A Distributed Procedure for Bandwidth-Centric Scheduling of Independent-Task Applications

Cyril Banino
Department of Computer and Information Science (IDI)
Norwegian University of Science and Technology (NTNU)
Sem Sælandsvei 7-9 NO-7491 Trondheim, NORWAY
Cyril.Banino@idi.ntnu.no

Abstract

The problem of scheduling independent tasks on heterogeneous trees is considered. The nodes of the tree may have different processing times, and links different communication times. The single-port, full overlap model is used for modeling the activities of the nodes. A distributed method for determining the maximum steady-state throughput of a tree is presented. Then, we show how each node can build up its own local schedule independently of the rest of the platform. In addition, the final schedule is asynchronous and event-driven, meaning that each node (except the root) acts without any time-related information. A local scheduling strategy which aims at minimizing the amount of tasks buffered at node locations during steady-state is introduced. As a consequence, the lengths of the start-up and wind-down phases are considerably reduced.

1. Introduction

A recent trend in high performance computing is to deploy computing platforms that span over large networks in order to harness geographically distributed computing resources. The aim is often to provide computing power to applications at unprecedented scale. Good candidates for such environments are Master-Worker applications, which are composed of a large number of computational tasks independent from each other, i.e. where no inter-task communications take place, and where the tasks can be computed in any order. Many applications have been and can be implemented under the Master-Worker paradigm. They include: The processing of large measurement data sets like the SETI@home project [1], biological sequence comparisons [14], or also distributed problems organized by companies like Entropia [9]. See [12] for more examples.

This paper is a follow on of recent work by Beaumont et al. [5] as well as Kreececk et al. [12], who also considered the problem of scheduling Master-Worker applications onto heterogeneous tree-shaped computing platforms. The resources composing the platform may have different computation and communication speeds, as well as different overlap capabilities. The platform topology is modeled by a tree, where each node represents some computing resource capable of computing and/or communicating with its neighbors via message passing over interconnection links. The main advantage of using trees, as opposed to the more general graphs is that no choices need to be made about how to route the data [4]. The estimation of the different bandwidths of the platform links can be obtained using tools such as the Network Weather Service [15], or by local measurements. The application tasks are modeled as requiring some input data file, and producing some output data file. A special processor, called the master, generates the input files associated to the application tasks, decides which tasks to execute, and how many tasks to delegate to each of its children. In turn, each child decides which tasks to execute, and how many tasks to forward to its own children. In this paper, applications such as SETI@home, i.e. where the output files produced by the tasks are much smaller than the input files, are considered. Consequently, the return of the output files to the master node is negligible. Throughout the rest of the paper, we assimilate input files to tasks, and consider that tasks can be communicated and computed.

In this paper, we present a lightweight distributed communication procedure which allows each node to build up its local schedule autonomously in order to attain the maximum steady-state throughput of the tree, i.e. that maximizes the number of tasks computed per time unit. Our procedure is an efficient, practical and scalable implementation of the theoretical results presented in [5]. The rest of the paper is organized as follows: Related work is reviewed in Section 2. In Section 3 we formally state our model of computation and communication. Section 4 reassesses the bandwidth-centric principle that lays the foundations of our work. A distributed method for determining the optimal steady-state

0-7695-2312-9/05/$20.00 (c) 2005 IEEE
throughput of a tree is given in Section 5. We show in Section 6 how each node can build up its local schedule autonomously. An efficient start-up strategy is given in Section 7. Our main results are illustrated with an example in Section 8. Future work is discussed in Section 9, and our contributions are summarized in Section 10.

2. Related Work

“The traditional objective of scheduling algorithms is makespan minimization: Given the application tasks and a set of resources, find a mapping of the tasks onto the set of processors and order the execution of the tasks so that (i) resource constraints are satisfied, and (ii) a minimum schedule length is provided” [6]. Recent studies have been conducted on makespan minimization under heterogeneous conditions. Beaumont et al. [7] revisited the Master-Worker paradigm with heterogeneous processors interconnected via a bus. Dutot extended this result to daisy-chains as well as “spider graphs” [10], and showed that the problem was NP-hard for heterogeneous trees [11]. We believe that the scheduling strategy presented in this paper is a good heuristic candidate to solve the problem studied by Dutot, since we are able to obtain the optimal platform throughput using quick start-up and wind-down phases.

“An idea to circumvent the difficulty of makespan minimization is to lower the ambition of the scheduling objective” [6]. The problem becomes to maximize the steady-state throughput of the platform, i.e. the number of tasks computed per time unit.

Shao et al. [13] considered general interconnection platform graphs, and solve the Master-Worker tasking problem in steady-state using a network-flow approach. The authors model the platform nodes using the *multiple-port, full overlap* model [4], where the number of simultaneous communications for a given node is not bounded. Banino et al. [2] showed how to solve this problem for general interconnection graphs whose nodes operate under the *single-port, full-overlap* model using a linear programming approach.

Heterogeneous trees were considered by Kreaseck et al. [12] who presented two *autonomous* bandwidth-centric scheduling protocols that address the practical problem of attaining the maximum steady-state rate after some start-up and maintaining that rate until wind-down. Two communication models are studied, the non-interruptible communication (which corresponds to our model), and the interruptible communication where a request from a higher priority child may interrupt a communication to a lower priority child. Although the work of Kreaseck et al. is a first step towards a practical implementation of bandwidth-centric scheduling algorithms, under the non-interruptible communication model, their autonomous protocol might take non-optimal decisions, generating hence long start-up phases as well as unnecessary large numbers of tasks buffered at node locations.

3. Our Steady-State Model

Our model builds on the model proposed by Beaumont et al. in [5] that we augment by introducing the task computing and task communicating rates of the processors.

The target architectural/application framework is represented by a node-weighted edge-weighted tree $T = (V, E, w, c)$ as depicted in Figure 1. Each node $P_i \in V$ represents a computing resource of weight $w_i$, meaning that node $P_i$ requires $w_i$ units of time to process one task. Each edge $e_{i,j} : P_i \rightarrow P_j$ corresponds to a communicating resource and is weighted by a value $c_{i,j}$ which represents the time needed by a parent node $P_i$ to communicate one task to its child $P_j$.

![Figure 1. A tree labeled with node (computation) and edge (communication) weights.](image)

All the $w_i$ are assumed to be positive rational numbers since they represent the nodes processing times. We disallow $w_i = 0$ since it would permit node $P_i$ to perform an infinite number of tasks, but we allow $w_i = +\infty$; then, $P_i$ has no computing power, but can still forward tasks to other processors (e.g. to model a switch). Similarly, we assume that all $c_{i,j}$ are positive rational numbers since they correspond to the communication times between two processors.

We introduce the computing rate $r_i = \frac{1}{w_i}$ which is the number of tasks that processor $P_i$ can process per time unit, and the communicating rate $b_{i,j} = \frac{1}{c_{i,j}}$ which is the number of tasks that processor $P_i$ can send to child $P_j$ per time unit. We let $\frac{1}{c_{i,j}} = 0$, and $C_i$ denotes the index set of the children of node $P_i$.

There are several scenarios for the operation mode of the processors, and we rely to the classification proposed in [4]. A processor can do three kinds of activity: (i) it can perform some computation, (ii) it can receive tasks from its parent, and (iii) it can send tasks to its children. The degree of simultaneity between these three activities indicates the level of performance of a processor and its networking device.
It is important to point out that different processors of the platform may operate under different modes. In this paper, we concentrate on the full overlap, single-port model [2,5], since it has been shown in [4] that the other models reduce to it. In the full overlap, single-port model, a processor can simultaneously receive tasks from its parent, perform some (independent) computation, and send tasks to one of its children. At any given time-step, a given processor may open only two connections, one in emission and one in reception. We state the communication model more precisely: If \( P_i \) sends one task to \( P_j \) at time-step \( t \), then:

- \( P_j \) cannot start executing or sending this task before time-step \( t + c_{i,j} \),
- \( P_j \) cannot initiate a new receive operation before time-step \( t + c_{i,j} \) (but, it can perform a send operation and independent computation),
- \( P_i \) cannot initiate another send operation before time-step \( t + c_{i,j} \) (but, it can perform a receive operation and independent computation).

### 4. The Bandwidth-Centric Principle

An iterative method that determines the maximum steady-state rate of a heterogeneous tree has been presented by Beaumont et al. in [5]. Interestingly, it turns out that this strategy is bandwidth-centric: If enough bandwidth is available to the node, then all the children are kept busy. However, if bandwidth is limited, then tasks should be allocated only to the children which have sufficiently fast communication times regardless of their computing speeds. Nevertheless, the computing speeds of the children determine the frequency at which children will request tasks to their parent. So a faster processor will request tasks more often than a slower one. What the bandwidth-centric principle says is that if two children are in concurrence for obtaining a task, priority should be given to the child with fastest communication time, as this will optimize the communication resource of the parent.

Formally, let us recall the Proposition presented in [5] to solve the case for fork graphs. A fork graph as shown in Figure 2 consists of a node \( P_0 \) and its \( k \) children \( P_1, \ldots, P_k \).\( P_0 \) needs \( c_1 \) units of time to communicate a task to child \( P_i \). Concurrently, \( P_0 \) can receive tasks from its own parent \( P_{-1} \), requiring \( c_{-1} \) time units per task.

**Proposition 1 ([5])** With the above notations, the minimal value of \( w_f \) for the fork graph is obtained as follows:

1. Sort the children by increasing communication times.
   Re-number them so that
   \[ c_1 \leq c_2 \leq \ldots \leq c_k. \]

![Figure 2. A fork graph and the reduced node of equivalent computing power.](image)

2. Let \( p \) be the largest index so that
   \[ \sum_{j=1}^{p} \frac{c_{i,j}}{w} \leq 1. \]
   If \( p < k \) let \( \varepsilon = 1 - \sum_{i=1}^{p} \frac{c_{i,j}}{w} \), otherwise let \( \varepsilon = 0 \).

3. Then \( w_f = \max \left\{ \frac{1}{\varepsilon \cdot \sum_{i=0}^{p} \frac{1}{w} + \frac{1}{w_{p+1}}} \right\} \)

Based on the bandwidth-centric principle, Beaumont et al. conceived a bottom-up method that iteratively determines the steady-state throughput of the tree: At each time-step, the leaves of the tree are reduced together with their parent into a single node of equivalent computing power determined by Proposition 1 (see Figure 2). The procedure ends when there remains a single node having a computing power equivalent to the entire tree. However, Beaumont et al. did not specify how to achieve this maximum steady-state throughput in practice.

### 5. Reversing the Tree Traversal

Although the bottom-up procedure based on the bandwidth-centric principle provides the optimal throughput of the tree, a large number of unnecessary operations are done for strongly bandwidth limited platforms (i.e. when there is a bottleneck somewhere high up in the hierarchy, causing that many nodes of the platform can not be fed with tasks). Indeed, in such cases, many fork graph reductions are performed unnecessarily since only few nodes of the platform will be actually used during the computation. Therefore, we propose to perform a depth-first traversal of the tree (according to the bandwidth-centric principle), allowing hence to visit only the nodes that will be used in the final schedule.

Beaumont et al. show in [5] that we can solve the problem within the time unit interval, and then build up a schedule where an integer number of tasks are sent and processed. Our procedure is based on this result and involves transactions between the nodes composing the platform.
Definition 1 A transaction is defined as a two-phase protocol between a parent node \( P_p \) and a child node \( P_c \). The first phase of the transaction consists in \( P_p \) sending a message to \( P_c \) containing a single number \( \beta \) that represents the number of tasks that \( P_p \) can supply to \( P_c \) per time unit. We term the first phase a proposal from \( P_p \) to \( P_c \). The second phase of the transaction consists in \( P_c \) sending a message back to its parent \( P_p \) containing a single number \( \theta \) that represents the number of tasks that \( P_p \) could not handle. We term the second phase an acknowledgment from \( P_c \) to \( P_p \). Hence when the transaction is closed, \( P_p \) knows that its child \( P_c \) can consume \((\beta - \theta)\) tasks per time unit.

We use the following notations: \( P_x \overset{\sigma}{\rightarrow} P_y \) indicates that node \( P_x \) sends a number \( \sigma \) to node \( P_y \), and \( P_x \overset{\sigma}{\leftarrow} P_y \) indicates that node \( P_x \) receives a number \( \sigma \) from \( P_y \).

Our procedure works as follows: The node \( P_0 \) currently visited during the traversal of the tree will receive a proposal from its parent \( P_{-1} \). \( P_0 \) will then evaluate how many tasks it can process per time unit, and if there are some tasks left, try to propose them to its children. \( P_0 \) will then deal with its children one by one according to the bandwidth-centric principle, i.e. starting to deal with children that have the fastest communication times. \( P_0 \) will open a transaction with its first child \( P_1 \) by proposing the maximum number of tasks that it can supply to \( P_1 \) per time unit. \( P_1 \) in turn faces the same situation than \( P_0 \), and will keep a maximum of tasks for itself, and if there are some tasks left, try to delegate them to its children by negotiating new transactions. Hence, proposals will propagate down the tree, until either we reach a leave of the tree, either all the tasks have been allocated, or the current node has fully utilized its bandwidth and can not forward tasks further down the tree. Then the node \( P_0 \) at which the proposal propagation stopped, will acknowledge its parent \( P_{-1} \) with the amount of tasks that it could not process, and the transaction between \( P_{-1} \) and \( P_c \) is closed. \( P_0 \) will then take into consideration its transaction with \( P_c \) by reserving enough bandwidth to honor the transaction, and if it has some bandwidth left, as well as more tasks to delegate, will open a new transaction with another child. Hence, proposals travel down the tree opening transactions while acknowledgments travel up the tree closing the transactions in a recursive fashion.

Formally, let \( P_0 \) be the node currently visited during the tree traversal. \( P_{-1} \) be the parent of \( P_0 \), and \( P_1, P_2, \ldots, P_k \) be the \( k \) children of \( P_0 \) with communication times \( c_1, c_2, \ldots, c_k \) respectively. Further, let \( \delta_0 \) be the number of virtual tasks owned by \( P_0 \), \( \tau_0 \) be the bandwidth time of \( P_0 \) (to send tasks to its children), and \( \alpha_0 \) be the number of tasks computed by \( P_0 \) per time unit. At the beginning of the procedure we have \( \delta_0 = 0 \), \( \tau_0 = 1 \) since the time unit interval is considered, and \( \alpha_0 = 0 \).

Proposition 2 With the above notations, the optimal throughput of a tree \( T_h \) of height \( h \), is obtained via applying the BW-First() procedure.

Algorithm 1: BW-First\((P_0)\)

```
begin
  \( \delta_0 := 0 \), \( \tau_0 := 1 \), \( \alpha_0 := 0 \);
  \( P_0 \overset{\lambda}{\rightarrow} P_{-1} \);
  \( \alpha_0 := \min\{\tau_0, \lambda\} \);
  \( \delta_0 := \lambda - \alpha_0 \);
  foreach child \( P_i \) taken according to the bandwidth-centric principle do
    if \( \delta_0 = 0 \) or \( \tau_0 = 0 \) then
      goto instruction 14;
      \( \beta_i := \min\{\delta_0, \tau_0 \times b_i\} \);
      \( P_0 \overset{\beta_i}{\rightarrow} P_i \);
      \( P_0 \overset{\theta}{\leftarrow} P_i \);
      \( \delta_0 := \delta_0 - (\beta_i - \theta_i) \times c_i \);
      \( \tau_0 := \tau_0 - (\beta_i - \theta_i) \times c_i \);
      \( P_0 \overset{\delta_0}{\rightarrow} P_{-1} \);
  end
end
```

Proof In order to apply procedure BW-First() on the root \( P_{\text{root}} \) of the tree, we merely create a link connecting the root to a virtual parent \( P_v \) with no computing power. Then, the maximum number of task \( t_{\text{max}} \) that the tree rooted in \( P_{\text{root}} \) can execute per time unit is evaluated. Under the single-port, full overlap model, we have \( t_{\text{max}} = t_{\text{root}} + \max\{b_i \mid i \in C_{\text{root}}\} \). We then make \( P_0 \) propose \( t_{\text{max}} \) tasks to \( P_{\text{root}} \), and call procedure BW-First() on \( P_{\text{root}} \). At the end of the procedure, \( P_v \) will receive an acknowledgments of \( \theta \) tasks from \( P_{\text{root}} \), and the optimal throughput of the tree is equal to the quantity \((t_{\text{max}} - \theta)\).

The proof is done by recurrence over \( h \), the height of the tree. For \( h = 1 \), the tree is actually a fork graph. We will prove that for fork graphs, the BW-First() procedure is equivalent to Proposition 1. Let \( r_f \) be the computing rate of the fork graph rooted in \( P_0 \). Let us show that \( r_f = \sum_{i=0}^{p} r_i + \varepsilon \times b_{p+1} \), where \( p \) and \( \varepsilon \) are defined in Proposition 1.

First \( P_0 \) receives a proposal of \( \lambda \) tasks from its parent \( P_{-1} \), and keeps as many tasks as possible for its own computation. Then, \( P_0 \) will propose the remaining tasks to its children according to the bandwidth-centric principle. For each child \( P_i, P_0 \) must determine whether all the remaining tasks can be communicated or not. Then a proposal is made to child \( P_i \). It is now the turn of \( P_i \) to execute the BW-First() procedure. \( P_i \) receives a proposal of \( \theta_i \) tasks from its parent \( P_0 \), keeps as many tasks as possible for its own computation and since \( P_i \) does not have any children, sends back to \( P_0 \).
the number of tasks that it could not execute. At that point, either \( P_i \) is fully utilized (\( \alpha_i = r_i \)) or not (\( \alpha_i < r_i \)). In the first case scenario, \( P_0 \) will proceed to its next child, with previously adjusting \( \delta_0 \) and \( \tau_0 \), considering that \( c_i \times r_i \) time units will be necessary to furnish \( r_i \) tasks to \( P_i \). \( P_0 \) will then establish new transactions with its children until the second case scenario takes place (i.e. a child \( P_{q+1} \) is not fully utilized). In this case, either all the virtual tasks owned by \( P_0 \) have been processed, or \( P_0 \) utilized all its bandwidth time and can not send as many tasks as \( P_{q+1} \) can consume. In the first case, the limiting factor is the number of tasks \( \lambda \) received from \( P_{-1} \). In the second case, the limiting factor is the bandwidth of \( P_0 \): \( P_0 \) fed fully its \( q \) first children with tasks, i.e. \( \forall i \leq q, \alpha_i = r_i \). \( P_0 \) will hence spend \( \sum_{i=1}^{q} c_i r_i \) time units to communicate with its \( q \) first children, and only \( c_{q+1} \alpha_{q+1} \) time units with \( P_{q+1} \). Since the bandwidth of \( P_0 \) is saturated, we have \( \sum_{i=1}^{q} c_i r_i + \alpha_{q+1} r_{q+1} = 1 \). Since \( P_{q+1} \) could consume all the tasks proposed by \( P_0 \) without being fully utilized, we have \( \alpha_{q+1} < r_{q+1} \) which by scaling both sides with \( c_{q+1} \) gives \( \alpha_{q+1} c_{q+1} < r_{q+1} c_{q+1} \). Consequently, we have \( \sum_{i=1}^{q} c_i r_i > 1 \), which gives \( q = \sum_{i=1}^{q} c_i r_i \). If \( q = k \), then all the children have been fully fed with tasks, and we have \( \varepsilon = 0 \).

We have hence established that

\[
  r_f = \min \left\{ \lambda, \sum_{i=0}^{p} r_i + \varepsilon \times b_{p+1} \right\},
\]

and, since in the fork graph case \( \lambda = b_{-1} \), we have \( r_f = \frac{1}{w_f} \), where \( w_f \) is given by Proposition 1. Consequently, procedure \( BW-First() \) applies Proposition 1 for the fork graphs case.

Assume now that Proposition 2 is true for rank \( h \), i.e. for trees of height \( h \). Let us now prove that Proposition 2 is also true for rank \( h + 1 \). If a node \( P_i \) is not visited while applying procedure \( BW-First() \) (e.g. its parent has no time left to communicate, or no more tasks to delegate), then we can merely remove all the sub-tree rooted in \( P_i \) without influencing on the final throughput of the tree. Assume hence that applying procedure \( BW-First() \) to a tree \( T_{h+1} \) involves visiting a node \( P_i \) of depth \( h \). If \( P_i \) does have some children, this implies that the sub-tree \( F_i \) rooted in \( P_i \) is a fork graph. The procedure \( BW-First() \) applied to the root \( P_i \) of the fork graph \( F_i \) will determine the throughput \( r_{F_i} \) of \( F_i \). The fork graph \( F_i \) is then equivalent to a single node having a computing rate equal to \( r_{F_i} \). Consequently, applying procedure \( BW-First() \) on a tree \( T_{h+1} \) of height \( h + 1 \) is equivalent to applying procedure \( BW-First() \) on a tree \( T_h \) of height \( h \). Then Proposition 2 holds for all \( h \geq 1 \).

The \( BW-First() \) procedure is more efficient than the bottom-up method, since only the nodes that are effectively used in the final schedule are visited. Moreover, it is more convenient through a straightforward recursive implementation. Indeed, we merely traverse the tree in a depth-first manner, and are hence released from the burden of identifying for each step which set of leaves should be transformed. Particularly, the \( BW-First() \) procedure might be a useful tool for topological studies, which aim at determining the best tree overlay network that is built on top of the physical network topology [12]. A quick way to evaluate the throughput of a tree allows to consider a wider set of trees.

Moreover, the \( BW-First() \) procedure can be implemented as a lightweight communication protocol between the nodes of the platform. Indeed, the optimal throughput of the tree is obtained without access to any global information. Each node makes its decisions based on information that is directly measurable plus additional information received from its parent and children. One could term such a distributed protocol semi-autonomous.

For dynamic adaptation concerns, one could imagine the following strategy: The root of the tree, receiving periodically the results of the computations, can measure if there has been a drop in throughput performance. Under a certain threshold, the root might initiate the \( BW-First() \) procedure once more in order to capture the actual state of the platform. Since the messages exchanged between two nodes during the \( BW-First() \) procedure are single numbers, we could argue that the running time of the \( BW-First() \) procedure is negligible as opposed to the time of communicating tasks. However, this last point needs more investigation, and we leave this issue for future work.

Finally, infinite networks have been studied by Bataineh and Robertazzi in [3]. The authors showed that a finite-size network tree load sharing a divisible job can perform almost as well as an infinite network tree. The \( BW-First() \) procedure allows to determine the throughput of infinite network trees, as opposed to the bottom-up method.

6. Reconstructing the Schedule

When the \( BW-First() \) procedure has been executed, each node has all the rational values of its activity variables as its disposal. Hence, during one time unit, let \( \eta_{i-1} = \frac{\mu_i}{\mu_{i-1}} = (\lambda - \delta_i) \) be the number of tasks that node \( P_0 \) receives from its parent, \( \eta_0 = \frac{\mu_0}{\mu_0} = \alpha_0 \) be the number of tasks that \( P_0 \) computes locally, and \( \eta_i = \frac{\mu_i}{\mu_{i-1}} = (\beta_i - \theta_i) \) be the number of tasks that \( P_0 \) sends to each child \( P_i \). Note that all the numerators and denominators are positive integers, and \( \eta_{-1}, \eta_0 \) and \( \eta_i \) can be equal to zero. The steady-state regime is ensured by the fact that node \( P_0 \) receives as many tasks as it can consume. This conservation law translates into equation (1).

\[
  \eta_{i-1} = \sum_{i=0}^{k} \eta_i
\]
Our aim is now to build up a periodic schedule where an integer number of tasks are sent and/or executed. As mentioned in [6], we can obtain a period \( T \) by taking the least common multiple of all the denominators \( \mu_i \) for each node. However, this approach has a major inconvenient: The period might be embarrassingly long, which makes it inconvenient to describe the activity of the nodes, and requires unnecessary large buffering spaces to store the tasks required from one period to another.

### 6.1. Asynchronous Schedule

In order to obtain a more compact description of the schedule, we propose to desynchronize the activities of the *single-port, full overlap* model, i.e. receiving tasks, computing tasks and sending tasks. After all, this model allows to perform these three activities concurrently.

\( T_0^s \) is defined as the shortest period during which node \( P \) receives an integer number \( \varphi_{-1} \) of tasks from its parent; \( T_0^r \) is defined as the shortest period during which node \( P \) executes an integer number \( \varphi_0 \) of tasks; and \( T_0^s \) is defined as the shortest period during which node \( P_0 \) sends an integer number \( \varphi_i \) of tasks to each child \( P_i \).

**Lemma 1** With the above notations, the minimal periods as well as the integer number of tasks treated are obtained as follows:

\[
\begin{aligned}
T_0^s &= \text{lcm}\{\mu_i | i \in C_0\} \\
T_0^r &= \mu_0 \\
T_0^s &= T_{-1}^s \\
\varphi_i &= \eta_i \times T_0^r, \forall i \in C_0 \\
\varphi_{-1} &= \eta_{-1} \times T_0^r
\end{aligned}
\]

**Proof** Node \( P_0 \) must send \( \eta_i = \frac{\rho_i}{\mu_i} \) tasks per time unit to each child \( P_i \). In order to obtain a minimal period where an integer number of tasks is sent to each child, we have to take the least common multiple of all the denominators \( \{\mu_i | i \in C_0\} \).

Node \( P_0 \) must compute \( \eta_0 = \frac{\rho_0}{\mu_0} \) tasks per time unit, which gives a minimal period of \( \mu_0 \) time units during which \( \rho_0 \) tasks are computed.

Since any node \( P_0 \) receives tasks only from its parent, the receiving period \( T_0^r \) of \( P_1 \) should be equal to the sending period \( T_{-1}^s \) of its parent \( P_{-1} \), which has been shown to be minimal. Obviously the root of the tree should not receive any tasks, and we can enforce \( T_{-1}^{r,\text{root}} = 0 \).

**Proposition 3** Any node \( P_0 \) can desynchronize its activities according to Lemma 1 without violating the conservation law.

**Proof** By taking the least common multiple of the three asynchronous periods, we obtain a period \( T_0 \) during which all the received tasks are consumed. That is to say, every \( T_0 \) time units, \( P_0 \) receives an integer number \( \chi_{-1} \) of tasks from its parent, computes an integer number \( \chi_0 \) of tasks, and sends an integer number \( \chi_i \) of tasks to each child \( P_i \). This translates into equation set (3).

\[
\begin{align*}
T_0 &= \text{lcm}\{T_0^s, T_0^r, T_0^s\} \\
\chi_{-1} &= \eta_{-1} \times T_0, \forall i \in C_0 \\
\chi_0 &= \eta_0 \times T_0 \\
\chi_{-1} &= \sum_{i=0}^k \chi_i
\end{align*}
\]

The only requirement for ensuring steady-state with asynchronous activities is to dispose of enough tasks buffered at node locations. For now, assume that \( \chi_{-1} \) tasks have been buffered during the start-up phase. Then, we have a steady number of tasks stored from one period \( T_0 \) to another, which ensures steady-state behavior.

### 6.2. Event-Driven Schedule

We now propose an event-driven schedule, where any time-related information has been removed (except for the root node). Consider first the case of any node \( P_0 \) different from the root. Lemma 1 gives the minimal period \( T_0^s \) and \( T_0^r \) for computing and sending tasks respectively. The minimal period \( T_0^s \) during which an integer number of tasks is consumed (either processed locally or delegated to a child) can be obtained by taking the least common multiple of \( T_0^s \) and \( T_0^r \). Let \( \psi_0 \) be the number of tasks executed by \( P_0 \) every \( T_0^r \) time units, and \( \psi_i \) be the number of tasks delegated to child \( P_i \) every \( T_0^r \) time units. We have hence the following equation set:

\[
\begin{align*}
T_0^s &= \text{lcm}\{T_0^s, T_0^r\} \\
\psi_0 &= \eta_0 \times T_0^r \\
\psi_i &= \eta_i \times T_0^r, \forall i \in C_0
\end{align*}
\]

\( P_0 \) does not need time-related information any longer. Instead, \( P_0 \) will handle incoming tasks by bunches of size \( \Psi = \sum_{i=0}^k \psi_i \). Indeed, of all the tasks that \( P_0 \) will receive from its parent, \( \psi_0 \) fraction of them are intended for itself, and \( \psi_i \) fraction of them are intended for each child \( P_i \) (provided that \( \psi_1 > 0 \)). Therefore, the only information which is necessary is to know how many tasks should be executed locally (i.e. \( \psi_0 \)), and how many tasks should be delegated to each child \( P_i \) (i.e. \( \psi_i \)) every \( T_0^r \) time units. \( P_0 \) will then handle incoming tasks by bunches of size \( \Psi \), without using any time-based information. The event-driven schedule for any node \( P_0 \) different from the root is summarized by procedure *Steady-State()*.  

Since the computing platform is a tree, nodes receive tasks only from their parent. One can then let a receiving thread blocked in reception, waiting for tasks to arrive from the parent, and storing them locally upon reception. Or one can use non-blocking receive calls. Consequently, we do not need to describe further the receiving activity.
6.3. Local Scheduling

Although the event-driven schedule ensures the steady-state regime, some scheduling decisions remain to be taken. Indeed, $P_0$ will treat tasks received from its parent by bunches of size $\Psi$, but in which order should it delegate tasks to its children? And which tasks should be kept for itself? All the schedules are equivalent in terms of steady-state throughput. However, some schedules might be more advantageous than others with respect to memory limitations. The one we are proposing has been designed with the objective of minimizing the number of tasks that will be buffered at steady-state. Obviously, minimizing the number of tasks buffered at steady-state, is of interest since it reduces memory usage during the computation. In addition, the low number of tasks required to enter the steady-state regime, will lead to fast start-up and wind-down phases (see Section 7).

Our local schedule strategy interleaves the incoming tasks proportionally to the $\psi$ quantities ($\varphi$ for the root). Let us start with node $P_0$ itself which should compute $\psi_0$ tasks. We merely split the unity domain into ($\psi_0 + 1$) parts, each of size $\Delta_0 = \frac{1}{\psi_0}$. The same operation is repeated for each child $P_i$, and the unity domain is split into ($\psi_i + 1$) parts of size $\Delta_i = \frac{1}{\psi_i} (\forall \psi_i > 0)$. When this is done, we obtain an order among the incoming tasks that corresponds to our allocation of tasks to nodes. Let us take an example to illustrate this strategy. Consider a node $P_0$ having two children $P_1$ and $P_2$. We let $\psi_0 = 1$, $\psi_1 = 2$ and $\psi_2 = 4$. The allocation for our example is depicted in Figure 3. The first task is sent to $P_2$, the second to $P_1$, the third to $P_2$, etc.

![Figure 3. Scheduling the incoming tasks](image)

If two processors are contesting for one task, we arbitrarily prioritize the processor with smallest $\psi$. If both processors have an equal $\psi$, we prioritize the one with smallest index. The rationale behind this strategy, is to space out tasks intended to one node as much as possible. Instead of giving the nodes all their tasks at once, we disseminate them along the period, making it possible for the nodes to consume tasks almost as fast as they receive them. Furthermore, due to symmetrical reasons, the description of the local schedules can be divided by two.

7. Efficient Start-Up Phase

Usually, the start-up phase is considered as just a way to enter the steady-state regime [2, 5, 6]. The traditional answer to this problem is to send tasks down the tree without doing any useful computation, until each node gets the number of tasks required to enter the steady-state regime. This takes $T$ times the maximum depth of the tree, where $T$ is the steady-state period of the tree [5].

Alternatively, Kreaseck et al. [12] propose a demand-driven start-up strategy, where nodes request tasks to their parent, which in turn will forward the demands up the hierarchy. However, they observed that in practice, their protocol experienced long start-up phases.

It is of prior importance to be able to reach steady-state as fast as possible. Indeed, under dynamic conditions, re-
computing the optimal schedule might be necessary in order to efficiently utilize the platform. Under these conditions, not being able to execute any tasks during the start-up phase becomes no longer acceptable. For these reasons, the importance of the start-up and wind-down phases should not be minimized. We propose a start-up phase where computations are allowed. In fact, every node will act according to its event-driven schedule from the beginning of the computation.

**Proposition 4** Applying the event-driven schedule, from the beginning of the computation, leads every node \( P_0 \) to its steady-state regime, in at most \( \sum T_i^\omega \mid i \in A_0 \) time units, where \( A_0 \) is the set of ancestors of node \( P_0 \).

**Proof** Intuitively, during the start-up phase, nodes will receive tasks at full rate, but since they do not have buffered tasks at their disposal, will starve waiting for them to arrive. Nodes will store locally incoming tasks and schedule them immediately. Hence, node buffers will be filled up with tasks, much like the way a pipeline is getting full, until the task consuming rate catches up with the task receiving rate.

Formally, consider any node \( P_0 \) different from the root. From equation set (3) at steady-state we have: \( P_0 \) receives \( \chi_1 - \lambda \) tasks every \( T_i^\omega \) time units from its parent \( P_{i-1} \). Assume that at time step \( t \), \( P_{i-1} \) is in steady-state. Assume also that during the time period \([t, t + T_{i-1}^\omega]\), \( P_0 \) can consume only \( \lambda \) of these \( \chi_1 \) tasks (since \( P_0 \) does not have any new tasks yet, some time will be spent waiting for them to arrive). The \( (\chi_1 - \lambda) \) tasks left are hence stored by \( P_0 \) for the next period of time \( T_{i-1}^\omega \). During the time period \([t + T_{i-1}^\omega, t + 2T_{i-1}^\omega]\), \( P_0 \) receives \( (\chi_1 - \lambda) \) new tasks. We know that \( P_0 \) can consume \( \lambda \) of the new tasks. Since \( P_0 \) owns \( (\chi_1 - \lambda) \) tasks from the previous period, \( P_0 \) is no longer idle while waiting for tasks. Hence, \( P_0 \) consumes \( \lambda + (\chi_1 - \lambda) = \chi_1 \) new tasks and is in steady-state at time-step \( t + T_{i-1}^\omega \). Since the root of the tree is already in steady-state from the beginning of the computation (at \( t = 0 \)), applying this reasoning from the root down the hierarchy gives \( t = \sum T_i^\omega \mid i \in A_i \).

It is important to point out that, due to the continuity between one time period to another, \( P_i \) will enter into its steady-state regime earlier than time step \( t = \sum T_i^\omega \mid i \in A_i \). While the entire tree enters the steady-state regime as soon as all the nodes entered in steady-state. Such behavior can be observed in the example of Section 8.

### 8. Example

Let us illustrate our results with an example taken from [4]. Consider the tree \( T \) depicted in Figure 4 (a). Procedure **BW-First** obtained a throughput of 10 tasks every 9 time units, which corresponds to the result obtained.
by the bottom-up method of Beaumont et al. [4]. The successive transactions established during the BW-First() procedure are depicted in Figure 4 (b). Note that nodes \( P_5, P_9, P_{10} \) and \( P_{11} \) were not visited, meaning that they will not be used in the final schedule. The number of tasks that each node receives \( (\tau_{i-1}) \) and computes \( (\tau_h) \) per time unit are depicted in Figure 4 (c). The final description of the local schedules is very compact and is depicted in Figure 4 (d). The final computation, with start-up and wind-down phases is depicted with a Gantt diagram in Figure 5.

We would like to point out few interesting observations. The tree has a steady-state period \( T \) of 360 time units, while the rootless tree has a throughput of 40 tasks every 40 time units. The start-up phase lasts for 40 time units, which is equal to one steady-state period of the rootless tree. During the start-up phase, the rootless tree executes 32 tasks, that is to say 80% of its optimal throughput. At an arbitrary point in steady-state (time step 115), we stopped delegating tasks to the tree, and observed that the wind-down phase lasts for only 10 time units (4 times shorter than the steady-state period of the rootless tree). This very short wind-down phase is the result of our local schedule strategy, which aim at minimizing the number of tasks buffered during steady-state.

9. Future Work

Handling the return of the results back to the master should be considered for future work. The bandwidth-centric principle does not hold when the return of the results are considered, despite the claim of Beaumont et al. [5] and Kreaseck et al. [12]. In their work, the aforementioned authors model the communication times between two processors \( P_i \) and \( P_j \) as being the time needed by a parent \( P_i \) to communicate the data for one task to a child \( P_j \) plus the time for the child to return the result when it is finished. They argue that “for the purpose of computing steady-state behavior, it does not matter what fraction of the communication time is spent sending a problem and what fraction is spent receiving the results” [5]. We will show that this simplification is erroneous. Although the simplification holds for the traffic of messages on the communication links, it neglects the receiving port resource. Let us illustrate this with a small platform example composed of three nodes. The master has only two children. Each child can process 1 task per time unit. It takes 0.5 time units to send one task from the master to its children, and 0.5 time units to return the results of one task from the children to the master. The optimal throughput of the platform is then 2 tasks per time units. If we join the time sending the input data with the time for receiving the results (as suggested in [5,12]), we obtain a platform throughput of 1 task per time unit. Hence, the simplification does not work for returning the results back to the master, and consequently this problem is still open.

It would be interesting to evaluate the BW-First() procedure using simulations (for example with the SimGrid toolkit [8]), and compare it to the autonomous protocol proposed by Kreaseck et al. [12]. Especially, measuring the overhead incurred by the global synchronization phase would give some insight on how frequently the BW-First() procedure might be initiated by the root. Finally, trying different local schedules might be interesting with respect to start-up and wind-down phases as well as memory limitations.

Figure 5. Final computation: \( S = \text{Send}, \ C = \text{Compute}, \ R = \text{Receive} \).
10. Conclusion

The problem of allocating a large number of independent, equal-sized tasks to heterogeneous trees was considered. We assumed that a specific node, the master initially, holds the data associated to the tasks, and that returning the results of the computations to the master is negligible. This paper made the following contributions to this problem:

- We proposed a distributed method, the $BW$-First() procedure which is an efficient, practical and scalable implementation of the theoretical results presented in [5].
- Based on the results of the $BW$-First() procedure, each node can then build up its own local schedule independently of the rest of the platform. The result is a loosely synchronized schedule, where nodes are synchronized only with their children, as opposed to the traditional approach where all the nodes of the platform are synchronized together.
- The resulting local schedules are event driven, meaning that every node (except the root) acts without any time-related information, and consequently, their description is very compact.
- We proposed a local schedule strategy that makes use of a small amount of tasks buffered at steady-state. Not only this approach requires less memory, but it also considerably reduces the length of the start-up and wind-down phases.
- We presented a start-up phase strategy which allows useful computation as opposed to the traditional approach.

The goal of this paper was to close the gap between theory and practice by embedding theoretical knowledge into a practical and scalable implementation. We believe that the techniques presented in this paper are valuable for conceiving scheduling strategies that tackle the platform dynamics, i.e. where resources exhibit dynamic performance characteristics and availability.

Acknowledgments

The author would like to thank his PhD thesis adviser Anne C. Elster as well as Olivier Beaumont for their comments and suggestions, which greatly improved the final version of the paper. This research was supported by the Department of Computer and Information Science (IDI), at the Norwegian University of Science and Technology (NTNU).

References