accelerated clusters. We investigate in this paper decentralized scheduling algorithms for heterogeneous systems. The guarantees of work-stealing algorithms no longer hold on such systems because it is an a posteriori algorithm which highly depends on the initial distribution of work.**

We focus on a priori decentralized scheduling algorithms for heterogeneous systems and we propose two distributed algorithms to balance the load on unrelated machines for two particular cases. The first one exploits a low heterogeneity in the task set and reaches an approximation ratio linear in the number of types of tasks. The second one focuses on the case where the system only uses two different types of machines and we show it is a 2-approximation if the system converges. In the case it does not converge, we study the dynamic equilibrium of the system. In the homogeneous case, we numerically compute the probability density function of the load imbalance and show that the imbalance is low on average. And we show using simulation that the heterogeneous case is similar to the homogeneous case and that the imbalance is low in both cases.

***Keywords*-Unrelated Machine Scheduling; Load-Balancing; Decentralized Algorithms; Approximation Algorithms**

I. Introduction

Scheduling the execution of a parallel program on multiple machines is one of the basic problems in parallel computing. Its solution is far from obvious as many hypotheses can be done to specify the conditions, especially on the heterogeneity of the machines used to execute the program. These problems have been studied since the '60s, but often from a centralized point of view. However, with the current size of parallel systems, the cost of the resolution of the problem on one machine cannot be overlooked anymore [16]. Centralized systems have a high overhead when the number of managed processors increases. Even a simple "least loaded machine first" scheduling policy takes a linear time in the number of tasks and logarithmic in the number of machines [12]. This obviously does not scale well when the size of the system increases and does not handle at all the inherent imprecision of all scheduling systems (runtimes are typically difficult to predict).

Decentralized load balancing algorithms have been designed to alleviate the bottleneck of having a single processor doing all the scheduling. The work stealing strategy, introduced in [7] and modified for Cilk [6], is certainly the most popular decentralized scheduler: every machine is responsible for its own charge, and when a machine no longer has jobs to execute, it tries to steal some from another machine. This strategy has guaranteed performance[1] when the machines are identical [6] and even when the machines process the tasks at different speeds [1].

However modern machines are composed of fully heterogeneous processors. Two processors, such as a CPU and a GPU, are not just different in term of their speed, they also differ in capabilities: for some tasks a CPU might be fast and a GPU slow while it could be the reverse for other tasks. Moreover, in the case of two CPU with different architecture, a task might be optimized for a specific architecture and perform worse on the other architecture. Common decentralized load balancing strategies could be applied to these systems. But they would no longer guarantee anything on the quality of the derived solution.

In this work, we investigate the problem of decentralized load-balancing for heterogeneous machines under the perspective of provable performance. The contribution of this work are as follow:

- We show in Theorem 1 that work stealing policy can lead to arbitrarily bad load balance in the fully heterogeneous case.
- We provide algorithm *MJTB* which is decentralized and converges to a solution no worse than k the optimal solution assuming there are only k groups of tasks (Theorem 5).

[1]Guaranteed refers to an approximation ratio; that is to say theoretically proven upper bound of the ratio between what the algorithm obtains and the optimal solution. Though in this case the guarantee is not achieved in the worst case but in average.

- We provide an iterative decentralized algorithm, *DLB2C*, for systems with only two types of processors (such as a CPU-GPU cluster) and show that if the algorithm converges, then the obtained solution is a 2-approximation (Theorem 7)
- In cases where *DLB2C* does not converge to a single solution, we study its behavior by modeling the one cluster case using markov chains and show the imbalance is bounded with high probability (Figure 2). We also show using simulations that the two clusters case behaves like the one cluster case (Figure 4) and that a solution no worse than 1.5 times a guaranteed centralized solution is typically reached within a few iterations (Figure 5).

The document is organized as follows. We formally introduce the problem in Section II. We quickly survey results related to our problem in Section III. Section IV discusses *a posteriori* and *a priori* load balancing algorithms. Then we investigate two cases for *a priori* algorithms. We investigate instances where the number of types of tasks is small in Section V. Section VI presents a decentralized algorithm for the case where there are only two types of processors and we show that if it converges the obtained solution is a 2-approximation. We study the case where it does not converge using markov models and simulation in Section VII. Section VIII concludes this document.

II. Problem Definition

The problem studied in this work is a load balancing problem, the goal is to construct a partition S of a set of sequential independent jobs J onto the machines in the set M in order to minimize an objective function. A job is a process that can only be executed on one machine at a time and it is independent from other jobs. We also assume that it has to be executed without interruptions to be completed. The machines do not share memory. For the purpose of this paper we will use interchangeably the words "task" and "job". Also, we will use interchangeably the words "processor" and "machine".

Many functions can be minimized but we focus on the makespan which is the time at which all the jobs have been executed. We denote by $p_{i,j}$ the time needed for the execution of job j on the machine i. And $S(i)$ is the set of jobs assigned to machine i. We denote the makespan as $C_{max}(S) = \max_{i \in M} C(S,i)$ where $C(S,i) = \sum_{j \in S(i)} p_{i,j}$ is the time when all the jobs of machine i have been completed.

In the following parts, we will denote $C(S,i)$ as $C(i)$ when there is no ambiguity on the solution used to compute the makespan. In this problem, the partition obtained is usually compared to the minimum possible makespan OPT which characterizes the optimal solutions, $\mathrm{OPT} = \min_S C_{max}(S)$.

The machines can be homogeneous, heterogeneous related, or heterogeneous unrelated.

- The machine set can be homogeneous (identical). In this case, the machines are said to be uniform, each job j can be executed on every machine i within the same amount of time, $\forall\ i, i' \in M,\ p_{i,j} = p_{i',j}$.
- In the heterogeneous related case, all the machines are different but only differ by a fixed factor. For all the jobs we have $\forall\ i, i' \in M, \exists\ \alpha, \forall\ j \in J,\ p_{i,j} = \alpha p_{i',j}$.
- The heterogeneous unrelated (fully heterogeneous) case is the most general. There is no assumption on the relation of processing times of the tasks on the processors, $p_{i,j}$ is an arbitrary value (which can be infinite).

The centralized version of the problem we are interested in is known as scheduling independent tasks on unrelated parallel machines to minimize the makespan and is denoted $R||C_{max}$ in the classical 3 field notation of Graham *et al.* [13]. This problem is known to be NP-Complete [10].

Here we are interested in the decentralized version of this problem. Therefore we will assume that the jobs have an arbitrary initial distribution. This could apply in cases where there is a static set of tasks which is pre-distributed. But it could also apply in cases where tasks directly appear on processors. Common reasons for this are if a task can locally spawn more tasks or if tasks are submitted to a particular processor.

III. Related Work

Certainly the most famous algorithm for these kinds of problem is List Scheduling proposed by Graham in 1966 [11] which consists in scheduling tasks greedily on the first machine available. On identical machines, this algorithm is a 2-approximation algorithm: the generated solution S is such that $C_{max}(S) \leq 2 \cdot \mathrm{OPT}$. In the offline case, the tasks are known in advance and one can order the tasks using a Largest Processing Task (LPT) first order, and then List Scheduling becomes a $\frac{4}{3}$-approximation algorithm [12]. Later, dual approximation methods lead the way to Polynomial Time Approximation Schemes (PTAS) for the identical machine case; they have a higher complexity but their approximation ratio can be arbitrarily close to 1 [15].

The problem of scheduling tasks on fully heterogeneous machines is not as well solved. Lawler and Labetoulle showed that the problem with *pre-emption*[2] can be solved in polynomial time using linear programming [18]. The problem without pre-emption can be approximated within a factor 2 also using a linear programming problem but then using intelligent rounding techniques to reconstruct an integer solution [20]. A faster algorithm using unsplittable flow has been designed, but still is a 2-approximation

[2]The possibility to pause a job on a machine and restart later it on another machine

Algorithm 1: *Work Stealing* for a particular machine

```
Data: m machine
Data: S(m) jobs assigned to m
while true do
    if S(m) = ∅ then
        Select randomly a target machine i
        if S(i) ≠ ∅ then
            Steal half of the non executed jobs of i
    else
        Start running a job j of S(m)
        Remove j of S(m)
```

algorithm [9]. It was also shown that the problem cannot be approximated with a better approximation ratio than $\frac{3}{2}$ unless $P = NP$ [20]. However, if the number of processors is fixed then approximation schemes can be designed (for instance if there are only two processors [17]).

All the previously mentioned works propose offline centralized algorithms. A common approach for the identical machine case is to make the system online by scheduling the tasks at the time of their submission. Typically such systems rely on the greedy properties of List Scheduling and maintain the load of each machine in a priority queue to schedule each task on the least loaded processor in $O(\log |M|)$ and guarantees that each intermediate solution is a 2-approximation [11]. This does not scale with the number of processors and is inherently centralized.

Approaches based on the "balls in bins" problem trade accuracy for complexity. Instead of picking the least loaded processor, the algorithm picks the least loaded processor of d randomly (uniformly) chosen processors in $O(d)$ and makes an error of roughly $\ln \ln |J| / \ln d$ [4]. This principle has even been successfully adapted in the heterogeneous related case [2], [3]. These methods are fully decentralized if one assumes that the load of a processor can be probed externally but does not apply to fully heterogeneous systems.

In the context of heterogeneous systems, Chen *et al.* addressed the problem of load balancing at submission time for clusters composed of two sets of identical machines. Machines within the same set are identical but two machines from different sets can be heterogeneous unrelated. They proposed a centralized 4-approximation algorithm [8].

The most common decentralized scheduler that does not distribute the tasks at submission time is the work-stealing strategy (Algorithm 1) introduced by Burton and Sleep [7]. This strategy is essentially a decentralized implementation of List Scheduling [21]. Each machine maintains a list of locally available tasks and when a machine finishes its last local job, it contacts its neighbours and tries to steal some of the non running jobs. This idea was applied to the Cilk middleware and some guarantees on the execution time of a batch of jobs using a work stealing algorithm on identical parallel machines have been derived [5], [6]. In particular, the expected maximum makespan is bounded by the average work per machine plus a big-O of the critical path[3] p^∞ of the problem, $\mathbb{E}(C_{max}(S)) \leq \sum_{j \in J} p_{i,j}/|M| + O(p^\infty)$. This result was later extended to the problem with heterogeneous related machines and the possibility to use pre-emption [1].

IV. A Priori Load Balancing

Load balancing at the task's submission time works in the case of identical machines (or eventually related machines) essentially because List Scheduling is a good greedy strategy for this problem. But there are no known good greedy algorithms for the heterogeneous case; which is why algorithms that work in this case only work under particular hypotheses, such as limiting the number of types of processors to two [8]. The inner difficulty of the problem makes the finding of a generally applicable method unlikely.

On the other hand, applying a work stealing strategy on unrelated machines has a shortcoming. Indeed, this *a posteriori* strategy starts stealing jobs from another machine only when the work previously scheduled on the machine has been executed. But if the initial distribution is poorly done (like in Table I), the first steal can happen long after the optimal makespan.

Theorem 1. *Applying a work stealing strategy on unrelated machines can induce an unbounded makespan.*

Proof: Table I presents the $p_{i,j}$ values of a 5 tasks, 3 machines instance of the problem. If the initial distribution follows the allocation denoted by a circle in the table, the first steal operation only happens after a time n, and the execution can be finished in $n + 1$ units of time. However, with a good schedule the work can be finished in 2 units of time. ■

Cost	Machine A	Machine B	Machine C
Job 1	1	ⓝ	n
Job 2	1	1	ⓝ
Job 3	ⓝ	1	1
Job 4	ⓝ	1	1
Job 5	ⓝ	1	1

Table I
THE CIRCLED DISTRIBUTION IS A BAD INITIAL DISTRIBUTION TO APPLY A *Work Stealing* STRATEGY; THE FIRST STEAL CAN ONLY HAPPEN AFTER n UNITS OF TIME.

One cannot rely on an efficient distribution of the tasks at submission time. And one cannot wait for a machine to finish processing its tasks to rebalance the work. The only remaining option would be *a priori* load balancing: balancing the load before processing the tasks.

To be decentralized, we choose to investigate algorithms that balance optimally (or at least in a guaranteed way) the load of a subset of the processors. Obviously picking a large subset of the processors is essentially a centralized solution.

[3]The algorithm is defined on graph of dependent task. The critical path denotes the length of the longest sequence of operations that must be sequential. In the independent task model, the processing time of the longest task

Therefore we focus on algorithms that balance the work by pairs. The following theorem informs us that such an approach might have a poor guarantee in some cases.

Proposition 2. *A generic algorithm balancing optimally each pair of machine can induce an unbounded makespan.*

Proof: It is possible to construct a situation with 3 machines and 3 jobs where all pairs of machines have their load optimally distributed but the global load is not optimally distributed and can even be infinitely longer than the optimal one.

This situation is described in Figure II. Each job can be run fast, slow or very slow. When a pair of machines rebalances their load, there are two jobs, one of which can be run fast and one that can not. If the job that can be run fast is scheduled on its fast processor, then the remaining task must either be executed at very slow rate, or put two tasks on the same machine: one slow and one fast. Both solutions have a worst makespan than the original solution which completes in n time units. It is clear that the optimal schedule has a makespan of 1, but the algorithm is stuck in a solution of makespan n. ■

Cost	Machine A	Machine B	Machine C
Job 1	1	ⓝ	n^2
Job 2	n^2	1	ⓝ
Job 3	ⓝ	n^2	1

Table II
A SITUATION WITH 3 MACHINES AND 3 JOBS WHERE PAIR-WISE BALANCING LEADS TO AN UNBOUNDED MAKESPAN. THE OPTIMAL MAKESPAN IN THIS SITUATION IS 1 WITH JOB 1 ON MACHINE A, JOB 2 ON MACHINE B AND JOB 3 ON MACHINE C. HOWEVER WITH THE CIRCLED DISTRIBUTION, THE SCHEDULE OF ALL PAIRS OF MACHINES IS OPTIMAL, DESPITE THE MAKESPAN IS n.

This proposition indicates that there is little hope to find generic methods with constant approximation ratio, so we look at particular cases.

We would like to share a few remarks on the use and applicability of *a priori* algorithms before studying them further. First, we say the balancing is done before the execution of the tasks for simplicity. But it could be done after each task submission to address the lack of good greedy property or simply done periodically.

By running it periodically, an *a priori* load balancer can naturally take into account the dynamicity of the computing system, some tasks might be shorter or longer than predicted, and some tasks might dynamically be created on a processor. These scenarios are easily taken into account contrarily to models that balance tasks at submission time only. Also, we will see later that these load balancers might not converge to a unique solution, therefore it might be necessary to run the balancing algorithm concurrently with the application.

A common concern is that the algorithm might move tasks many time causing the data of the tasks to be continously bounced around the network. We showed previously in a workstealing based load balancer [14] that it is also possible to separate, within a runtime system, the load balancing itself and the data movement. This allows to optimize the data movements and to minimize the impact of moving the data many times in the computing system.

V. LOAD BALANCING PER TYPE OF JOB

In this section, the jobs are grouped by type. In each type, all the jobs have the same processing time: $\forall j, j' \in T,\ j$ has the same type as $j' \Rightarrow \forall i \in M,\ p_{i,j} = p_{i,j'}$. This distinction can easily be made in real systems where simple queries can represent most of the jobs of a system: even if the jobs are not exactly the same, their processing time is similar and only vary depending on which machine executes them.

We first study the case where there is only one type of job and provide an optimal decentralized algorithm, and then extend the algorithm to derive a guaranteed decentralized algorithm in the general case.

A. Balancing only one type of job

Algorithm 2: *Basic Greedy*

Data: machines m and i
Data: S, a distribution of jobs
$A := S(i) \cup S(m)$
$S(m) := \emptyset$
$S(i) := \emptyset$
while $A \neq \emptyset$ **do**
  let j be a job in A
  if $C(m) + p_{m,j} \leq C(i) + p_{i,j}$ **then**
    $S(m) := S(m) \cup \{j\}$
  else
    $S(i) := S(i) \cup \{j\}$
  $A := A \backslash \{j\}$

Algorithm 3: *OJTB*

Data: m, the host machine
Data: S, an initial distribution of the jobs
while *true* **do**
  Select randomly (uniformly) $i \in M$
  Distribute optimally the load of i and m with the *Basic Greedy*

The *One Job Type Balancing algorithm*, denoted as *OJTB* (Algorithm 3) is run by each machine. It is quite simple. It randomly chooses a target and balances the load of the two machines in an optimal way. Getting the optimal load balancing for two machines in this problem is ensured by *Basic Greedy* (Algorithm 2) thanks to the fact that the jobs'

cost is defined by the machine only. We prove that *OJTB* is optimal.

Lemma 3. Basic Greedy *provides an optimal distribution when there is only one type of jobs.*

Proof: Since there are only two machines and all the jobs are equivalent, the optimal solution is obviously given by a greedy algorithm using Earliest Completion Time. ■

Lemma 4. OJTB *(Algorithm 3) converges to a distribution S of the jobs which is optimal.*

Proof: Let $S(n)$ be the solution created after the n-th execution of the algorithm. Note that $C_{max}(S(n+1)) \leq C_{max}(S(n))$; If not, the balancing done between the two machines at step $n+1$ would not be optimal. Hence the function $C_{max}(S(n))$ is non-increasing. Let S be the distribution created by Algorithm 3 after an infinite number of executions.

As all the jobs have the same processing time we denote as p_i the processing time of any job on machine i. We can now represent $S(i)$ only by its cardinality which we denote as $N(i)$, $N(i) = |S(i)|$. We also have $C(i) = N(i)p_i$. Let S^* be an optimal distribution and $N^*(i) = |S^*(i)|$.

By contradiction, let us assume that $C_{max}(S) > C_{max}(S^*)$. There exists $i_{max} \in M$ such that $C(i_{max}) = C_{max}(S)$. In particular, $N(i_{max}) > N^*(i_{max})$, but this also implies that there exists i such that $N(i) < N^*(i)$ (because $\sum_{i \in M} N(i) = |J| = \sum_{i \in M} N^*(i)$). So at least one job can be moved from i_{max} to i to have a better local distribution (C_{max} would decrease). Hence the distribution S is not optimal for the pair of machine i_{max} and i but as Algorithm 3 has been applied an infinite number of times and because C_{max} is non-increasing, S should be optimal for i_{max} and i.

We have $C_{max}(S) \leq C_{max}(S^*)$, hence $C_{max}(S) = C_{max}(S^*)$. ■

B. Extension to multiple types of jobs

OJTB can be extended into the *Multiple Job Type Balancing* (*MJTB*) algorithm to balance multiple types of jobs; however, the performance guarantee of the algorithm becomes linear in the number of types of jobs (denoted as k in this section).

Theorem 5. MJTB *(Algorithm 4) applied to k types of jobs, converges to a distribution S of the jobs which is a k-approximation.*

Proof: Let T_l be the set of jobs of type l, if j and j' are in T_l, then $\forall i \in M, p_{i,j} = p_{i,j'}$. Moreover $\forall l, l'$ such that $l \neq l', \forall j \in T_l$ and $\forall j' \in T_{l'}, \exists i \in M, p_{i,j'} \neq p_{i,j}$. Let k be the number of distinct types of jobs.

Algorithm 3 is applied for all the types. Type l is optimally distributed so the makespan $C(T_l)$ of the distribution of T_l is smaller than OPT. When all the types of jobs have been partitioned, we have $C_{max} \leq \sum_{i=1}^{k} C(T_i) \leq k\text{OPT}$. ■

Algorithm 4: *MJTB*

Data: m, the host machine
Data: S, initial distribution of the jobs
Data: k, number of types of jobs
Result: $C_{max}(S) \leq k \cdot \text{OPT}$
while *true* **do**
  Select $i \in M$
  foreach l*: type of job* **do**
    Distribute the jobs of type l between i and m using *Basic Greedy*

VI. Load balancing with two types of machines

In this section, we limit the problem to two different clusters of identical machines M^1 and M^2 ($\forall i, i' \in M^1 (resp.\ M^2),\ \forall j \in J,\ p_{i,j} = p_{i',j}$). This is meaningful in practice since the advent of GPU-accelerated clusters. We develop a new algorithm with a proven approximation ratio of 2 under the assumption that the cost of any task is smaller than the optimal makespan for the problem, which is formally expressed as $\forall i \in M, \forall j \in J, p_{i,j} \leq \text{OPT}$. This hypothesis is realistic in the sense that there is typically no wild variation in what heterogeneous machines can do. For instance, GPUs used to be thought as a hundred time faster than CPUs. However, careful analysis show that the ratio is much smaller [19]. It also supposes that there is not a job which can only run on one cluster (and if this was the case, they could be assigned beforehand). Moreover, this hypothesis also suggests that there is a large amount of work, which is the starting point for creating a decentralized algorithm.

To simplify the notations, in this section we will denote as $p_{i,j}$ the cost of job j on cluster i. We first focus on the centralized version of the problem and then use this centralized algorithm as a stepping stone to create a decentralized one.

A. A centralized algorithm

This problem is a sub-problem of the scheduling problem of m unrelated machines which has been studied by Lenstra *et al.* [20]. They provide an algorithm with an approximation ratio of 2. However, the algorithm requires solving a linear program which seems difficult to decentralize reasonably. This is why we developed a new greedy algorithm to balance the load between two clusters.

The idea behind this new algorithm, called *CLB2C* (Algorithm 5) for *Centralized Load Balancing for Two Clusters*, is quite simple: we first sort the jobs according to the ratio $p_{1,j}/p_{2,j}$ so that the jobs which are executed faster on the first cluster are at the beginning of the list and the one executed faster on the second cluster are at the end of the

list. Then we evaluate the decision of placing the first job of the list on the machine of the first cluster with minimal makespan and placing the last job of the list on the second cluster on the machine with minimal makespan. The job placed is the one that minimizes the makespan of those two machines. That way, if a job is not placed on the cluster where it can be executed at its minimal cost, we know this choice does not have a significant impact on the overall makespan.

Algorithm 5: *CLB2C*

Data: M^1, The cluster of identical machines of type 1
Data: M^2, The cluster of identical machines of type 2
Data: J, The list of n jobs such that
$\forall j \in \{1, ..., n\}, p_{1,J(j)} \leq \text{OPT}$ and
$p_{2,J(j)} \leq \text{OPT}$
Result: A partition S of the jobs for each machine
Sort jobs in J in increasing order of $p_{1,j}/p_{2,j}$ ($J(j)$ denote the j^{th} job in J)
Initialize S with one empty set per machine
$j^1 := 1$
$j^2 := n$
while $j^1 \leq j^2$ **do**
  Select $i^1 \in M^1$ such that $C(i^1) = \min_{i \in M^1} C(i)$
  Select $i^2 \in M^2$ such that $C(i^2) = \min_{i \in M^2} C(i)$
  if $C(i^1) + p_{1,J(j^1)} \leq C(i^2) + p_{2,J(j^2)}$ **then**
    $S(i^1) := S(i^1) \cup \{J(j^1)\}$
    $j^1 := j^1 + 1$
  else
    $S(i^2) := S(i^2) \cup \{J(j^2)\}$
    $j^2 := j^2 - 1$
return S

Theorem 6. CLB2C *(Algorithm 5) is a* 2*-approximation algorithm.* $C_{max}(S) \leq 2\text{OPT}$.

Proof: Let i_{max} be a machine of $M = M^1 \cup M^2$ such that $C_{max}(S) = C(i_{max})$. We can suppose, without loss of generality, that $i_{max} \in M^1$. Let j_{max} be the last job placed on i_{max}. Let S' be the incomplete partition of J just before the choice is made by Algorithm 5 to place j_{max} on i_{max}. Let C^1 be $C(S', i_{max})$ and C^2 be such that $C^2 = \min_{i \in M^2} C(S', i)$. Note that C^1 has also the property $C^1 = \min_{i \in M^1} C(S', i)$. Indeed, the next job placed is j_{max}, which is placed on i_{max}, so $C^1 = \min_{i \in M^1} C(S', i)$.

Let us compare $\min(C^1, C^2)$ to OPT. Let S'^1 be the jobs placed on cluster 1 in S', and S'^2 the ones placed on cluster 2, $S'^k = \bigcup_{i \in M^k} S'(i)$.

To compare C^1 and C^2, we will use the notion of work, which is defined as the processing time of the jobs assigned to the machines. The work done on the cluster 1, W^1, has the property $W^1 \geq |M^1|C^1$, similarly, the work W^2 done on cluster 2 is such that $W^2 \geq |M^2|C^2$. Let S^* be an optimal solution with S^{*1} the set of jobs placed on cluster 1 and S^{*2} the set of jobs placed on cluster 2. The work done on cluster 1 with an optimal solution is denoted as W^{*1}, the work done on cluster 2 is denoted as W^{*2}

To create an optimal solution from S', the jobs $J^1 = S'^1 \cap S^{*2}$ should be moved from the cluster 1 to the cluster 2, and the jobs $J^2 = S'^2 \cap S^{*1}$ from cluster 2 to cluster 1. We have the equations $W^{*1} = W^1 - \sum_{j \in J^1} p_{1,j} + \sum_{j \in J^2} p_{1,j}$ and $W^{*2} = W^2 - \sum_{j \in J^2} p_{2,j} + \sum_{j \in J^1} p_{2,j}$.

By contradiction, let us assume that $W^{*1} < W^1$ and $W^{*2} < W^2$, from the two previous equations we deduce

$$\sum_{j \in J^2} p_{1,j} < \sum_{j \in J^1} p_{1,j} \text{ and } \sum_{j \in J^1} p_{2,j} < \sum_{j \in J^2} p_{2,j} \quad (1)$$

Let $\alpha = p_{1,j_{max}}/p_{2,j_{max}}$. Because the algorithm sorts the jobs, we have $\forall j \in J^1,\ p_{1,j}/p_{2,j} \leq \alpha$ and $\forall j \in J^2,\ p_{1,j}/p_{2,j} \geq \alpha$ and with Equation 1 we have

$$\sum_{j \in J^2} p_{1,j} < \sum_{j \in J^1} p_{1,j} \leq \alpha \sum_{j \in J^1} p_{2,j} < \alpha \sum_{j \in J^2} p_{2,j} \quad (2)$$

But we also have $\sum_{j \in J^2} p_{1,j} \geq \alpha \sum_{j \in J^2} p_{2,j}$, which is in contradiction with Equation 2. From this, we deduce that $W^{*1} \geq W^1$ or $W^{*2} \geq W^2$, in particular, $\text{OPT} \geq \max(W^{*1}/|M^1|, W^{*2}/|M^2|) \geq \min(W^1/|M^1|, W^2/|M^2|) \geq \min(C^1, C^2)$.

In conclusion, $\min(C^1, C^2) \leq \text{OPT}$.

Let j' be the job compared to j_{max} when j_{max} is placed. Because j_{max} has been placed on i_{max} from the first cluster, we have $C^1 + p_{1,j_{max}} \leq C^2 + p_{2,j'}$.

If $C^1 \leq C^2$, $C^1 \leq \text{OPT}$ and $p_{1,j_{max}} \leq \text{OPT}$ so $C_{max}(S) \leq 2\text{OPT}$. Else $C^2 < C^1$, so we have $C^2 \leq \text{OPT}$, $p_{2,j'} \leq \text{OPT}$ and $C^1 + p_{1,j_{max}} \leq C^2 + p_{2,j'}$ so $C_{max}(S) \leq 2\text{OPT}$.

In both cases, $C_{max}(S) \leq 2\text{OPT}$. ■

B. Decentralized algorithm

CLB2C is the base for a decentralized algorithm to balance the load of machines on two clusters of uniform machines.

Indeed, like the peer to peer approach from the *Work Stealing* algorithm the algorithm *DLB2C* (for *Decentralized Load Balancing for Two Clusters*, Algorithm 7) is executed on each machine: each machine randomly selects a target (a machine), and if both machines are from the same cluster, a *Greedy Load Balancing* is applied, if both machines are from different clusters, *CLB2C* (Algorithm 5) is used to balance both machines (considering two sub-clusters of one machine each).

This algorithm uses *Greedy Load Balancing* to balance two machines from the same cluster (Algorithm 6). *DLB2C* ensures an interesting property: when the situation is stable, the makespan of the job distribution is bounded by twice the optimal makespan.

Algorithm 6: *Greedy Load Balancing*

Data: m^1, m^2 machines of the same cluster to balance
Data: S distribution of jobs
$A = S(m^1) \cup S(m^2)$
if m^1 *is in cluster 1* **then**
  Sort jobs in A in increasing order of $p_{1,j}/p_{2,j}$
else
  Sort jobs in A in increasing order of $p_{2,j}/p_{1,j}$
Initialize S with one empty set for m^1 and m^2
while $A \neq \emptyset$ **do**
  j first job in A
  if $C(m^1) \leq C(m^2)$ **then**
    $S(m^1) := S(m^1) \cup \{j\}$
  else
    $S(m^2) := S(m^2) \cup \{j\}$
  $A := A \backslash \{j\}$
return S

Algorithm 7: *DLB2C*

Data: m, the host machine
Data: $J(m)$ Set of jobs initially on m
Result: $C(J(m)) \leq 2 * \mathrm{OPT}$
while *true* **do**
  Select $i \in M$
  if i *is in the same cluster as* m **then**
    Apply *Greedy Load Balancing* to i and m
  else
    $M_1 := \{m\}$
    $M_2 := \{i\}$
    Apply *CLB2C* to M_1 and M_2 with the set of jobs $J(m) \cup J(i)$

Theorem 7. *If the distribution S provided by the execution of Algorithm 7 becomes stable (for every pair of machine, the algorithm does not move any job), S is such that $C_{max}(S) \leq 2\mathrm{OPT}$.*

Proof: Let S^1 be the set of jobs assigned to cluster M^1 ($S^1 = \bigcup_{m \in M^1} S(m)$) and S^2 the set of jobs assigned to cluster M^2.

We first show that the jobs in S^2 have a ratio $p_{1,j}/p_{2,j}$ larger the ones of the jobs in S^1. Let j^1 be such that $p_{1,j^1}/p_{2,j^1} = \max_{j \in S^1} p_{1,j}/p_{2,j}$ and j^2 be $p_{1,j^2}/p_{2,j^2} = \min_{j \in S^2} p_{1,j}/p_{2,j}$.

By contradiction let us assume that $p_{1,j^2}/p_{2,j^2} < p_{1,j^1}/p_{2,j^1}$. Let us denote by m^1 the machine on which j^1 is scheduled, and m^2 the machine on which j^2 is scheduled. Because the distributed *DLB2C* is running on all machines, it has been executed for all pairs of machines and does not change the solution (the distribution is stable), it has been executed to balance m^1 and m^2 in particular. As shown in the proof of Theorem 6, there exists α such that $\forall j \in S(m^1),\ \forall j' \in S(m^2), p_{1,j}/p_{2,j} \leq \alpha$ and $p_{1,j'}/p_{2,j'} \geq \alpha$. From these we conclude that

$$\forall j^1 \in S^1, \forall j^2 \in S^2,\ p_{1,j^1}/p_{2,j^1} \leq p_{1,j^2}/p_{2,j^2} \quad (3)$$

Let i_{max} be a machine such that $C_{max}(S) = C(i_{max})$. Without any loss of generality, we assume that $i_{max} \in M^1$. Let j_{max} be a job assigned to i_{max} such that $p_{1,j_{max}}/p_{2,j_{max}} = \max_{j \in S_{i_{max}}} p_{1,j}/p_{2,j}$. Let $C^1 = C(i_{max}) - p_{1,j_{max}}$. Since *Greedy Load Balancing* has been applied between the machines of the same cluster, we have the property

$$\forall i \in M^1,\ C^1 \leq C(i) \quad (4)$$

Let C^2 and i^2 be such that $C^2 = C(i^2) = \min_{i \in M^2} C(i)$. Using the same reasoning by contradiction as in the proof of Theorem 6 with $\alpha = p_{1,j^1}/p_{2,j^1}$, we get $\min(C^1, C^2) \leq \mathrm{OPT}$.

Let us consider an execution of *CLB2C* between i_{max} and i^2.

- if j_{max} is not the last job placed by the algorithm, then j_{max} has been compared to j which has been placed afterward, so $C^1 + p_{1,j_{max}} \leq C^2$ but $C^1 + p_{1,j_{max}} \geq C^2$ (because $C^1 + p_{1,j_{max}} = C_{max}(S)$).
  Hence $C^1 \leq C^2$ so we deduce $C^1 \leq \mathrm{OPT}$ and with $p_{1,j_{max}} \leq \mathrm{OPT}$ we have $C_{max}(S) \leq 2\mathrm{OPT}$
- if j_{max} is the last job placed by the algorithm, we have $C^1 + p_{1,j_{max}} \leq C^2 + p_{2,j_{max}}$.
  Hence $C_{max}(S) \leq 2\mathrm{OPT}$.

In both cases we have $C_{max}(S) \leq 2\mathrm{OPT}$. ■

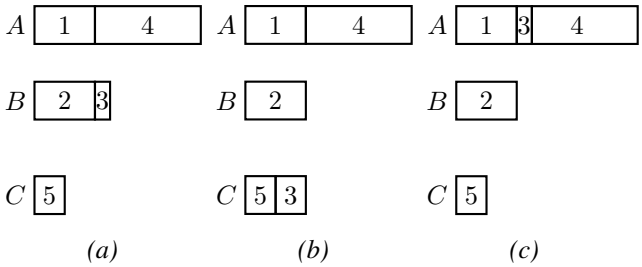
VII. Dynamic equilibrium of *DLB2C*

DLB2C has a guaranteed performance according to Theorem 7 only if the schedule it generates is stable; that is to say when there are no more pair wise exchange possible. The proof can be extended to the non stable cases as long as the machine i_{max} (as defined in the proof) can not exchange jobs with any other machine. Unfortunately, the algorithm might not converge. In the remainder of this section, we study the properties of the dynamic equilibrium of *DLB2C* using a markovian model and simulations. First we show the algorithm might not reach a static equilibrium.

Proposition 8. *In some cases,* DLB2C *does not converge to a stable solution.*

Proof: Figure 1 presents an instance of the problem with 5 jobs and 3 machines organized in 2 clusters. The details of the instance is given in Figure 1(d). With a particular initial distribution of the jobs to the machines, the algorithm is trapped in a cycle between the three schedules given in Figure 1(a), 1(b) and 1(c).

■



(a) *(b)* *(c)*

Job	Cluster 1 = {A,B}	Cluster 2 = {C}
1	4ϵ	1
2	4ϵ	1
3	ϵ	2ϵ
4	$1-8\epsilon$	1
5	1	2ϵ

(d) Cost of each job

Figure 1. With the instance given in (d), *DLB2C* can get trappped in a cycle and never reach stability. Once in schedule (a) *CLB2C* will balance machines B and C and reach schedule (b). Then the only possible move is to balance A and C which leads to schedule (c). Schedule (a) is then obtained by balancing A and B.

A. Modelling the one cluster case

We consider the use of algorithm *DLB2C* on only one cluster of m machines. Since there is only one cluster, the amount of work is fixed and equal to $\sum p_i$, and the longest task is of size $p_{max} = \max p_i$. For the purpose of this section, we represent the state of the system at any time by an integer load vector $L = \{L_1, \ldots, L_m\}$, L_i being the load of machine i. A load vector is valid under two conditions $\forall j, L_j \geq 0$ and $\sum_j L_j = \sum p_i$. The makespan of a load vector is simply the value of its largest dimension $C_{max} = \max L_j$.

The *DLB2C* algorithm can transform a valid load vector L into another valid load vector L' under the conditions:

- Only two machines are impacted: $\forall j \neq j_1, j_2, L_j = L'_j$
- The load is conserved: $L_{j_1} + L_{j_2} = L'_{j_1} + L'_{j_2}$
- The new imbalance is bounded: $|L'_{j_1} - L'_{j_2}| \leq p_{max}$

This relation allows us to represent an instance of the one cluster case by a directed graph where the vertices are valid load vectors and where there is a directed edge between L and L' if *DLB2C* can transform L into L'. Notice that this model relaxes the notion of tasks in the sense that there is no more task assignment but simply a load assignment. That is to say, there might not exist an actual task assignment that maps to that particular load. However, one can easily convince herself that the following stays meaningful.

A load vector is *perfectly balanced* if $\forall j, \left\lfloor \frac{\sum_{i=1}^{m} p_i}{m} \right\rfloor \leq L_j \leq \left\lceil \frac{\sum p_i}{m} \right\rceil$. Notice that all the perfectly balanced load vectors are equivalent up to a reordering of the dimensions of the load vector. The directed graph constructed above can be decomposed in many strongly connected component. However, only one component has no outgoing edges, we call this component the *sink* component.

Theorem 9. *The* sink *component is the only strongly connected component that does not have any outgoing edges. And this component contains the perfectly balanced state(s).*

Proof: First notice that all the perfectly balanced vectors are within the same component since they have the same number of dimension with value $\left\lceil \frac{\sum p_i}{m} \right\rceil$ and with value $\left\lfloor \frac{\sum p_i}{m} \right\rfloor$. One can go from one perfectly balanced vector to an other one by appropriately balancing the offending dimensions. Therefore, one can reduce all the perfectly balanced state to a single one.

The theorem is proven by showing that from each load vector there is a path to the perfectly balanced state. Therefore, the *sink* component is the only one without outgoing edges.

Let L be a load vector with $C_{max} > \left\lceil \frac{\sum p_i}{m} \right\rceil$. Let j_{max} be such that $L_{j_{max}} = C_{max}$ and j_{min} be such that $L_{j_{min}} = \min_j L_j$. When j_{min} and j_{max} are balanced, the load vector can become L' with $L'_{j_{max}} = \left\lceil \frac{L_{j_{max}} + L_{j_{min}}}{2} \right\rceil$, $L'_{j_{min}} = L_{j_{max}} + L_{j_{min}} - L'_{j_{max}}$ and $\forall j \neq j_{max}, j_{min} L'_j = L_j$.

If the makespan of L' is $\left\lceil \frac{\sum p_i}{m} \right\rceil$, then the theorem is proven. Otherwise, there are two cases: either the makespan strictly decreased $\max_j L'_j < C_{max}$. Or it stayed the same $\max_j L'_j = C_{max}$; but in this case the number of values in L' equal to C_{max} is one less than in L, by repeating this process, the makespan will eventually decrease. This creates a path from L to a perfectly balanced load vector. ■

Theorem 10. *All the load vectors in the sink component have a makespan smaller or equal to* $\frac{\sum p_i}{m} + \frac{m-1}{2} p_{max}$.

Proof: Let us assume that X is the maximum makespan of a load vector L of the sink component. Because of the condition that the new imbalance is bounded, the relationship encoded by the edges of the graph there exist a machine j with a load greater than (or equal to) $X - p_{max}$ (could be 0). If that machine could reach a higher load without taking work from the machine that reaches the maximum makespan, X would not be maximal. Therefore, we can recursively find machines with load equal to $\max(X - kp_{max}, 0)$ for all $k \in [0..m-1]$. But, $\sum p_i \geq \sum_{k=0}^{m-1} X - kp_{max}$ thus $X \leq \frac{\sum p_i}{m} + \frac{m-1}{2} p_{max}$. ■

This analysis tells us that, eventually, the state of the system will enter the *sink* component. Even if there exists states in that component with really large makespans, the system can only reach such a state by carefully choosing pairs of machines to balance to reach this worst case scenario. In practice, the pairs of machines are chosen randomly and this worst case scenario is unlikely to be reached.

We transform the graph of states into a Markov chain

by introducing probabilities for transitioning from one state to the other. We set the probabilities so that the rebalanced pair of machines j_1, j_2 is chosen uniformly. Once this pair is chosen and balanced, the remaining imbalance is also uniformly chosen in $\{0, \ldots, p_{max}\}$. Because there is only one sink component to the underlying graph, we can reduce the analysis to that component. The stationary distribution of the chain is well defined.

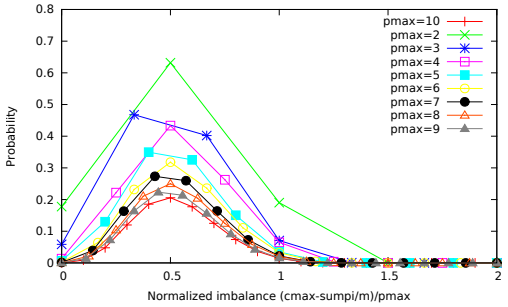
We expanded the graph for various parameters of m, p_{max} and $\sum p_i$. We set $\sum p_i$ so that the maximum imbalance given in Theorem 10 can be reached. We numerically computed the stationary distribution of the Markov chain using an iterative method. With the stationary distribution, we computed the probability distribution of the makespan in the steady state. The computational cost quickly increases with m and p_{max}, making larger run prohibitively long. However we will see that we can safely extrapolate our conclusions from relatively small values.

Figure 2 presents the distribution of makespan numerically computed for $m = 6$ and various values of p_{max} and for $p_{max} = 4$ and various values of m. The makespan is normalized so that the X-axis shows the deviation of the makespan from the optimally balanced case as a factor of p_{max}. It is striking that the distributions are unimodal and the mode is 0.5. All the distributions are qualitatively the same. Increasing p_{max} only smoothen the shape of the distribution (see Figure 2(a)). This allows us to confidently look at a reasonably small value of p_{max} in Figure 2(b). When the number of machine increases, the probability that the makespan is before the mode decreases and that it is after the mode increases. However, in all computed cases, $C_{max} \leq \frac{\sum p_i}{m} + 1.5 p_{max}$ with very high probability.

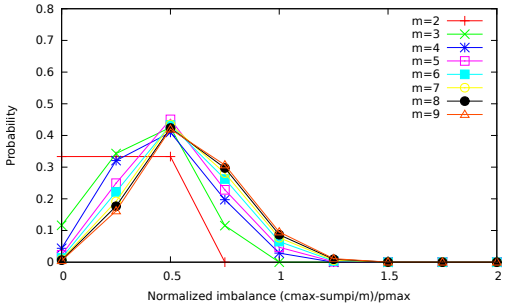
B. Simulation

To perform an analysis of the performance of the algorithm in the heterogeneous case and compare them to the one obtained in the homogeneous one, we developed a simulator that implements *CLB2C* and *DLB2C* (Algorithms 5 and 7). We focus on two clusters of similar size: there are often one or two CPU and one GPU per machine. As the clusters are composed of homogeneous machines, and because the clusters behave differently for each job, only the size of the cluster matter; The time to execute a job on each cluster is a characteristic of the job itself. We randomly generate some jobs with two arbitrary speeds (one for each cluster) and size, then we distribute them randomly among the machines (details are given per experiments). For each iteration of the simulation, a pair of machine is randomly chosen and then balanced using the decentralized algorithm. As the partition can cycle, each simulation is stopped when the number of iterations is bigger than a threshold depending on the number of machines.

We computed the makespan of the solution obtained as well as the stability of the partition for two clusters of



(a) $m = 6$



(b) $pmax = 4$

Figure 2. Probability density function of the makespan. The x-axis presents normalized imbalance, *i.e.*, $C_{max} = \frac{\sum p_i}{m} + X p_{max}$.

64 and 32 machines and for one homogeneous cluster of 96 machines. For each experiment, 768 jobs were used, they have length uniformely taken in the interval [1;1000]. The results presented in Figure 3 show that the stability is rarely obtained in the heterogeneous case. However, if we look at the evolution of C_{max} over time (Figure 4), we observe runs that quickly reach a value and oscillate around it. Even if there is no stability in the run, variations stay close to the minimum value the run has reached. We can see similarities between the runs on an homogeneous cluster and two different clusters: both have some stable runs and some cycles, they both quickly reach a makespan a lot smaller than the initial one. The variations are more intense on two clusters as there are more possibilities for better solutions. But the homogeneous and heterogeneous cases are qualitatively similar.

We now focus on the number of exchanges per machine needed to reach a makespan better than 150% of the makespan obtained with *CLB2C* (we denote this value as $1.5cent$) for the first time (Figure 5). This threshold is hopefully much smaller than 3 times the optimal makespan (*CLB2C* is a 2-approximation). For all experiments, the jobs were created using a uniform distribution on [1;1000] to define their length. In the case of two clusters of 64 and 32 machines, 90% of runs reach this value in less than 4 exchanges per machines even if the number of jobs to balance increase greatly (for 1 to 32 jobs per machine). This also means that this value is reach with less than 4

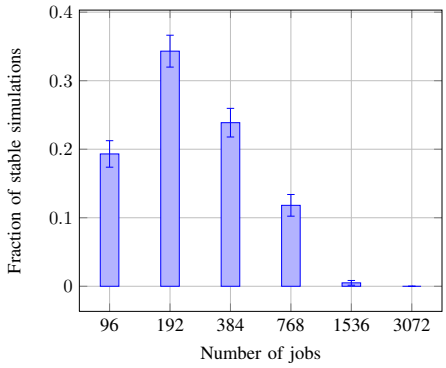
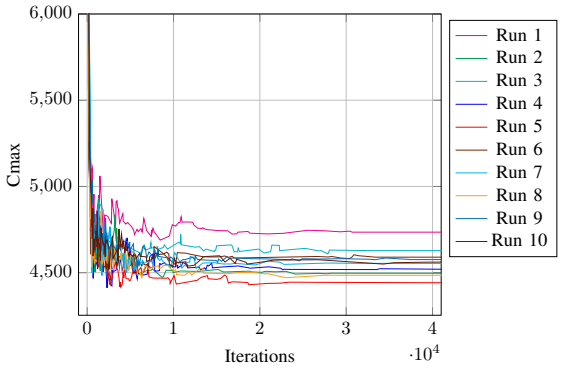
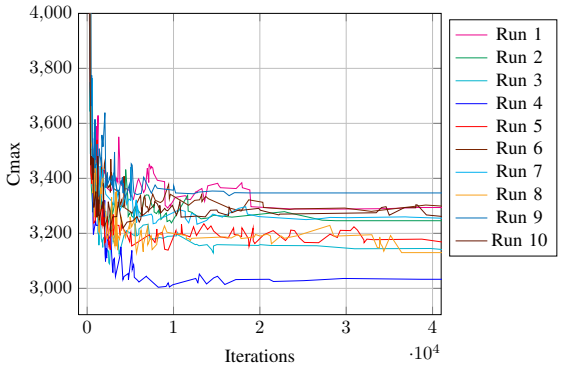


Figure 3. Stability of *DLB2C* on two clusters of size 64 and 32, length of each job taken uniformely on the interval [1;1000]. Most instances do not reach a stable solution.



(a) 1 cluster of 96 machines. 10 different runs with identical parameters: 768 jobs with a length uniformely taken between 1 and 1000.



(b) 2 clusters of 64 and 32 machines. 10 different runs with identical parameters: 768 jobs with a length uniformely taken between 1 and 1000.

Figure 4. Evolution of C_{max} in multiple simulations. The one cluster and two clusters cases do not show significant differences.

communications per machine (about 400 communications). When the size of the cluster increases to 512 and 256 machines (8 times more), 90% of the machines reach $1.5cent$ for the first time in less than 5 exchanges per machine. The shape of those results is similar to the one obtained for one homogeneous cluster of 96 machines, except for the fact that the initial distribution of jobs in the homogeneous case is more likely to be close to an optimal situation: there are less possibilities for the algorithm to optimize the distribution.

VIII. Conclusion

In this work we studied how to balance the load of an heterogeneous cluster using decentralized algorithms. Classical submission time methods and work stealing based methods are unlikely to achieve desirable properties. Also general methods are unlikely to be found, we investigated two different cases. *MJTB* is useful when the number of types of jobs is limited and has an approximation ratio equal to the number of types of jobs considered. In the case of a cluster with two types of machines (such as a CPU-GPU cluster), we proposed *DLB2C* which is a 2-approximation ratio if the algorithm converges. However, we showed that it might not converge and we studied its dynamic equilibrium using a Markov model and simulations. The Markov model provides a theoretical distribution of the makespan when there is only one cluster which highlights that the makespan typically stays within a factor 2 of the optimal. The simulation shows that the one cluster and two cluster case behave similarly. It also shows that the two cluster cases reaches good solutions within a few iterations. Even without converging, *DLB2C* is a sensible algorithm to use.

Providing strong theoretical evidence that the distributions of the makespan obtained by *DLB2C* is suitable and its extension to more than two clusters of machines are possible future works. Of course, designing another decentralized algorithm that converges to a provably good solution remains our main goal. Also the current model ignores the amount of tasks exchanged; minimizing the number of tasks exchanged (or network usage) would certainly be of interest.

References

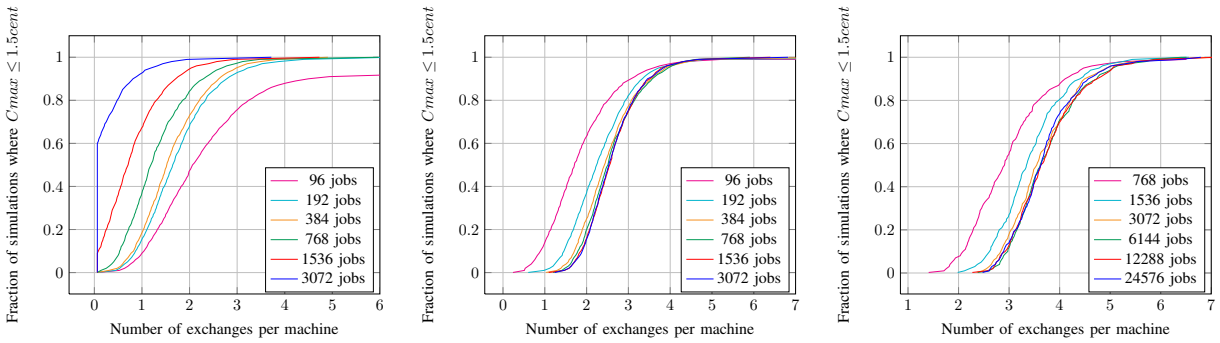
[1] Michael A. Bender and Michael O. Rabin. Online scheduling of parallel programs on heterogeneous systems with applications to Cilk. *Theory of Computing Systems, Special Issue on SPAA*, 35(3):289–304, 2002.

[2] P. Berenbrink, A. Brinkmann, T. Friedetzky, and L. Nagel. Balls into non-uniform bins. In *Proc. of IPDPS*, 2010.

[3] Petra Berenbrink, André Brinkmann, Tom Friedetzky, and Lars Nagel. Balls into non-uniform bins. *J. Parallel Distrib. Comput.*, 74(2):2065–2076, February 2014.

[4] Petra Berenbrink, Kamyar Khodamoradi, Thomas Sauerwald, and Alexandre Stauffer. Balls-into-bins with nearly optimal load distribution. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA, pages 326–335, 2013.

[5] Robert D. Blumofe. Scheduling multithreaded computations by work stealing. In *FOCS*, pages 356–368, 1994.



(a) 1 cluster of 96 machines (b) 2 clusters of 64 and 32 machines (c) 2 clusters of 512 and 256 machines

Figure 5. Study of time to reach a makespan less than $1.5cent$. The number of iterations is normalized with respect to the number of machines. Most cases reaches the threshold within 5 pairwise exchanges per machine.

[6] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, page 207–216, 1995.

[7] F. Warren Burton and M. Ronan Sleep. Executing functional programs on a virtual tree of processors. In *Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture*, FPCA '81, pages 187–194, 1981.

[8] Lin Chen, Deshi Ye, and Guochuan Zhang. Online scheduling on a CPU-GPU cluster. In T-H.Hubert Chan, LapChi Lau, and Luca Trevisan, editors, *Theory and Applications of Models of Computation*, volume 7876 of *Lecture Notes in Computer Science*, pages 1–9. Springer Berlin Heidelberg, 2013.

[9] Martin Gairing, Burkhard Monien, and Andrea Woclaw. *Automata, Languages and Programming*, volume 3580, chapter A Faster Combinatorial Approximation Algorithm for Scheduling Unrelated Parallel Machines, pages 828–839. Springer, aug 2005.

[10] Michael R. Garey and David S. Johnson. "Strong" NP-completeness results: Motivation, examples, and implications. *J. ACM*, 25(3):499–508, 1978.

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

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

[13] Ronald L. Graham, Eugene L. Lawler, Jan Karel Lenstra, and Alexander. H. G. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. *Annals of discrete mathematics*, 5(2):287–326, 1979.

[14] Timothy D. R. Hartley, Erik Saule, and Umit V. Catalyurek. Improving performance of adaptive component-based dataflow middleware. *Parallel Computing*, 38(6-7):289–309, 2012.

[15] Dorit S. Hochbaum and David B. Shmoys. Using dual approximation algorithms for scheduling problems: Practical and theoretical results. *Journal of ACM*, 34:144–162, 1987.

[16] Ralf Hoffmann, Matthias Korch, and Thomas Rauber. Performance evaluation of task pools based on hardware synchronization. In *Proc. of Super Computing*, 2004.

[17] Ellis Horowitz and Sartaj Sahni. Exact and approximate algorithms for scheduling nonidentical processors. *J. ACM*, 23(2):317–327, April 1976.

[18] Eugene L. Lawler and Jacques Labetoulle. On preemptive scheduling of unrelated parallel processors by linear programming. *J. ACM*, 25(4):612–619, 1978.

[19] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Ngyuen, Nadathur Satish, Mikhail Smelyanskiy, Srinivis Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the 100x GPU vs. CPU myth: An evaluation of throughput computing on CPU and GPU. In *ISCA*, 2010.

[20] Jan Karel Lenstra, David B. Shmoys, and Éva Tardos. Approximation algorithms for scheduling unrelated parallel machines. *Mathematical Programming*, 46:259–271, 1990.

[21] Marc Tchiboukdjian, Nicolas Gast, Denis Trystram, Jean-Louis Roch, and Julien Bernard. A tighter analysis of work stealing. In *International Symposium on Algorithms and Computation (ISAAC)*, 2010.