SCHEDULING ALGORITHMS FOR DATA REDISTRIBUTION AND LOAD-BALANCING ON MASTER-SLAVE PLATFORMS

LORIS MARCHAL, VERONIKA REHN, YVES ROBERT, FRÉDÉRIC VIVIEN
LIP, ENS Lyon, 46 allée d’Italie
69364 Lyon, Cedex 07, France

Received September 2006
Revised January 2007
Communicated by Guest Editors

ABSTRACT
In this work we are interested in the problem of scheduling and redistributing data on master-slave platforms. We consider the case were the workers possess initial loads, some of which having to be redistributed in order to balance their completion times. We assume that the data consists of independent and identical tasks. We prove the NP completeness of the problem for fully heterogeneous platforms. Also, we present optimal polynomial algorithms for special important topologies: a simple greedy algorithm for homogeneous star-networks, and a more complicated algorithm for platforms with homogeneous communication links and heterogeneous workers.

Keywords: Master-slave platform, scheduling, data redistribution, one-port model, independent tasks, divisible load theory.

1. Introduction
In this work we consider the problem of scheduling and redistributing data on master-slave architectures in star topologies. Because of variations in the resource performance (CPU speed or communication bandwidth), or because of unbalanced amounts of current load on the workers, data must be redistributed between the participating processors, so that the updated load is better balanced in terms that the overall processing finishes earlier.

We adopt the following abstract view of our problem. There are $m + 1$ participating processors $P_0, P_1, \ldots, P_m$, where $P_0$ is the master. Each processor $P_k$, $1 \leq k \leq m$ initially holds $L_k$ data items. During our scheduling process we try to determine which processor $P_i$ should send some data to another worker $P_j$ to equilibrate their finishing times. The goal is to minimize the global makespan, that is the time until each processor has finished to process its data. Furthermore we suppose that each communication link is fully bidirectional, with the same bandwidth for receptions and sendings. This assumption is quite realistic in practice, and does not change the complexity of the scheduling problem, which we prove NP-complete in the strong sense.

We assume that data items consist in independent and uniform (same-size) tasks. There are many practical applications who use fixed identical and independent tasks. A famous example is BOINC [1], the Berkeley Open Infrastructure for Network Computing, an open-source software platform for volunteer computing. It works as a centralized scheduler that distributes tasks for participating applications. These projects consists in the treatment of computation extensive and expensive scientific problems of multiple domains, such as biology, chemistry or mathematics. SETI@home [2] for example uses the accumulated computation power for the search of extraterrestrial intelligence. In the astrophysical domain, Einstein@home [3] searches for spinning neutron stars using data from the LIGO and GEO gravitational wave detectors. To get an idea of the task dimensions, in this project a task is about 12 MB and requires between 5 and 24 hours of dedicated computation. Also, from a theoretical viewpoint, the scheduling problem is obviously NP complete when tasks have different sizes (trivial reduction from 2-PARTITION [4], which provides yet another reason to restrict to same-size tasks).

As already mentioned, we suppose that all data are initially situated on the workers, which leads us to a kind of redistribution problem. Existing redistribution algorithms have a different objective. Neither do they care how the degree of imbalance is determined, nor do they include the computation phase in their optimizations. They expect that a load-balancing algorithm has already taken place. After the load-balancing phase, a redistribution algorithm determines the required communications and organizes them in minimal time. We could use such an approach: redistribute the data first, and then enter a purely computational phase. But our problem is more complicated as we suppose that communication and computation can overlap, i.e., every worker can start computing its initial data while the redistribution process takes place.

To summarize our problem: as the participating workers are not equally charged and/or because of different resource performance, they might not finish their computation processes at the same time. We are looking for algorithms to redistribute the loads in order to finish the whole computation process in minimal time. We enforce the hypothesis that charged workers can compute at the same time as they communicate.

The rest of this paper is organized as follows. The problem framework is detailed in Section 2. In Section 3 we discuss the case of general platforms. We are able to prove the NP-completeness for the general case of the problem, and the polynomiality of a restricted instance. The following sections consider some particular platforms: an optimal algorithm for homogeneous star networks is presented in Section 4. An optimal algorithm for platforms with homogenous communication links and heterogeneous workers is detailed in Section 5. Section 6 briefly presents some related work. Finally, we give some conclusions in Section 7.

2. Framework

We consider a star network $S = P_0, P_1, \ldots, P_m$ shown in Figure 1. The processor $P_0$ is the master and the $m$ remaining processors $P_i, 1 \leq i \leq m$, are workers. The initial data are distributed on the workers, so every worker $P_i$ possesses a number $L_i$ of initial tasks. All tasks are independent and identical. As we assume a linear cost
model, each worker $P_i$ has a (relative) computing power $w_i$ for the computation of one task: it takes $X.w_i$ time units to execute $X$ tasks on the worker $P_i$. The master $P_0$ can communicate with each worker $P_i$ via a communication link. A worker $P_i$ can send some tasks via the master to another worker $P_j$ to decrement its execution time. It takes $X.c_i$ time units to send $X$ units of load from $P_i$ to $P_0$ and $X.c_j$ time units to send these $X$ units from $P_0$ to a worker $P_j$. Without loss of generality we assume that the master is not computing, and only communicating.

![Fig. 1. Example of a star network.](image1)

The platforms discussed in sections 4 and 5 are a special cases of star networks: all communication links have the same characteristics, i.e., $c_i = c$ for each processor $P_i$, $1 \leq i \leq k$. Such a platform is called a bus network as it has homogeneous communication links.

We use the bidirectional one-port model for communication. This means that the master can only send data to, and receive data from, a single worker at a given time-step. But it can simultaneously receive data and send one. A given worker cannot start an execution before it has terminated the reception of the message from the master; similarly, it cannot start sending the results back to the master before finishing the computation.

The objective function is to minimize the makespan, that is the time at which all loads have been processed.

<table>
<thead>
<tr>
<th>Worker</th>
<th>$c$</th>
<th>$w$</th>
<th>load</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>1</td>
<td>1</td>
<td>13</td>
</tr>
<tr>
<td>$P_2$</td>
<td>8</td>
<td>1</td>
<td>13</td>
</tr>
<tr>
<td>$P_3$</td>
<td>1</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>$P_4$</td>
<td>1</td>
<td>10</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 1. Platform parameters.

![Fig. 2. Example of an optimal schedule on a heterogeneous platform, where a sending worker also receives a task.](image2)

3. General platforms

3.1. NP-completeness

One of the main difficulties seems to be the fact that we cannot partition the
workers into disjoint sets of senders and receivers. There exist situations where, to minimize the global makespan, it is useful that sending workers also receive tasks. Consider the following example. We have four workers (see Table 1 for their parameters). An optimal solution is shown in Figure 2, with a makespan $M = 12$:

Workers $P_3$ and $P_4$ do not own any task, and they are computing very slowly. So each of them can compute exactly one task. Worker $P_1$, which is a fast processor and communicator, sends them their tasks and receives later another task from worker $P_2$ that it can compute just in time. Note that worker $P_1$ is both sending and receiving tasks. Trying to solve the problem under the constraint that no worker also sends and receives, it is not feasible to achieve a makespan of 12. Worker $P_2$ has to send one task either to worker $P_3$ or to worker $P_4$. Sending and receiving this task takes 9 time units. Consequently the processing of this task can not finish earlier than time $t = 18$.

**Definition 1** (Scheduling Problem SP). Let $N$ be a star-network with one special processor $P_0$ called “master” and $m$ workers. Let $n$ be the number of identical tasks distributed to the workers. For each worker $P_i$, let $L_i$ be its initial number of tasks, and $w_i$ be its computation time for one task. Each communication link, link$_i$, has an associated communication time $c_i$ for the transmission of one task. Finally let $T$ be a deadline. The decision problem of SP is: “Is it possible to redistribute the tasks and to process them in time $T$?”.

**Theorem 1.** SP is NP-complete in the strong sense.

For the proof we refer to the companion research report [5].

### 3.2. Polynomiality when computations are neglected

A major difficulty of the problem is the overlap of computation and the redistribution process. In this section we provide an optimal polynomial algorithm when neglecting the computations.

Without computations we get a classical data redistribution problem, where we can suppose that we already know the imbalance of the system. More precisely, we adopt the following abstract view of the new problem: the $m$ participating workers $P_1, P_2, \ldots P_m$ hold their initial uniform tasks $L_i$, $1 \leq i \leq m$. For a worker $P_i$ the chosen algorithm for the computation of the imbalance has decided that the new load should be $L_i - \delta_i$. If $\delta_i > 0$, this means that $P_i$ is overloaded and it has to send $\delta_i$ tasks to some other processors. If $\delta_i < 0$, $P_i$ is underloaded and it has to receive $-\delta_i$ tasks from other workers. We have heterogeneous communication links and all sent tasks pass by the master. So the goal is to determine the order of senders and receivers to redistribute the tasks in minimal time.

**Theorem 2.** Knowing the imbalance $\delta_i$ of each processor, an optimal solution for heterogeneous star-platforms is to order the senders by non-decreasing $c_i$-values and the receivers by non-increasing order of $c_i$-values.

**Proof.** To prove that the scheme described by Theorem 2 returns an optimal schedule, we take a schedule $S'$ computed by this scheme. Then we take any other schedule $S$. We are going to transform $S$ in two steps into our schedule $S'$ and prove that the makespans of the both schedules hold the following inequality: $M(S') \leq M(S)$. 

In the first step we take a look at the senders. The sending from the master can not start before tasks are available on the master. We do not know the ordering of the senders in $S$ but we know the ordering in $S'$: all senders are ordered in non-decreasing order of their $c_i$-values. Let $i_0$ be the first task sent in $S$ where the sender of task $i_0$ has a bigger $c_i$-value than the sender of the $(i_0+1)$-th task. We then exchange the senders of task $i_0$ and task $(i_0+1)$ and call this new schedule $S_{\text{new}}$. Obviously the reception time for the second task is still the same. But as you can see in Figure 3(a), the time when the first task is available on the master has changed: after the exchange, the first task is available earlier and ditto ready for reception. Hence this exchange improves the availability on the master (and reduces possible idle times for the receivers). We use this mechanism to transform the sending order of $S$ in the sending order of $S'$ and at each time the availability on the master is improved. Hence at the end of the transformation the makespan of $S_{\text{new}}$ is smaller than or equal to that of $S$ and the sending order of $S_{\text{new}}$ and $S'$ is the same.

In the second step of the transformation we take care of the receivers (cf. Figures 3(b) and 3(c)). Having already changed the sending order of $S$ by the first transformation of $S$ into $S_{\text{new}}$, we start here directly by the transformation of $S_{\text{new}}$. Using the same mechanism as for the senders, we call $j_0$ the first task such that the receiver of task $j_0$ has a smaller $c_i$-value than the receiver of task $j_0+1$. We exchange the receivers of the tasks $j_0$ and $j_0+1$ and call the new schedule $S_{\text{new}}(1)$. $j_0$ is sent at the same time than previously, and the processor receiving it, receives it earlier than it received $j_{0+1}$ in $S_{\text{new}}$. $j_{0+1}$ is sent as soon as it is available on the master and as soon as the communication of task $j_0$ is completed. The first of these two conditions had also to be satisfied by $S_{\text{new}}$. If the second condition is delaying the beginning of the sending of the task $j_0+1$ from the master, then this communication ends at time $t_{\text{in}} + c_{\pi'(j_0)} + c_{\pi'(j_{0+1})} = t_{\text{in}} + c_{\pi(j_0+1)} + c_{\pi(j_0)}$ and this communication ends at the same time than under the schedule $S_{\text{new}}$ (here $\pi(j_0)$ ($\pi'(j_0)$) denotes the receiver of task $j_0$ in schedule $S_{\text{new}}$ ($S_{\text{new}}(1)$, respectively)). Hence the finish time of the communication of task $j_0+1$ in schedule $S_{\text{new}}(1)$ is less than or equal to the finish time in the previous schedule. In all cases, $M(S_{\text{new}}(1)) \leq M(S_{\text{new}})$. Note that this transformation does not change anything
for the tasks received after }_{j_{0+1}} \text{ except that we always perform the scheduled communications as soon as possible. Repeating the transformation for the rest of the schedule } S_{\text{new}} \text{ we reduce all idle times in the receptions as far as possible. We get for the makespan of each schedule } S_{\text{new}}(k): M(S_{\text{new}}(k)) \leq M(S_{\text{new}}) \leq M(S). \text{ As after these (finite number of) transformations the order of the receivers will be in non-decreasing order of the } c_i \text{-values, the receiver order of } S_{\text{new}}(\infty) \text{ is the same as the receiver order of } S' \text{ and hence we have } S_{\text{new}}(\infty) = S'. \text{ Finally we conclude that the makespan of } S' \text{ is smaller than or equal to any other schedule } S \text{ and hence } S' \text{ is optimal.} \square


In this section we present the Best-Balance Algorithm (BBA), an algorithm to schedule on homogeneous star platforms. We use a bus network with communication speed } c, \text{ but additionally we suppose that the computation powers are homogeneous as well. So we have } w_i = w \text{ for all } i, 1 \leq i \leq m.

The idea of BBA is simple: in each iteration, we look if we could finish earlier if we redistribute a task. If so, we schedule the task, if not, we stop redistributing. The algorithm has polynomial run-time. It is a natural intuition that BBA is optimal on homogeneous platforms, but the formal proof is rather complicated.

4.1. Notations used in BBA

BBA schedules one task per iteration } i. \text{ Let } L_k^{(i)} \text{ denote the number of tasks of worker } k \text{ after iteration } i, \text{ i.e., after } i \text{ tasks were redistributed. The date at which the master has finished receiving the } i\text{-th task is denoted by } m_{\text{in}}^{(i)}. \text{ In the same way we call } m_{\text{out}}^{(i)} \text{ the date at which the master has finished sending the } i\text{-th task. Let } end_k^{(i)} \text{ be the date at which worker } k \text{ would finish processing the load it would hold if exactly } i \text{ tasks are redistributed. The worker } k \text{ in iteration } i \text{ with the biggest finish time } end_k^{(i)}, \text{ who is chosen to send one task in the next iteration, is called } \text{sender. We call } receiver \text{ the worker } k \text{ with smallest finish time } end_k^{(i)} \text{ in iteration } i \text{ who is chosen to receive one task in the next iteration.}

In iteration } i = 0 \text{ we are in the initial configuration: All workers own their initial tasks } L_k^{(0)} = L_k \text{ and the makespan of each worker } k \text{ is the time it needs to compute all its tasks: } end_k^{(0)} = L_k^{(0)} \times w. \text{ m}_{\text{in}}^{(0)} = m_{\text{out}}^{(0)} = 0.

4.2. The Best Balance Algorithm - BBA

We first sketch BBA: In each iteration } i \text{ do: Compute the time } end_k^{(i-1)} \text{ it would take worker } k \text{ to process } L_k^{(i-1)} \text{ tasks. A worker with the biggest finish time } end_k^{(i-1)} \text{ is arbitrarily chosen as } \text{sender, he is called } sender. \text{ Compute the temporary finish times } end_k^{(i)} \text{ of each worker if it would receive from } sender \text{ the } i\text{-th task. A worker with the smallest temporary finish time } end_k^{(i)} \text{ will be the receiver, called } receiver. \text{ If there are multiple workers}
with the same temporary finish time \( \end{(i)} \), we take the worker with the smallest finish time \( \end{(i-1)} \). If the finish time of sender is strictly larger than the temporary finish time \( \end{sender} \) of sender, sender sends one task to receiver and iterate. Otherwise stop.

**Lemma 1.** On homogeneous star-platforms, in iteration \( i \) the Best-Balance Algorithm (Algorithm 1) always chooses as receiver a worker which finishes processing the first in iteration \( i - 1 \).

**Proof.** As the platform is homogeneous, all communications take the same time and all computations take the same time. In Algorithm 1 the master chooses as receiver in iteration \( i \) the worker \( k \) that would end the earliest the processing of the \( i \)-th task sent. To prove that worker \( k \) is also the worker which finishes processing in iteration \( i - 1 \) first, we have to consider two cases:

1. **Task \( i \) arrives when all workers are still working.**

   As all workers are still working when the master finishes to send task \( i \), the master chooses as receiver a worker which finishes processing the first, because this worker will also finish processing task \( i \) first, as we have homogeneous conditions.

2. **Task \( i \) arrives when some workers have finished working.**

   If some workers have finished working when the master can finish to send task \( i \), all these workers could start processing task \( i \) at the same time. Our algorithm chooses in this case a worker which finished processing first. \( \square \)

In the following lemma we will show that schedules, where sending workers also receive tasks, can be transformed in a schedule where this effect does not appear.

**Lemma 2.** On a platform with homogeneous communications, if there exists a schedule \( S \) with makespan \( M \), then there also exists a schedule \( S' \) with a makespan \( M' \leq M \) such that no worker both sends and receives tasks.

**Proof.** We will prove that we can transform a schedule where senders might receive tasks in a schedule with equal or smaller makespan where senders do not receive any tasks.

If the master receives its \( i \)-th task from processor \( P_j \) and sends it to processor \( P_k \), we say that \( P_k \) receives this task from processor \( P_j \).

Whatever the schedule, if a sender receives a task we have the situation of a sending chain: at some step of the schedule a sender \( s_i \) sends to a sender \( s_k \), while in another step of the schedule the sender \( s_k \) sends to a receiver \( r_j \). So the master is occupied twice. As all receivers receive in fact their tasks from the master, it does not make a difference for them which sender sent the task to the master. So we can break up the sending chain in the following way: We look for the earliest time, when a sending worker, \( s_k \), receives a task from a sender, \( s_i \). Let \( r_j \) be a receiver that receives a task from sender \( s_k \). There are two possible situations:

1. **Sender \( s_i \) sends to sender \( s_k \) and later sender \( s_k \) sends to receiver \( r_j \), see Figure 4(a).** This case is simple: As the communication from \( s_i \) to \( s_k \) takes place first and we have homogeneous communication links, we can replace this communication by an emission from sender \( s_i \) to receiver \( r_j \) and just delete the second communication.
2. Sender $s_k$ sends to receiver $r_j$ and later sender $s_i$ sends to sender $s_k$, see Figure 4(b). In this case the reception on receiver $r_j$ happens earlier than the emission of sender $s_i$, so we cannot use exactly the same mechanism as in the previous case. But we can use our hypothesis that sender $s_k$ is the first sender that receives a task. Therefore, sender $s_i$ did not receive any task until $s_k$ receives. So at the moment when $s_k$ sends to $r_j$, we know that sender $s_i$ already owns the task that it will send later to sender $s_k$. As we use homogeneous communications, we can schedule the communication $s_i \rightarrow r_j$ when the communication $s_k \rightarrow r_j$ originally took place and delete the sending from $s_i$ to $s_k$.

As in both cases we gain in communication time, but we keep the same computation time, we do not increase the makespan of the schedule, but we transformed it in a schedule with one less sending chain. By repeating this procedure for all sending chains, we transform the schedule $S$ in a schedule $S'$ without sending chains while not increasing the makespan.

Proposition 1. **Best-Balance Algorithm (Algorithm 1) calculates an optimal schedule $S$ on a homogeneous star network, where all tasks are initially located on the workers and communication capabilities as well as computation capabilities are homogeneous and all tasks have the same size.**

**Proof.** To prove that BBA is optimal, we take a schedule $S_{algo}$ calculated by Algorithm 1. Then we take an optimal schedule $S_{opt}$. (Because of Lemma 2 we can assume that in the schedule $S_{opt}$ no worker both sends and receives tasks.) We are going to transform by induction this optimal schedule into our schedule $S_{algo}$.

As we use a homogeneous platform, all workers have the same communication time $c$. Without loss of generality, we can assume that both algorithms do all communications as soon as possible. So we can divide our schedule $S_{algo}$ in $s_a$ steps and $S_{opt}$ in $s_o$ steps. A step corresponds to the emission of one task, and we number in this order the tasks sent. Accordingly the $s$-th task is the task sent during step $s$ and the actual schedule corresponds to the load distribution after the $s$ first tasks. We start our schedule at time $T = 0$.

Let $S(i)$ denote the worker receiving the $i$-th task under schedule $S$. Let $i_0$ be the first step where $S_{opt}$ differs from $S_{algo}$, i.e., $S_{algo}(i_0) \neq S_{opt}(i_0)$ and $\forall i < i_0, S_{algo}(i) = S_{opt}(i)$. We look for a step $j > i_0$, if it exists, such that $S_{opt}(j) = S_{algo}(i_0)$ and $j$ is minimal.

We are in the following situation: schedule $S_{opt}$ and schedule $S_{algo}$ are the same for all tasks $[1..(i_0 - 1)]$. As worker $S_{algo}(i_0)$ is chosen at step $i_0$, then, by definition

![](image)
Fig. 5. BEST-BALANCE ALGORITHM

\[ i \leftarrow 0; \ m_{\text{in}}^{(i)} \leftarrow 0; \ m_{\text{out}}^{(i)} \leftarrow 0; \]
\[ \forall k L_k^{(i)} \leftarrow L_k; \ \text{end}^{(i)} \leftarrow L_k^{(i)} \times w; \]
\[ \text{while true do} \]
\[ \quad \text{sender} \leftarrow \max_k \text{end}^{(i)}; \ m_{\text{in}}^{(i+1)} \leftarrow m_{\text{in}}^{(i)} + c; \]
\[ \quad \text{task_arr_worker} = \max(m_{\text{in}}^{(i+1)}, m_{\text{out}}^{(i)}) + c; \]
\[ \quad \text{foreach } k \text{ do} \]
\[ \quad \quad \text{end}^{(i)} \leftarrow \max_k \text{end}^{(i)} \times \text{task}_{\text{arr_worker}} + w \]
\[ \quad \quad \text{select receiver such that end}_{\text{receiver}}^{(i+1)} = \min_k \text{end}^{(i)} \]
\[ \quad \quad \text{and if there are several processors with the same minimum end}^{(i)} \]
\[ \quad \quad \text{choose one with smallest end}^{(i)}; \]
\[ \quad \quad \text{if end}_{\text{sender}}^{(i)} \leq end_{\text{receiver}}^{(i)} \text{ then} \]
\[ \quad \quad \quad \text{break;} \] /* we cannot improve the makespan */
\[ \quad \quad \text{else} \]
\[ \quad \quad \quad \text{/* we improve the makespan by sending the task to the receiver */} \]
\[ \quad \quad \quad m_{\text{out}}^{(i+1)} \leftarrow \text{task}_{\text{arr_worker}}; \]
\[ \quad \quad \quad \text{end}_{\text{sender}}^{(i+1)} \leftarrow \text{end}_{\text{sender}}^{(i)} - w; \]
\[ \quad \quad \quad L_{\text{sender}}^{(i+1)} \leftarrow L_{\text{sender}}^{(i)} - 1; \]
\[ \quad \quad \quad \text{end}_{\text{receiver}}^{(i+1)} \leftarrow \text{end}_{\text{receiver}}^{(i)}; \]
\[ \quad \quad \quad L_{\text{receiver}}^{(i+1)} \leftarrow L_{\text{receiver}}^{(i)} + 1; \]
\[ \quad \quad \quad \text{foreach } j \neq \text{receiver and } j \neq \text{sender do} \]
\[ \quad \quad \quad \quad \text{end}_{j}^{(i+1)} \leftarrow \text{end}_{j}^{(i)}; \]
\[ \quad \quad \quad \quad L_{j}^{(i+1)} \leftarrow L_{j}^{(i)}; \]
\[ \quad \quad \quad i \leftarrow i + 1 \]

of Algorithm 1, this means that this worker finishes first its processing after the reception of the \((n_0 - 1)\)-th tasks (cf. Lemma 1). As \(S_{\text{opt}}\) and \(S_{\text{algo}}\) differ in step \(i_0\), we know that \(S_{\text{opt}}\) chooses worker \(S_{\text{opt}}(i_0)\) that finishes the schedule of its load after step \((n_0 - 1)\) no sooner than worker \(S_{\text{algo}}(i_0)\).

Case 1: Let us first consider the case where there exists such a step \(j\) So \(S_{\text{algo}}(i_0) = S_{\text{opt}}(j)\) and \(j > i_0\). We know that worker \(S_{\text{opt}}(j)\) under schedule \(S_{\text{opt}}\) does not receive any task between step \(i_0\) and step \(j\) as \(j\) is chosen minimal.

We use the following notations for the schedule \(S_{\text{opt}}\), depicted on Figures 6, 7, and 8:

\(T_j\): the date at which the reception of task \(j\) is finished on worker \(S_{\text{opt}}(j)\), i.e., \(T_j = j \times c + c\) (the time it takes the master to receive the first task plus the time it takes him to send \(j\) tasks).

\(T_{i_0}\): the date at which the reception of task \(i_0\) is finished on worker \(S_{\text{opt}}(i_0)\), i.e., \(T_{i_0} = i_0 \times c + c\).

\(F_{\text{pred}(j)}\): time when computation of task \(\text{pred}(j)\) is finished, where task \(\text{pred}(j)\) denotes the last task which is computed on worker \(S_{\text{opt}}(j)\) before task \(j\) is computed.

\(F_{\text{pred}(i_0)}\): time when computation of task \(\text{pred}(i_0)\) is finished, where task \(\text{pred}(i_0)\) denotes the last task which is computed on worker \(S_{\text{opt}}(i_0)\) before task \(i_0\) is com-
We have to consider two sub-cases:

**Theorem 1.** Let\( T_j \leq T_{\text{pred}(i_0)} \) (Figure 6(a)).

This means that we are in the following situation: the reception of task \( j \) on worker \( S_{\text{opt}}(j) \) has already finished when worker \( S_{\text{opt}}(i_0) \) finishes the work it has been scheduled until step \( i_0 - 1 \).

In this case we exchange the tasks \( i_0 \) and \( j \) of schedule \( S_{\text{opt}} \) and we create the following schedule \( S'_{\text{opt}} \):

\[
S'_{\text{opt}}(i_0) = S_{\text{opt}}(j) = S_{\text{algo}}(i_0),
\]

\[
S'_{\text{opt}}(j) = S_{\text{opt}}(i_0)
\]

and \( \forall i \neq i_0, j \), \( S'_{\text{opt}}(i) = S_{\text{opt}}(i) \). The schedule of the other workers is kept unchanged. All tasks are executed at the same date than previously (but maybe not on the same processor).

![Fig. 6. Schedule \( S_{\text{opt}} \) before and after exchange of tasks \( i_0 \) and \( j \).](attachment:figure6.png)

Now we prove that this kind of exchange is possible.

We know that worker \( S_{\text{opt}}(j) \) is not scheduled any task later than step \( i_0 - 1 \) and before step \( j \), by definition of \( j \). So we know that this worker can start processing task \( j \) when task \( j \) has arrived and when it has finished processing its amount of work scheduled until step \( i_0 - 1 \). We already know that worker \( S_{\text{opt}}(j) = S_{\text{algo}}(i_0) \) finishes processing its tasks scheduled until step \( i_0 - 1 \) at a time earlier than or equal to that of worker \( S_{\text{opt}}(i_0) \) (cf. Lemma 1). As we are in homogeneous conditions, communications and processing of a task takes the same time on all processors. So we can exchange the destinations of steps \( i_0 \) and \( j \) and keep the same moments of execution, as both tasks will arrive in time to be processed on the other worker: task \( i_0 \) will arrive at worker \( S_{\text{opt}}(j) \) when it is still processing and the same for task \( j \) on worker \( S_{\text{opt}}(i_0) \). Hence task \( i_0 \) will be sent to worker \( S_{\text{opt}}(j) = S_{\text{algo}}(i_0) \) and worker \( S_{\text{opt}}(i_0) \) will receive task \( j \). So schedule \( S_{\text{opt}} \) and schedule \( S_{\text{algo}} \) are the same for all tasks \( \{1 \ldots i_0 \} \) now. As both tasks arrive in time and can be executed instead of the other task, we do not change anything in the makespan \( M \). And as \( S_{\text{opt}} \) is optimal, we keep the optimal makespan.

**Theorem 1.** Let\( T_j \geq T_{\text{pred}(i_0)} \) (Figure 7(a)).

In this case we have the following situation: task \( j \) arrives on worker \( S_{\text{opt}}(j) \), when worker \( S_{\text{opt}}(i_0) \) has already finished processing its tasks scheduled until step \( i_0 - 1 \).

In this case we exchange the schedule destinations \( i_0 \) and \( j \) of schedule \( S_{\text{opt}} \) beginning at tasks \( i_0 \) and \( j \) (see Figure 7). In other words we create a schedule \( S'_{\text{opt}} \).
∀i ≥ i₀ such that S⁻¹ j (i) = S⁻¹ j (i₀); S⁻¹ j (i) = S⁻¹ j (i₀)
∀i ≥ j such that S⁻¹ j (i) = S⁻¹ j (j); S⁻¹ j (i) = S⁻¹ j (i₀)
and ∀i ≤ i₀ S⁻¹ j (i) = S⁻¹ j (i). The schedule Sopt of the other workers is kept unchanged. We recompute the finish times F⁻¹ Sopt (j) of workers Sopt (j) and Sopt (i₀) for all steps s > i₀.

Now we prove that this kind of exchange is possible. First of all we know that worker Salgo (i₀) is the same as the worker chosen in step j under schedule Sopt and so Salgo (i₀) = Sopt (j). We also know that worker Sopt (j) is not scheduled any tasks later than step i₀ − 1 and before step j, by definition of j. Because of the choice of worker Salgo (i₀) = Sopt (j) in Salgo, we know that worker Sopt (j) has finished working when task j arrives: at step i₀ worker Sopt (j) finishes earlier than or at the same time as worker Sopt (i₀) (Lemma 1) and as we are in the case where T j ≥ F pred(i₀), Sopt (j) has also finished when j arrives. So we can exchange the destinations of the workers Sopt (i₀) and Sopt (j) in the schedule steps equal to, or later than, step i₀ and process them at the same time as we would do on the other worker. As we have shown that we can start processing task j on worker Sopt (i₀) at the same time as we did on worker Sopt (j), and the same for task i₀, we keep the same makespan. And as Sopt is optimal, we keep the optimal makespan.

**Case 2:** If there does not exist a j, i.e., we can not find a schedule step j > i₀ such that worker Salgo (i₀) is scheduled a task under schedule Sopt, so we know that no other task will be scheduled on worker Salgo (i₀) under the schedule Sopt. As our algorithm chooses in step s the worker that finishes task s + 1 the first, we know that worker Salgo (i₀) finishes at a time earlier or equal to that of Sopt. Worker Salgo (i₀) will be idle in the schedule Sopt for the rest of the algorithm, because otherwise we would have found a step j. As we are in homogeneous conditions, we can simply displace task i₀ from worker Sopt (i₀) to worker Salgo (i₀) (see Figure 8). As we have Sopt (i₀) ≠ Salgo (i₀) and with Lemma 1 we know that worker Salgo (i₀) finishes processing its tasks until step i₀ − 1 at a time earlier than or equal to Sopt (i₀), and we do not downgrade the execution time because we are in homogeneous conditions.

Once we have done the exchange of task i₀, the schedules Sopt and Salgo are the same for all tasks [1..i₀]. We restart the transformation until Sopt = Salgo for all tasks [1..min(sₐ, sᵢ₀)] scheduled by Salgo.
Now we will prove by contradiction that the number of tasks scheduled by $S_{algo}$, $s_a$, and $S_{opt}$, $s_o$, are the same. After min$(s_a, s_o)$ transformation steps $S_{opt} = S_{algo}$ for all tasks $[1..\text{min}(s_a, s_o)]$ scheduled by $S_{algo}$. So if after these steps $S_{opt} = S_{algo}$ for all $n$ tasks, both algorithms redistributed the same number of tasks and we have finished.

We now consider the case $s_a \neq s_o$. In the case of $s_a > s_o$, $S_{algo}$ schedules more tasks than $S_{opt}$. At each step of our algorithm we do not increase the makespan. So if we do more steps than $S_{opt}$, this means that we scheduled some tasks without changing the global makespan. Hence $S_{algo}$ is optimal.

If $s_a < s_o$, this means that $S_{opt}$ schedules more tasks than $S_{algo}$ does. In this case, after $s_a$ transformation steps, $S_{opt}$ still schedules tasks. If we take a look at the schedule of the $(s_a + 1)$-th task in $S_{opt}$: regardless which receiver $S_{opt}$ chooses, it will increase the makespan as we prove now. In the following we will call $s_{algo}$ the worker our algorithm would have chosen to be the sender, $r_{algo}$ the worker our algorithm would have chosen to be the receiver. $s_{opt}$ and $r_{opt}$ are the sender and receiver chosen by the optimal schedule. Indeed, in our algorithm we would have chosen $s_{algo}$ as sender such that it is a worker which finishes last. So the time worker $s_{algo}$ finishes processing is $F_{s_{algo}} = M(S_{algo})$. $S_{algo}$ chooses the receiver $r_{algo}$ such that it finishes processing the received task the earliest of all possible receivers and such that it also finishes processing the receiving task at the same time or earlier than the sender would do. As $S_{algo}$ did not decide to send the $(s_a + 1)$-th task, this means, that it could not find a receiver which fitted. Hence we know, regardless which receiver $S_{opt}$ chooses, that the makespan will strictly increase (as $S_{algo} = S_{opt}$ for all $[1..s_a]$). We take a look at the makespan of $S_{algo}$ if we would have scheduled the $(s_a + 1)$-th task. We know that we can not decrease the makespan anymore, because in our algorithm we decided to keep the schedule unchanged. So after the emission of the $(s_a + 1)$-th task, the makespan would become $M(S_{algo}) = F_{r_{algo}} \geq F_{s_{algo}}$. And $F_{r_{algo}} \leq F_{r_{opt}}$, because of the definition of receiver $r_{algo}$. As $M(S_{opt}) \geq F_{r_{opt}}$, we have $M(S_{algo}) \leq M(S_{opt})$. But we decided not to do this schedule as $M(S_{algo})$ is smaller before the schedule of the $(s_a + 1)$-th task than afterwards. Hence we get that $M(S_{algo}) < M(S_{opt})$. So the only possibility why $S_{opt}$ sends the $(s_a + 1)$-th task and still be optimal is that, later on, $r_{opt}$ sends a task to some other processor $r_k$. (Note that even if we choose $S_{opt}$ to have no such chains in the beginning, some might have appeared because of our previous transformations). In the same manner as we transformed sending chains in Lemma 2, we can suppress this sending chain,
by sending task \((s_a + 1)\) directly to \(r_k\) instead of sending to \(r_{opt}\). With the same argumentation, we do this by induction for all tasks \(k\), \((s_a + 1) \leq k \leq s_o\), until schedule \(S_{opt}\) and \(S_{algo}\) have the same number \(s_o = s_a\) and so \(S_{opt} = S_{algo}\) and hence \(M(S_{opt}) = M(S_{algo})\). \(\square\)

**Complexity:** The initialization phase is in \(O(m)\), as we have to compute the finish times for each worker. The while loop can be run at maximum \(n\) times, as we can not redistribute more than the \(n\) tasks of the system. Each iteration is in the order of \(O(m)\), which leads us to a total run time of \(O(m \times n)\).

5. **Scheduling on platforms with homogeneous communication links and heterogeneous computation capacities**

In this section we present an algorithm for star-platforms with homogeneous communications and heterogeneous workers, the **Moore Based Algorithm (MBA)**. As the name says, this algorithm is based on **Moore’s algorithms** [6], [7], whose aim is to maximize the number of tasks to be processed in-time, i.e., before tasks exceed their deadlines. Moore’s algorithm gives a solution to the \(1\|\sum U_j\) problem when the maximum number, among \(n\) tasks, has to be processed in time on a single machine. Each task \(k\), \(1 \leq k \leq n\), has a processing time \(w_k\) and a deadline \(d_k\), before which it has to be processed.

For a given makespan, we compute if there exists a possible schedule to finish all work in time. If there is one, we optimize the makespan by a binary search.

5.1. **Framework and notations for MBA**

We keep the star network of Section with homogeneous communication links. In contrast to Section we suppose \(m\) heterogeneous workers who own initially a number \(L_i\) of identical independent tasks.

Let \(M\) denote the objective makespan for the searched schedule \(\sigma\) and \(f_i\) the time needed by worker \(i\) to process its initial load. During the algorithm execution we divide all workers in two subsets, where \(S\) is the set of senders (\(s_i \in S\) if \(f_i > M\)) and \(R\) the set of receivers (\(r_i \in R\) if \(f_i < M\)). As our algorithm is based on Moore’s, we need a notation for deadlines. Let \(d_{r_i}^k\) be the deadline to receive the \(k\)-th task on receiver \(r_i\). \(l_s\) denotes the number of tasks sender \(i\) sends to the master and \(l_r\) stores the number of tasks receiver \(i\) is able to receive from the master. With help of these values we can determine the total amount of tasks that must be sent as \(L_{send} = \sum s_i \cdot l_s\). The total amount of tasks if all receivers receive the maximum amount of tasks they are able to receive is \(L_{recv} = \sum r_i \cdot l_r\). Finally, let \(L_{sched}\) be the maximal amount of tasks that can be scheduled by the algorithm.

5.2. **Moore based algorithm - MBA**

**Principle of the algorithm:** Considering the given makespan we determine overcharged workers, which can not finish all their tasks within this makespan. These overcharged workers will then send some tasks to undercharged workers, such that all of them can finish processing within the makespan. The algorithm solves the following two questions: Is there a possible schedule such that all workers
Parallel Processing Letters

Fig. 9. Moore Based Algorithm

initialize $f_i$ for all workers $i$, $f_i = L_i \times w_i$;
compute $R$ and $S$, order $S$ by non-decreasing values $c_i$ such that $c_{i_1} \leq c_{i_2} \leq \ldots$;

for each $s_i \in S$ do

$\begin{aligned}
& l_{s_i} \leftarrow \left\lceil \frac{f_{s_i} - T}{w_{s_i}} \right\rceil; \\
& \text{if } \left\lfloor \frac{T}{c_{s_i}} \right\rfloor < l_{s_i}, \text{ then} \\
& \quad \text{return } (false, \emptyset); /* M too small */
\end{aligned}$


\[ \text{total number of tasks to send: } L_{send} \leftarrow \sum s_i l_{s_i}; \]
\[ \text{D } \leftarrow \emptyset; \]

foreach $r_i \in R$ do

\[
\begin{aligned}
& l_{r_i} \leftarrow 0; \\
& \text{while } f_{r_i} \leq M - (l_{r_i} + 1) \times w_{r_i} \text{ do} \\
& \quad l_{r_i} \leftarrow l_{r_i} + 1; d^{(l_{r_i})}_{(r_i)} \leftarrow M - (l_{r_i} \times w_{r_i}); \\
& \quad D \leftarrow D \cup (d^{(l_{r_i})}_{(r_i)}, r_i);
\end{aligned}
\]

\[ \text{# of tasks that can be received: } L_{recv} \leftarrow \sum r_i l_{r_i}; \]

senders send in non-decreasing order of values $c_{s_i}$;

order deadline-list $D$ by non-decreasing values of deadlines $d_{r_i}$ and rename the deadlines in this order from 1 to $L_{recv};$

\[ \sigma \leftarrow \emptyset; t \leftarrow c_{s_1}; L_{sched} = 0; \]

for $i = 1$ to $L_{recv}$ do

\[ \begin{aligned}
& \quad (d_i, r_i) \leftarrow i-th \text{ element } (d_{(r_j)}, r_k) \text{ of } D; \\
& \quad \sigma \leftarrow \sigma \cup \{r_i\}; t \leftarrow t + c_{r_i}; L_{sched} \leftarrow L_{sched} + 1; \\
& \text{if } t > d_i, \text{ then} \\
& \quad \text{Find } (d_j, r_j) \text{ in } \sigma \text{ s.t. } c_{r_j} \text{ value is largest;} \\
& \quad \sigma \leftarrow \sigma \setminus \{(d_j, r_j)\}; t \leftarrow t - c_{r_j}; L_{sched} \leftarrow L_{sched} - 1;
\end{aligned} \]

\[ \text{return } ((L_{sched} \geq L_{send}), \sigma); \]

can finish in the given makespan? In which order do we have to send and receive to obtain such a schedule?

The algorithm can be divided into four phases:

**Phase 1** decides which of the workers will be senders and which receivers, depending of the given makespan (see Figure 10). Senders are workers which are not able to process all their initial tasks in time, whereas receivers are workers which could treat more tasks in the given makespan $M$ than they hold initially.

**Phase 2** fixes how many transfers have to be scheduled from each sender such that the senders all finish their remaining tasks in time. Sender $s_i$ will have to send an amount of tasks $l_{s_i} = \left\lceil \frac{f_{s_i} - T}{w_{s_i}} \right\rceil$ (i.e., the number of light colored tasks of a sender in Figure 10).

**Phase 3** computes for each receiver the deadline of each of the tasks it can
receive, i.e., a pair \((d^{(i)}_{r_j}, r_j)\) that denotes the \(i\)-th deadline of receiver \(r_j\). See Figure 11 for an example.

**Phase 4** is the proper scheduling step: The master decides which tasks have to be scheduled on which receivers and in which order. If the schedule is able to send at least \(L_{send}\) tasks the algorithm succeeds, otherwise it fails.

Algorithm 2 describes MBA in pseudo-code. Note that the algorithm is written for heterogeneous conditions, but here we study it for homogeneous communication links.

**Theorem 3.** MBA (Algorithm 2) succeeds to build a schedule \(\sigma\) for a given makespan \(M\), if and only if there exists a schedule with makespan less than or equal to \(M\), when the platform is made of one master, several workers with heterogeneous computation power but homogeneous communication capabilities.

Moore’s Algorithm constructs a maximal set \(\sigma\) of early jobs on a single machine scheduling problem. In [5] we show that our algorithm can be reduced to this problem.

**Proposition 2.** Performing a binary search with precision \(\frac{1}{\lambda}\), where \(\lambda = \text{lcm}\{\beta_i, \delta_i\}, 1 \leq i \leq m\), on Algorithm 2 returns in polynomial time an optimal schedule \(\sigma\) for the following scheduling problem: minimizing the makespan on a star-platform with homogeneous communication links and heterogeneous workers where the initial tasks are located on the workers.

For the proof we refer to the companion research report [5].

### 6. Related work

To the best of our knowledge, there are no papers dealing with the same type of data redistribution algorithms which can be overlapped by computations (provided that enough data is available locally).

However, REDISTRIBUTION ALGORITHMS have been well studied in the literature. Unfortunately already simple redistribution problems are NP complete [8]. For this reason, optimal algorithms can be designed only for particular cases, as it
is done in [9]. In their research, the authors restrict the platform architecture to ring topologies, both uni-directional and bidirectional. In the homogeneous case, they were able to prove optimality, but the heterogenous case is still an open problem. In spite of this, other efficient algorithms have been proposed. For topologies like trees or hypercubes some results are presented in [10].

The load balancing problem is not directly dealt with in this paper. Anyway we want to quote some key references to this subject, as the results of these algorithms are the starting point for the redistribution process. Generally load balancing techniques can be classified into two categories. Dynamic load balancing strategies and static load balancing. Dynamic techniques might use the past for the prediction of the future as it is the case in [11] or they suppose that the load varies permanently [12]. That is why for our problem static algorithms are more interesting: we are only treating star-platforms and as the amount of load to be treated is known \textit{a priori} we do not need prediction. For homogeneous platforms, the papers in [13] survey existing results. Heterogeneous solutions are presented in [14] or [15]. This last paper is about a dynamic load balancing method for data parallel applications, called the \textit{working-manager method}: the manager is supposed to use its idle time to process data itself. So the heuristic is simple: when the manager does not perform any control task it has to work, otherwise it schedules.

7. Conclusion

We have dealt with the problem of scheduling and redistributing independent and identical tasks on heterogeneous master-slave platforms. We have proved the NP completeness (in the strong sense) of the problem for fully heterogeneous platforms. We have also proved that this problem is polynomial when computations are negligible, which shows the additional complexity induced by the overlap between communications and computations in the general case. Also, we were able to present optimal polynomial algorithms for special important topologies: a simple greedy algorithm for homogeneous star-networks, and a more complicated algorithm for platforms with homogeneous communication links and heterogeneous workers. The proof of optimality for both algorithms turned out rather complicated. On the more practical side, examples and simulations to compare the performance of the different algorithms are available in [5].

A natural extension of this work would be to derive approximation algorithms, i.e., heuristics whose worst-case is guaranteed within a certain factor to the optimal, for the fully heterogeneous case. However, it is often the case in scheduling problems for heterogeneous platforms that approximation ratios contain the quotient of the largest platform parameter by the smallest one, thereby leading to very pessimistic results in practical situations.

More generally, much work remains to be done along the same lines of load-balancing and redistributing while computation goes on. We can envision dynamic master-slave platforms whose characteristics vary over time, or even where new resources are enrolled temporarily in the execution. We can also deal with more complex interconnection networks, allowing slaves to circumvent the master and exchange data directly.
Instructions for Typesetting Camera-Ready Manuscripts

[3] Einstein@Home. \url{http://einstein.phys.usm.edu}.