A Realistic Model and an Efficient Heuristic for Scheduling with Heterogeneous Processors

Olivier Beaumont, Vincent Boudet, Yves Robert

HAL Id: hal-02101846

https://hal-lara.archives-ouvertes.fr/hal-02101846

Submitted on 17 Apr 2019

A Realistic Model and an Efficient Heuristic for Scheduling with Heterogeneous Processors

Olivier Beaumont
Vincent Boudet
Yves Robert

September 2001

A Realistic Model and an Efficient Heuristic for Scheduling with Heterogeneous Processors

Olivier Beaumont
Vincent Boudet
Yves Robert

September 2001

Abstract

Scheduling computational tasks on processors is a key issue for high-performance computing. Although a large number of scheduling heuristics have been presented in the literature, most of them target only homogeneous resources. Moreover, these heuristics often rely on a model where the number of processors is bounded but where the communication capabilities of the target architecture are not restricted. In this paper, we deal with a more realistic model for heterogeneous networks of workstations, where each processor can send and/or receive at most one message at any given time-step. First, we state a complexity result that shows that the model is at least as difficult as the standard one. Then, we show how to modify classical list scheduling techniques to cope with the new model. Next we introduce a new scheduling heuristic which incorporates load-balancing criteria into the decision process of scheduling and mapping ready tasks. Experimental results conducted using six classical testbeds: (LAPLACE, LU, STENCIL, FORK-JOIN, DOOLITTLE, and LDMt) show very promising results.

Keywords: heterogeneous processors, scheduling, mapping, scheduling heuristics, complexity

Résumé

Nous nous intéressons à l'ordonnancement de graphes de tâches avec des ressources de calcul hétérogènes. Nous pensons que le modèle usuel *macro-dataflow* n'est pas adapté, car les contentions sur les communications ne sont pas prises en compte. Nous proposons d'utiliser plutôt le modèle *one-port*, où un processeur donné peut envoyer/recevoir un seul message à tout instant. Ce modèle s'avère aussi difficile que le modèle usuel: l'ordonnancement d'un simple graphe *fork* est prouvé NP-complet. Nous montrons comment adapter les heuristiques de liste classiques au modèle *one-port*. Enfin, nous proposons une nouvelle heuristique, dont la caractéristique principale est de traiter un ensemble de tâches prêtes (plutôt qu'une seule) simultanément: le but est d'assurer un meilleur équilibrage de charge et de minimiser les communications.

Mots-clés: processeurs hétérogènes, heuristiques d'ordonnancement, distribution, équilibrage de charge, complexité

1 Introduction

The efficient scheduling of application tasks is critical to achieving high performance in parallel and distributed systems. The objective of scheduling is to find a mapping of the tasks onto the processors, and to order the execution of the tasks so that: (i) task precedence constraints are satisfied; and (ii) a minimum schedule length is provided.

Task graph scheduling is usually studied using the so-called *macro-dataflow* model, which is widely used in the scheduling literature: see the survey papers [16, 1, 4, 7] and the references therein. This model was introduced for homogeneous processors, and has been (straightforwardly) extended for heterogeneous computing resources. In a word, there is a limited number of computing resources, or processors, to execute the tasks. Communication delays are taken into account as follows: let task T be a predecessor of task T' in the task graph; if both tasks are assigned to the same processor, no communication overhead is paid, the execution of T' can start right at the end of the execution of T; on the contrary, if T and T' are assigned to two different processors P_i and P_j, a communication delay is paid. More precisely, if P_i finishes the execution of T at time-step t, then P_j cannot start the execution of T' before time-step $t + comm(T, T', P_i, P_j)$, where $comm(T, T', P_i, P_j)$ is the communication delay (which depends upon both tasks T and T' and both processors P_i and P_j). Because memory accesses are typically one order of magnitude cheaper than inter-processor communications, it makes good sense to neglect them when T and T' are assigned to the same processor.

However, the major flaw of the macro-dataflow model is that communication resources are not limited. First, a processor can send (or receive) any number of messages in parallel, hence an unlimited number of communication ports is assumed (this explains the name *macro-dataflow* for the model). Second, the number of messages that can simultaneously circulate between processors is not bounded, hence an unlimited number of communications can simultaneously occur on a given link. In other words, the communication network is assumed to be contention-free, which of course is not realistic as soon as the processor number exceeds a few units.

We strongly believe that the macro-dataflow task graph scheduling model should be modified to take communication resources into account. Recent papers [12, 13, 23, 22], made a similar statement and introduced variants of the model (see the discussion in Section 2). In this paper, we suggest to use the bi-directional one-port architectural model, where each processor can communicate (send and/or receive) with at most another processor at a given time-step. In other words, a given processor can simultaneously send a message, receive another message, and perform some (independent) computation. The next section (Section 2) is devoted to a brief discussion of all these scheduling models: (i) the macro-dataflow model extended to deal with heterogeneous resources, (ii) the variants suggested in the literature referenced above, and (iii) the bi-directional one-port model.

The new one-port model turns out to be computationally even more difficult than the macro-dataflow model: in Section 3, we prove that scheduling a simple fork graph with an unlimited number of homogeneous processors is NP-hard. Note that this problem has polynomial complexity in the macro-dataflow model [10]: we have to resort to fork-join graphs to get NP-completeness in the macro-dataflow model [4].

An impressive list of scheduling heuristics has been proposed in the literature for the macro-dataflow model with a limited number of homogeneous processors (see the tutorial [1] and the references therein). More recently, several heuristics have been introduced to deal with different-speed processors [15, 17, 26, 21, 3]. Unfortunately, all these heuristics assume no restriction on the communication resources, which renders them somewhat unrealistic to model real-life applications. Section 4 is devoted to the design and analysis of a new heuristic targeted to scheduling task graphs with a limited number of different-speed processors, under the bi-directional one-port communication model.

In Section 5, we report simulation results from comparisons conducted using six classical testbeds: LAPLACE, LU, STENCIL, FORK-JOIN, DOOLITTLE, and LDMt. We obtain very favorable results. Finally, some concluding remarks are given in Section 6.

2 Models

2.1 The macro-dataflow model

In this section, we briefly recall the macro-dataflow model, which is widely used in the scheduling literature [1]. This model was introduced for homogeneous processors but has been extended to deal heterogeneous computing resources. For each task scheduling algorithm, the input is composed of two entities:

1. A directed vertex-weighted edge-weighted acyclic graph $\mathcal{G} = (V, E, w, c)$, that models the application to be scheduled.

2. A set of computing resources $\mathcal{P} = (P, t, link)$ that models the target computing resources (processors and communication network).

The relationships between $\mathcal{G}$ and $\mathcal{P}$ are the following:

Computation

- $V = \{v_i : i = 1, \cdots, N\}$ is a set of N nodes (or tasks). Each task v has a nonnegative computation cost $w(v) \in \mathbb{N}$ which is defined as the amount of computation cycles needed to process it.

- $P = \{P_i : i = 1, \cdots, p\}$ is a set of p processors. Each processor P_i has a cycle-time t_i, which is defined as the inverse of its (relative) speed. For instance if processor P_1 is twice faster, say, than processor P_2, then $t_2 = 2t_1$.

- The number of time-steps required to execute a task v on processor P_i is the product $w(v) \times t_i$ of the task computation cost by the processor cycle-time[1]. If all processors are identical, then we let $t_i = 1$ for $1 \leq i \leq p$.

- For each task v_i, $\sigma(v_i)$ is the time-step at which its execution begins. We let $alloc(v_i)$ be the number of the processor which v_i is assigned to.

- Any processor can compute and communicate simultaneously, but can execute at most one task at each time-step. Tasks are non-preemptive: once started on a given processor, their execution must continue until completion.

Precedence constraints

- Each edge $e_{i,j} \in E$ corresponds to a precedence constraint from task v_i to task v_j and is labeled with a communication volume $data(i, j)$, which is the number of data items to be transferred from v_i to v_j after the execution of v_i.

- For each edge $e_{ij} : v_i \to v_j$, if v_i is executed on processor P_q and v_j on processor P_r (in other words if $alloc(v_i) = q$ and $alloc(v_j) = r$), we have the scheduling constraint

$$\sigma(v_i) + w(v_i) \times t_q + comm(i, j, q, r) \leq \sigma(v_j)$$

which states that the execution of v_j on P_r cannot start before the end of the execution of v_i on P_q, i.e. $\sigma(v_i) + w(v_i) \times t_q$, plus some communication overhead $comm(i, j, q, r)$ that is detailed below (and chosen to be zero whenever $q = r$).

Communication

- The communication matrix *link* models the time needed to transfer a single data item from one processor to another. As stated above, we assume that the main diagonal of the 2D matrix *link* is composed of zero entries.

---

[1] This is the so-called *consistent* restriction of a more general model where the cost of executing a task on a processor is given by the corresponding entry of a two-dimensional matrix. When the rows of this matrix are not comparable, it means that a processor P_i which executes a task v faster than another processor P_j may well be slower than P_j for the execution for another task v', hence the name *inconsistent* in this general setting. All the results and heuristics of this paper can directly be extended to cope with the fully general model.

- The communication overhead $comm(i, j, q, r)$ is equal to

$$comm(i, j, q, r) = data(i, j) \times link(q, r),$$

i.e. the product of the message length by the capacity of the communication link.

Objective

The objective is to minimize the *makespan*, or scheduling length, i.e.

$$\max_{v \in V} \left(\sigma(v) + w(v) \times t_{alloc(v)} \right).$$

This scheduling problem is NP-complete in the macro-data flow model, even for simple fork-join graphs with an infinite number of same-speed processors ($t_i = 1$ for all i), and a fully homogeneous communication network ($link(i, j) = 1$ forall $i \neq j$): see [4].

2.2 Communication-aware models from the literature

Communication-aware models restrict the use of communication links in various manners. In the model proposed by Sinnen and Sousa [23, 22, 24], the underlying communication network is no longer fully-connected. There are a limited number of communication links, and each processor is provided with a routing table which specifies the links to be used to communicate with an other processor (hence the routing is fully static). The major modification is that at most one message can circulate on one link at a given time-step, so that contention for communication resources is taken into account.

Similarly, Hollermann et al. [12] and Hsu et al. [13] target networks of processors and introduce the following model: each processor can either send or receive a message at a given time-step (bidirectional communication is not possible); also, there is a fixed latency between the initiation of the communication by the sender and the beginning of the reception by the receiver. Still, the model is rather close to the standard one-port model discussed below.

Finally, note that there are several other papers that include restrictions on the communication resources: these include work by Tan et al. [25], Orduna et al. [18] and Roig et al. [20].

2.3 The bi-directional one-port model

As stated above, communication resources are taken into account for the bi-directional one-port model. This is quite natural, and quite similar to the assumptions made for computation resources in the macro-dataflow model.

Formally, we keep all previous notations and scheduling rules, and we add the following new rule: at a given time-step, any processor can communicate with at most another processor in both directions: sending to *and* receiving from another processor. We also assume communication/computation overlap (but as before, a processor can execute at most one task at each step). Note that several communications can occur in parallel, provided that they involve disjoint pairs of sending/receiving processors. The one-port model nicely models switches like Myrinet that can implement permutations [5] or even multiplexed bus architectures [14].

Several variants could be considered: no communication/computation overlap, uni-directional communications, or even a combination of both restrictions. But the bi-directional one-port model seems closer to the actual capabilities of modern processors.

Serializing communications performed by the processors has a dramatic impact on the scheduling makespan. Consider the following simple example of the task graph represented in Figure 1: $w(v_i) = 1$ for $0 \leq i \leq 6$ and $data(0, i) = 1$ for $1 \leq i \leq 6$. Assume five same-speed processors and a fully homogeneous network: $t_i = 1$ for $1 \leq i \leq 5$ and $link(i, j) = 1$ for $1 \leq i, j \leq 5, i \neq j$. In the macro-dataflow model, we assign v_0 and the first two children v_1 and v_2 to processor P_0. We assign one of the remaining children v_3, v_4, v_5 and v_6 to each remaining processor. Processor P_0 executes task v_0 at time-step 0; then P_0 can perform all the four communications in parallel at time-step 1. The total makespan is then equal to 3. In the one-port model, the same allocation of tasks to processors would lead to a makespan at least 6: 1 for the parent task, 4 for the four messages to be sent sequentially, and 1 for the last task to be executed. One optimal solution is
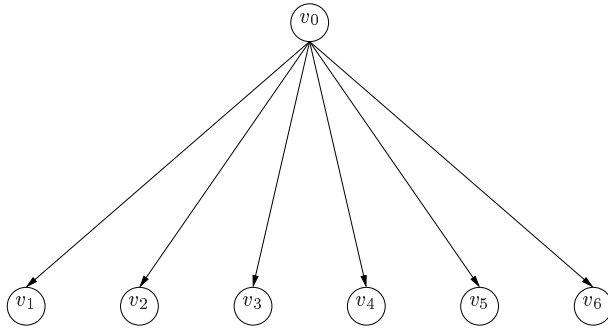
Figure 1: Task graph for the example: all weights (nodes and communications) are equal to 1.
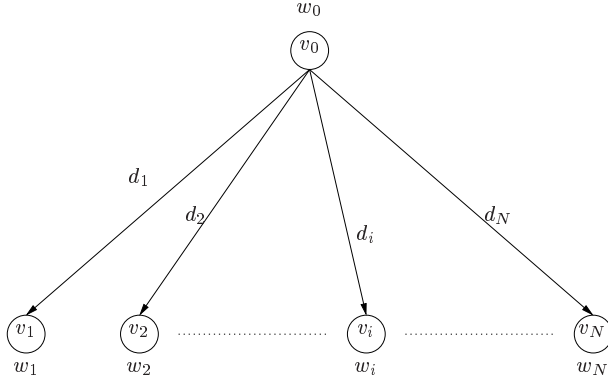

Figure 2: A fork-graph

to assign three children tasks to P_0 and one remaining child task to a distinct processor (which makes one processor useless), for a makespan equal to 5. It is clear that communications from the parent node to the children has become the bottleneck. Of course we could use larger task graphs and greater communication costs to come up with arbitrarily large differences in the makespans.

3 Complexity

In this section, we prove a NP-completeness result for the one-port scheduling model. A N-children fork-graph is a task-graph of $N+1$ nodes labeled $v_0, v_1, \ldots, v_N$, as illustrated in Figure 2. There is an edge directed from the parent node v_0 to each child node v_i, $1 \leq i \leq N$. To simplify notations, we let $w_i = w(v_i)$ for $0 \leq i \leq N$ and $d_i = data(0, i)$ for $1 \leq i \leq N$. We target a simple architecture with an unlimited number of same-speed processors and a fully homogeneous communication network. With the above notations, $p = N+1$ (we never need more processors than tasks), $t_i = 1$ for $1 \leq i \leq p$, $link(i, j) = 1$ for $1 \leq i, j \leq p, i \neq j$ (and $link(i, i) = 0$ for $1 \leq i \leq p$).

Given a fork graph and the target architecture (unlimited number of same-speed processors connected through a fully homogeneous network), the decision problem is the following:

Definition 1 *FORK-SCHED(G,P,T): Given a fork-graph G of $N+1$ nodes, a set P of an unlimited number of same-speed processors connected through a fully homogeneous network, and given a time-bound T, is there a valid schedule σ whose makespan is not greater than T?*

Theorem 1 *The FORK-SCHED(G,P,T) decision problem is NP-complete.*

4

Proof We use a reduction from 2-PARTITION, a well-known NP-complete problem [8]: given a set of n integers $\mathcal{A} = \{a_1, \ldots, a_n\}$, is there a partition of $\{1, \ldots, n\}$ into two subsets $\mathcal{A}_1$ and $\mathcal{A}_2$ such that

$$\sum_{i \in \mathcal{A}_1} a_i = \sum_{i \in \mathcal{A}_2} a_i \quad ?$$

We start with an arbitrary instance of 2-PARTITION, i.e. a set $\mathcal{A} = \{a_1, \ldots, a_n\}$ of n integers. We have to polynomially transform this instance into an instance of the FORK-SCHED problem which has a solution iff the original instance of 2-PARTITION has a solution.

We let $2S = \sum_{i=1}^{n} a_i$ (if the sum is odd there is no solution to the instance of 2-PARTITION). Let $M = \max_{1 \le i \le n} a_i$ and $m = \min_{1 \le i \le n} a_i$. We construct the following instance of FORK-SCHED:

- the fork-graph has $N + 1$ nodes, where $N = n + 3$

- the parent node v_0 has weight $w_0 = 0$

- for $1 \le i \le n$, the i-th child node v_i has weight $w_i = 10(M + a_i + 1)$

- the last three children have the same weight $w_{n+1} = w_{n+2} = w_{n+3} = 10(M + m) + 1$. Let $w_{\min}$ denote this common value, this is indeed the minimum of $w_i, 1 \le i \le n + 3$.

- data volumes: for $1 \le i \le n + 3$, $d_i = w_i$

- time bound: $T = \frac{1}{2} \sum_{i=1}^{n} w_i + 2w_{\min} = 5n(M + 1) + 10S + 20(M + m) + 2$

Note that $w_{\min} \le w_i \le 2w_{\min}$ for $1 \le i \le n$ (straightforward verification). Clearly, the size of the constructed instance of FORK-SCHED is polynomial (even linear) in the size of the original instance of 2-PARTITION.

Assume that the original instance of 2-PARTITION admits a solution: let $\mathcal{A}_1$ and $\mathcal{A}_2$ be a partition of $\{1, \ldots, n\}$ such that $\sum_{i \in \mathcal{A}_1} a_i = \sum_{i \in \mathcal{A}_2} a_i = S$. We derive a scheduling for the instance of FORK-SCHED as follows:

- Processor P_0 is assigned the execution of node v_0, nodes $v_i, i \in \mathcal{A}_1$ and nodes v_{n+1} and v_{n+2}. Obviously, P_0 needs exactly T units of time to process these tasks.

- Each other node is assigned to a distinct processor, hence we are using $|\mathcal{A}_2| + 1$ processors in addition to P_0

- The ordering of the communication messages sent by P_0 is by increasing values of the index i; in particular, the last message sent by P_0 is to node v_{n+3}

- The processor responsible for node v_{n+3} completes the reception of the message from P_0 at time-step $\sum_{i \in \mathcal{A}_2} d_i + d_{n+3}$, and terminates the execution at time-step $\sum_{i \in \mathcal{A}_2} d_i + d_{n+3} + w_{\min} = T$

- All the other processors terminate their execution earlier, because they receive their message not later than $\sum_{i \in \mathcal{A}_2} d_i$ and their execution time w_i is not greater than $2w_{\min}$

Therefore, we have derived a valid scheduling that matches the time-bound, hence a solution to the FORK-SCHED instance.

Reciprocally, assume that the FORK-SCHED instance admits a solution, i.e. a valid scheduling σ that achieves the time-bound T. Let P_0 be the processor which executes v_0, and $\mathcal{A} = \{i, 1 \le i \le n + 3, alloc(v_i) = 0\}$ be the index set of the tasks assigned to P_0. The processing time of P_0 is thus at least $A = \sum_{i \in \mathcal{A}} w_i$. All the remaining tasks are assigned to other processors than P_0. The processor which receives the last message from P_0 to execute a task, say, v_{last} (whose index is not in $\mathcal{A}$), cannot complete execution before time-step $B = \sum_{i \notin \mathcal{A}, 1 \le i \le n+3} d_i + w_{last}$. Since σ achieves the time bound, $\max(A, B) \le T$. But $A + B = \sum_{i=1}^{n+3} w_i + w_{last} = 2T + w_{last} - w_{\min}$, hence $A = B = T$ and $w_{last} = w_{\min}$. Since $A = B$, $A \equiv B \mod 10$, hence $\mathcal{A}$ contains exactly two indices of the set $\{n + 1, n + 2, n + 3\}$. We let $\mathcal{A}_1$ be equal to $\mathcal{A}$ minus these two indices and $\mathcal{A}_2 = \{1, \ldots, n\} \setminus \mathcal{A}_1$ to derive a solution to the original instance of 2-PARTITION. ■

4 Heuristics

In this section, we introduce a new heuristic for the one-port model. This heuristic builds upon ideas from the HEFT heuristic and from the ILHA heuristic, both designed for the macro-dataflow model. We briefly review these heuristics before discussing their adaptation to the one-port model.

4.1 HEFT for the macro-dataflow model

In this section we briefly describe the *Heterogeneous Earliest Finish Time* (HEFT) heuristic introduced by Topcuoglu, Hariri and Wu [26] for the macro-dataflow model. This heuristic is a natural extension of list-scheduling heuristics to cope with heterogeneous resources. More in particular, HEFT builds upon the old Modified Critical Path heuristic [9, 6] and use bottom levels to assign priorities to tasks.

More precisely, the HEFT heuristic works as follows:

- the task graph is traversed so that the bottom level of each task is computed. The bottom level of a task is defined as the length of the longest path that leads to an exit node in the graph (intuitively, the longer the path, the more urgent the task).

- bottom levels are used to assign priorities to tasks

- at each step, a ready task (i.e. a task whose predecessors have all been scheduled) with highest priority is selected for scheduling

- the task is assigned to the processor that allows the earliest completion time, taken into account all previous decisions; the task is then marked "scheduled" and the list of ready tasks is updated

Further explanations are in order. First, how to compute bottom levels with different speed processors? Because the length of a path in the graph is the sum of computation and communication times, we need to properly average those to define bottom levels in this context, as explained below.

As for computation times, assume that there are p available processors of respective cycle-times $t_1, \ldots, t_p$. Assume also that there is a collection of several independent tasks of total weight W. Ideally, these tasks should be distributed to processors so that the load is equally balanced. Processor P_i should receive a fraction c_i (with $0 \leq c_i \leq 1$) of the total weight W such that its processing time $(c_i W) t_i$ is the same as that of all processors. We derive

$$c_i = \frac{\frac{1}{t_i}}{\sum_{i=1}^{p} \frac{1}{t_i}}.$$

The tasks are processed within $\frac{W}{\sum_{i=1}^{p} \frac{1}{t_i}}$ time-units by the p processors. We deduce that the weight $w(T)$ of a given task should be estimated by the quantity $\frac{p \times w(T)}{\sum_{i=1}^{p} \frac{1}{t_i}}$ when computing bottom-levels.

Similarly, the weight of a communication should be multiplied by a factor estimated to the average bandwidth of the links (replace $link(q, r)$ by the inverse of the harmonic mean). Note that all communication costs are accounted for in the calculation of bottom levels. In other words, it is (conservatively) estimated that communications cannot be avoided (by assigning the source and the sink to the same processor).

4.2 ILHA for the macro-dataflow model

In a previous paper [3], we have introduced the *Iso-Level Heterogeneous Allocation (ILHA)* heuristic for the macro-dataflow model. In a word, the main characteristic of the ILHA heuristic is a better load-balancing at each decision step, which is achieved by considering a chunk of several ready tasks rather than a single one; the idea is to allocate to each processor a number of the tasks in the chunk that is proportional to its computing power.

We have outlined how to achieve a good load-balancing in the previous section: each processor P_i with cycle-time t_i should receive a fraction c_i of the total work of size W to be executed. There is a slight complication due to the fact that tasks are indivisible units of computation, so that the values c_i may have to be replaced by approximations. For instance when W corresponds to n independent tasks, each requiring

the same amount of work, P_i receives $c_i n$ tasks, which is an integer for all $1 \leq i \leq p$ only if n is a multiple of $C = lcm(t_1, t_2, \cdots, t_p) \sum_{j=1}^{p} \frac{1}{t_j}$, a quantity that may be very large. For the general case, the following algorithm provides the best solution [2]:

OPTIMAL DISTRIBUTION

1: $\forall i \in \{1, \ldots, p\}$, $c_i = \left\lfloor \dfrac{\frac{1}{t_i}}{\sum_{i=1}^{p} \frac{1}{t_i}} \times n \right\rfloor$.

2: **for** $m = c_1 + c_2 + \ldots + c_p$ **to** n

3:    find $k \in \{1, \ldots, p\}$ such that $t_k \times (c_k + 1) = \min\{t_i \times (c_i + 1))\}$

4:    $c_k = c_k + 1$

We are ready for a first outline of the ILHA heuristic. We split the task graph into *levels* made up of independent tasks, by considering the tasks that will be ready at the same time-step. In other words, two tasks belong to the same *level* if they have the same top-level, using the terminology of [27]. This is done by a traversal of the graph. Initially, the 0-level is composed of the entry tasks. The $(i+1)$-th level groups the tasks that are ready when the i-th level is achieved. A first version of the $ILHA$ algorithm is the following: we traverse the task graph to split it into levels made of independent tasks. We compute the number of tasks that we allocate to each processor using the load-balancing algorithm. Once this is done, we have to determine exactly which task is given to each processor. The criteria is to minimize the communication costs. So for each task of the level, we consider its predecessors. If they are all allocated to the same processor, we try to allocate the task to the same processor (i.e. if the processor may receive another task), otherwise, we allocate the task to the fastest processor that is not yet saturated (able to receive new tasks according to the load-balancing strategy).

In the previous version of the ILHA algorithm, we process all the ready tasks at each step. In some cases, it would be better to take into account the bottom level of the ready tasks and to consider first the tasks on a critical path. To this purpose, we sort the ready tasks according to their bottom level. Then, we introduce a parameter B, the maximal number of ready tasks that will be considered at each step. We consider those B tasks with the higher bottom levels and we allocate them using the load balancing algorithm. Then, we update the set of ready tasks (indeed some new tasks may have become ready) and we re-sort them according to their bottom level. Thus, we expect that the tasks on a critical path will be processed as soon as possible. Unfortunately, we face a tradeoff for choosing an appropriate value for B. On one hand if B is large, it will be possible to better balance the load and minimize the communication cost. On the other hand, a small value of B will enable us to process the tasks on the critical path sooner. Of course B must be at least equal to the number of processors, otherwise some processors would be kept idle. The choice of B will be discussed furthermore in the Section 5.

We obtain the final version of the $ILHA$ algorithm:

THE ILHA ALGORITHM

1: Compute the bottom level of each task

2: $ReadyTask \leftarrow \{$Entry tasks$\}$ sorted by decreasing value of their bottom level

3: While $ReadyTask$ is not empty

4:    Take the B first tasks of the $ReadyTask$

5:    Compute the optimal distribution with B tasks

6:    For each task t of $ReadyTask$

7:     If all predecessors of t are on p and p is free

8:      Assign t to p

9:    For each task t of $ReadyTask$ not yet assigned

10:    Assign t to the first free processor

11:    Update the list of $ReadyTask$ by inserting the new ready tasks in the sorted list

12:End while

The ILHA heuristic was compared [3] with five heuristics taken from the literature: the *minimum Partial Completion Time static priority* (PCT) heuristic [15], the *Best Imaginary Level* (BIL) heuristic [17], the

7

Critical Path on a Processor (CPOP) heuristic [26], the *Generalized Dynamic Level* (GDL) heuristic [21] and the previous HEFT heuristic. For the experimental comparisons, we have used six classical testbeds (LAPLACE, LU, STENCIL, FORK-JOIN, DOOLITTLE, and LDMt:see Section 5). All these comparisons showed that the best results are obtained for HEFT and ILHA. We now proceed to adapting these two heuristics to the one-port model.

4.3 HEFT for the one-port model

Modifying HEFT for the one-port model is not difficult. When the highest priority ready-task is selected, we still search for the processor that allows earliest completion time. But now we have to take constraints in communication resources. This means that in addition to scheduling the selected task we must also schedule eventual incoming communications. Since we have access to current communication schedules for all processors, we can assign the new communications as early as possible, in a greedy fashion.

Consider the following example with three processors P_1, P_2 and P_3. Assume that the selected task T has two incoming edges, one from a task T_1 already allocated to processor P_1 and the other from a task T_2 already allocated to processor P_2. If we try to allocate the selected task to P_1, we can neglect the first communication. We schedule the communication from P_2 to P_1 as soon as possible, with the one-port constraint: we look for the first available time-interval during which P_2 is not sending and P_1 is not receiving. This interval must start after the completion of the source task on P_2 and must be long enough so that the entire communication, of duration $comm(T_1, T, P_2, P_1) = data(T_1, T) \times link(P_2, P_1)$, can take place. In passing, note that the model can easily be extended to the case where the interconnection network is such that messages must be routed between some processor pairs: if there is no direct link from P_2 to P_1, we redo the previous step for all intermediate messages between adjacent processors. Having scheduled the communications, we can now look at the computation schedule of P_1 to find the earliest possible starting time for the execution of the selected task. We thus derive the completion time on P_1. We take the minimum completion time on P_1, P_2, and P_3 to decide which processor will execute the task.

4.4 ILHA for the one-port model

Modifying ILHA for the one-port model is more challenging. This is because several ready tasks are dealt with simultaneously, which leads to handle many more communications.

As before, we select B (independent) ready tasks of highest priority (i.e. of largest bottom level). Let W be the sum of the weights of these B tasks. To minimize the number of communications, we then proceed in two steps:

Step 1 We scan the list of B tasks (starting with highest priorities first), and check whether a given task T can be assigned without generating any communication , i.e. whether all the parents of T have already been allocated to the same processor, say P_i. In that case, we allocate T to processor P_i, provided that the current workload of P_i does not exceed the fraction $c_i W$ of the total work. Here c_i is the value returned by the load-balancing algorithm discussed above; if P_i happens to have already received its share of the work, we do nothing and proceed to the next task in the list, until the list is exhausted.

Step 2 At the end of Step 1, some tasks have been allocated to processors. We suppress them from the list of B ready tasks, which we scan a second time, in the same order. We use the same strategy as in HEFT to allocate the tasks: we select the processor that allows for the earliest completion time.

To exemplify the differences between HEFT and ILHA, consider the toy-example represented in Figure 3 and assume two same-speed processors P_0 and P_1 are available ($t_0 = t_1 = link(P_0, P_1) = 1$). The bottom level of all the children nodes is the same, so assume they are ranked in the order $a_1, a_2, a_3, ab_1, ab_2, b_3, b_2, b_1$. In the following, a task that would end at the same time-step on both processors is always assigned to P_0 (this is just an arbitrary way to break ties).

HEFT first schedules a_0 on P_0 and b_0 on P_1. Then a_1 is assigned to P_0. Next a_2 is assigned to P_0 again, because of the tie-breaking rule. After that a_3 goes to P_1, and so on: see Figure 4.

ILHA also start by scheduling a_0 on P_0 and b_0 on P_1. After that, if $B \geq 8$, ILHA benefits from its global view: it assigns no-communication tasks, i.e. a_1, a_2 and a_3 to P_0 and b_3, b_2 and b_1 to P_1. Note that
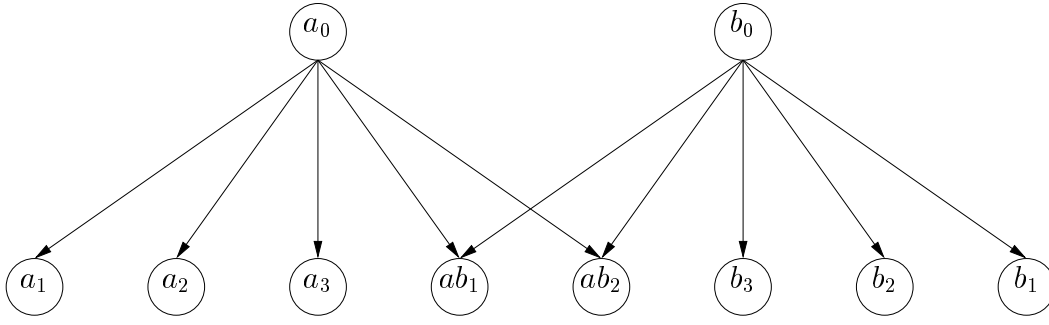
Figure 3: A toy example: all computation and communication costs are equal to 1.
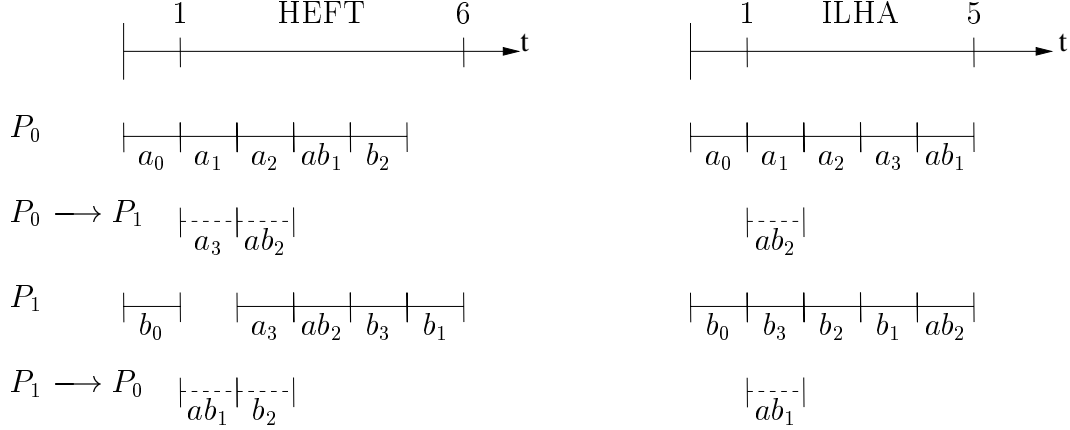

Figure 4: HEFT and ILHA scheduling for the toy example.

$c_1 = c_2 = 0.5$, hence each processor could receive up to 4 tasks in this allocation step. Next we turn to HEFT scheduling, as outlined in Figure 4. Note that the makespan is smaller, but also that the number of communication has dramatically been reduced. Reducing communications while achieving a good load balance is the objective that has guided the design of ILHA.

Note that several variations in the design of ILHA could be implemented. First, there is no reason to limit the scan of the B ready tasks to those tasks incurring no communication. We could add another scan for tasks that can be scheduled at the price of a single communication, and so on. Second, and more importantly, we could limit the use of HEFT at Step 2 to a pre-allocation of tasks to processors, and re-schedule all communications in a third step. In other words, after Step 2 all tasks have been allocated to processors. We can forget about the schedule times computed during Step 1 or during Step 2 (using HEFT) and keep only the allocation function. We then try to re-schedule the whole set of B tasks: indeed, the scheduling has been made simpler, because the allocation is known. Unfortunately, this scheduling problem remains NP-complete (see the proof in the Appendix). Still, we could use greedy-like heuristics to improve the scheduling after the allocation resulting from the two scans.

5 Simulation results

5.1 Testbeds

In order to compare the different algorithms, we consider six classical kernels representing various types of parallel algorithms. The selected task graphs are:

- LU: *LU decomposition*

- LAPLACE: *Laplace equation solver*

- STENCIL: *stencil algorithm*

- FORK-JOIN: *fork-join graph*

- DOOLITTLE: *Doolittle reduction*

- LDMt: LDM^t *decomposition*

Miniature versions of each task graph are shown in Figure 5.

5.2   Weights and speeds

Task weights   For the LAPLACE, STENCIL, and FORK-JOIN testbeds, all tasks have same weight, which we normalize to 1. For the linear algebra testbeds, i.e. LU, DOOLITTLE and LDMt, the situation is more complicated, because the amount of work to be done at each step of the algorithm is not constant (see [11, 19]). For the LU kernel, the weight of a task at level k is $N - k$, where N is the size of the graph. For the DOOLITTLE and LDMt kernels, the the weight of a task at level k is k, where k varies from 1 to N, the size of the graph.

Processor speeds   We use 10 processors: five processors with cycle time 6, three processors with cycle time 10, and two processors with cycle time 15. Remember that the time to execute a task is the product of its weight by the processor cycle-time. The speedup that can be achieved with these 10 processors is bounded as follows:

With this set of processors, the smallest value to perfectly load-balance the work is $B = 38$. Indeed we give 5 tasks to each processor of cycle time 6 (hence 30 tasks), 3 tasks to each processor of cycle time 10 (hence 9 tasks) and finally 2 tasks to each processor of cycle time 15 (hence 4 tasks). So in 30 time-units we process $25 + 9 + 4 = 38$ tasks. To compute these 38 tasks in a sequential way, using one of the fastest processors, we would need $38 \times 6 = 228$ time-units. So we may improve the sequential time by a factor at most $\frac{228}{30} = 7.6$. Note that this is only an upper bound, since all communication costs are neglected here, and since it is assumed that no dependence would keep any processor idle at any time-step.

Communication costs   For each testbed, we let the communication costs be proportional to the task weights: indeed in each kernel, we always communicate the data that has just been updated. In other words, the communication cost from a task v to a task v' is equal to c times the weight of v, where c is a parameter that models the communication-to-computation ratio of the target platform. Because we want to stress the impact of communication costs, we use a large value for c: we let $c = 10$, which is rather representative of workstations linked with a slow (Ethernet) network.

5.3   Results

We start with the FORK-JOIN kernel: see Figure 6. We see that HEFT and ILHA lead to the same scheduling. The value of B has no impact on ILHA in this case (we used $B = 38$ in the experiments). The speedup is quite limited: the gain is 1.58, to be compared with the theoretical bound of 7.6. But in fact, the scheduling found by both heuristics is efficient. Indeed, to reach a speedup factor s with homogeneous processors, we would have to generate $\frac{s-1}{s} \times N$ communications, where N is the number of intermediate nodes in the graph. Even if we succeeded in overlapping all these communications with computations, we would not do better than $c \times \frac{s-1}{s} \times N + 3 \times wt$ time-steps, where w is the task weight, t the processor cycle-time and c the communication cost. The sequential time is equal to $(N + 2) \times wt$, hence the speedup is $s \approx \frac{wt}{cs-1}$ (with N large enough). This leads to $s \leq \frac{wt}{c} + 1$. Here with $t = 6$, $c = 10$ et $w = 1$, the bound is 1.6: contrarily to the appearances, 1.58 turns out to be a very good result!

We continue with the LU decomposition kernel: see Figure 7. Here the best value for B has been experimentally found to be $B = 4$. This small value can be explained as follows: the shape of the LU task graph is such that the critical path must be executed rapidly, hence the need for a smaller value of B. We
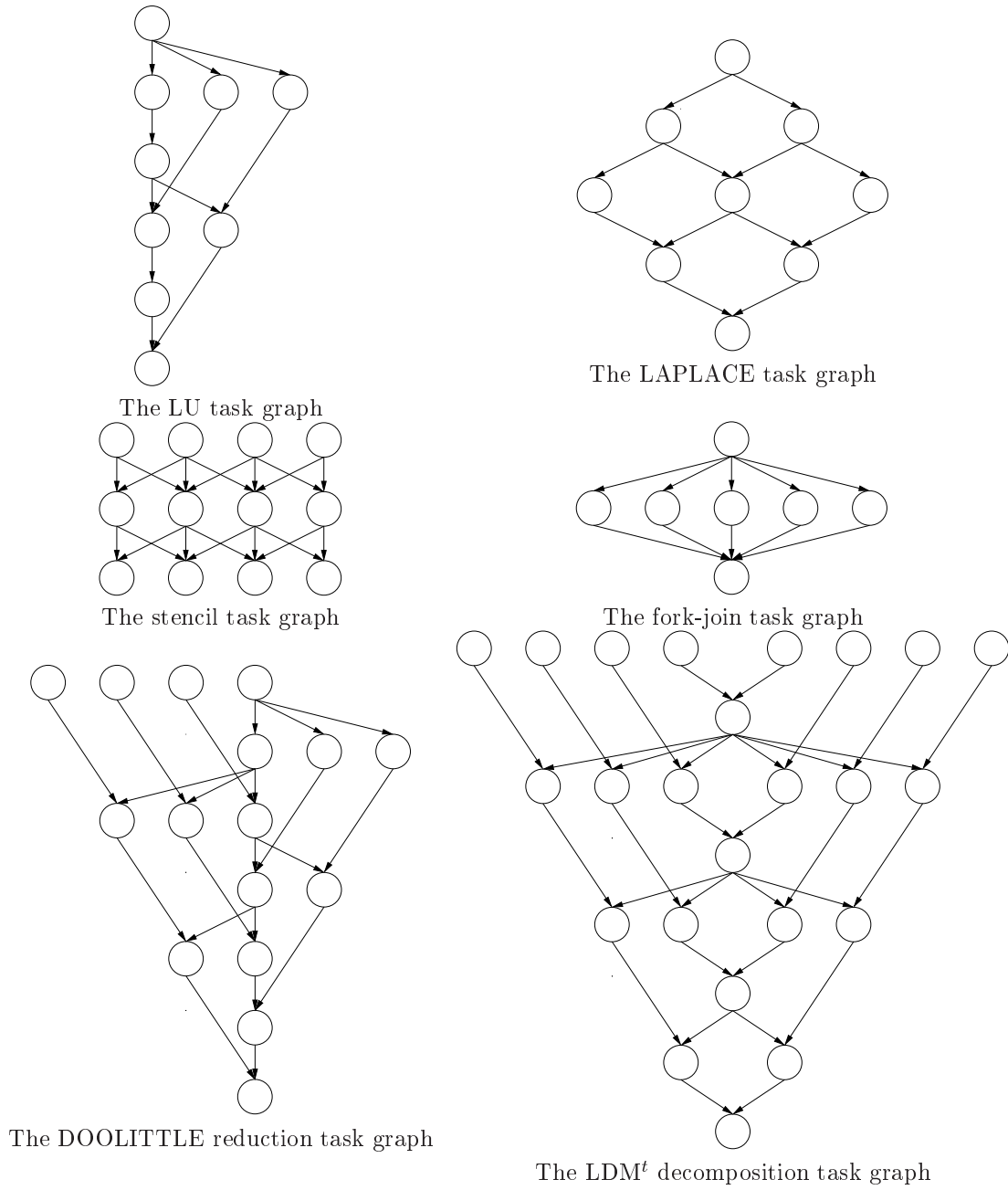
The LU task graph

The LAPLACE task graph

The stencil task graph

The fork-join task graph

The DOOLITTLE reduction task graph

The LDM^t decomposition task graph
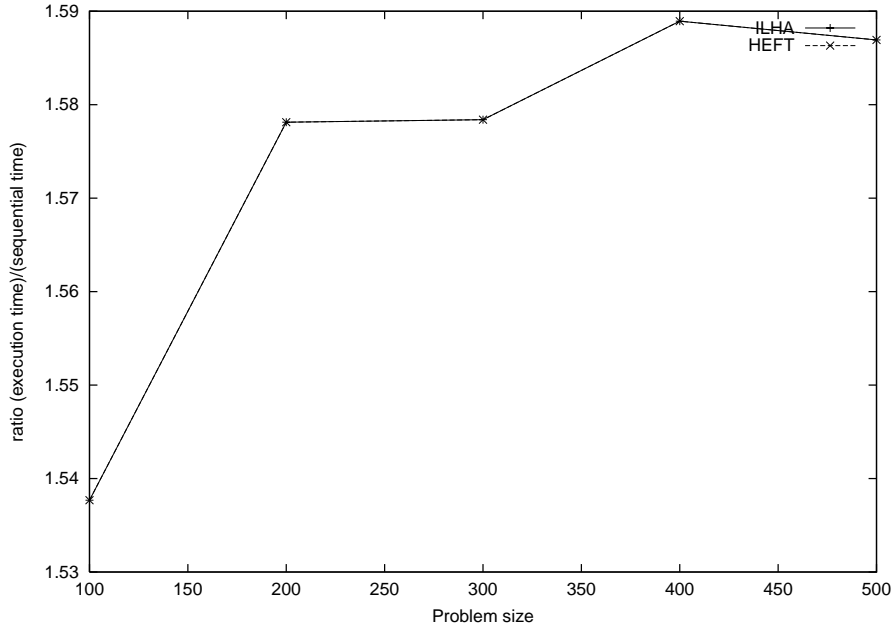
Figure 5: The different testbeds.

11

Figure 6: Comparison of HEFT and ILHA for the FORK-JOIN problem, with 10 processors and a communication cost equal to 10.

point out that HEFT and ILHA achieve similar performances for $n = 100$, but ILHA gains more and more as the problem size increases. For $n = 500$ ILHA obtains a speedup equal to 5, while HEFT is limited to 4.5.

Results are similar for the LAPLACE, LDM^t and DOOLITTLE kernels (see Figures 8, 9 and 10). For each of them, ILHA roughly gains 10% over à HEFT. For LAPLACE we used $B = 38$: all nodes are on a critical path, and a larger value of B allows both to load-balance computations and to minimize communications. For $n = 500$, ILHA achieves a speedup equal to 5.6. For DOOLITTLE and LDM^t, the best value for B is $B = 20$, a tradeoff between a good load-balancing and an early processing of the critical path. The speedup for LDM^t is 4.9, and for DOOLITTLE, it is equal to 4.4. The gain over HEFT is significant.

Finally for the STENCIL kernel (see Figure 11), we observe a new phenomenon: for both heuristics, the speedup decreases as the problem size increases. This can be explained as follows: as the graph becomes larger, we have to use all processors in parallel on each row of the graph, and this induces many communications to be done sequentially, and these become the bottleneck. ILHA obtains a low speedup equal to 2.7, slightly better than HEFT which reaches 2.4. The optimal value for B is $B = 38$.

We point out that the best results for ILHA have been obtained by trying several values for B. Unfortunately, we have not found any systematic technique to predict the optimal value of B. Note however that the range of B is limited: with equal-size tasks and p processors of cycle-times $t_1, t_2, \ldots, t_p$, we can sample the interval $[1..M]$, where the value $M = \text{lcm}(t_1, t_2, \ldots, t_p) \sum_{i=1}^{p} \frac{1}{t_i}$ ensures a perfect load balancing.

6 Conclusion

In this paper we have argued that the (bi-directional) one-port scheduling model was more realistic than the macro-dataflow model to design and analyse the execution of parallel algorithms onto networks of workstations. Indeed, the scarcity of communication resources is fully taken into account, just as the scarcity of commputing resources was dealt with in the macro-dataflow model with a limited number of processors.

We have assessed the intrinsic complexity of task graph scheduling under the one-port model. The NP-completeness result obtained for fork graphs is no surprise at all, but motivates the design of efficient heuristics. We have shown how to extend the HEFT heuristic [26] to cope with the new model. The HEFT
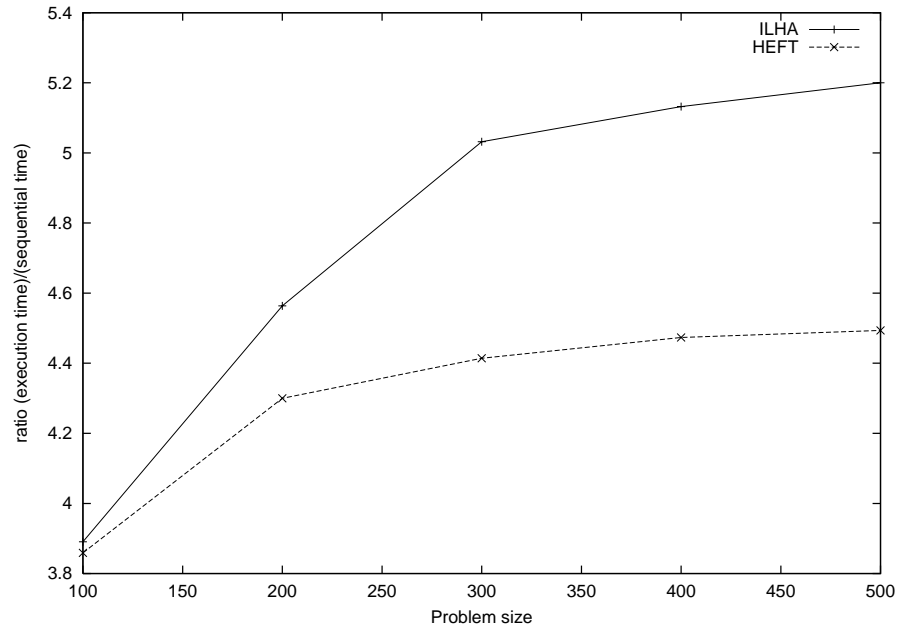
12

Figure 7: Comparison of HEFT and ILHA for the LU problem, with 10 processors and a communication cost equal to 10.
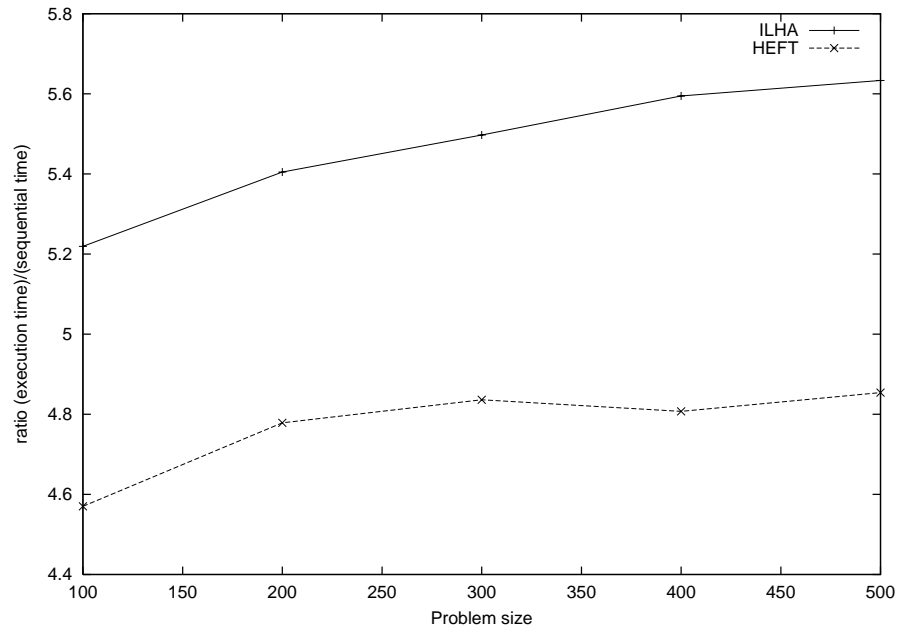

Figure 8: Comparison of HEFT and ILHA for the LAPLACE problem, with 10 processors and a communication cost equal to 10.
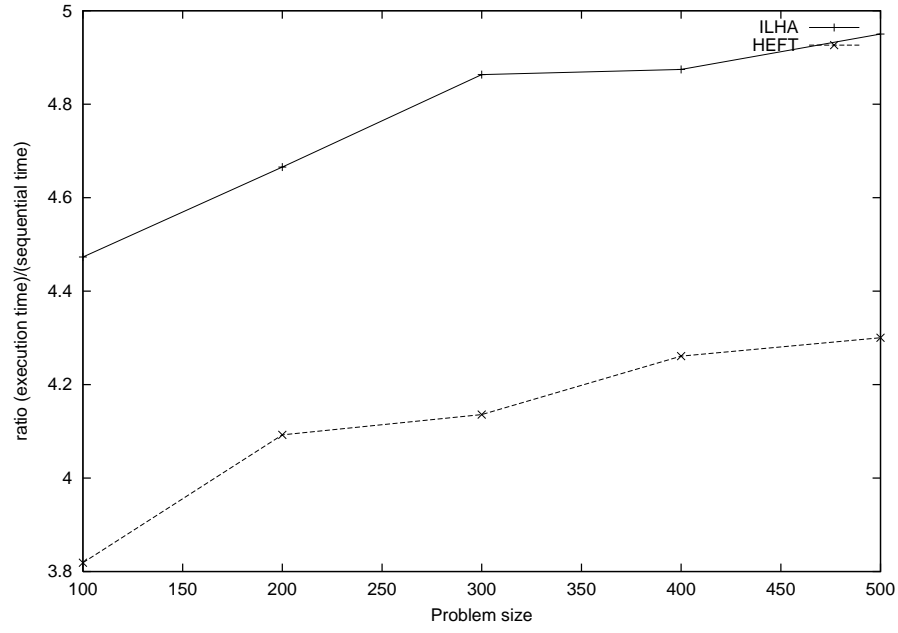
13

Figure 9: Comparison of HEFT and ILHA for the LDM^t problem, with 10 processors and a communication cost equal to 10.
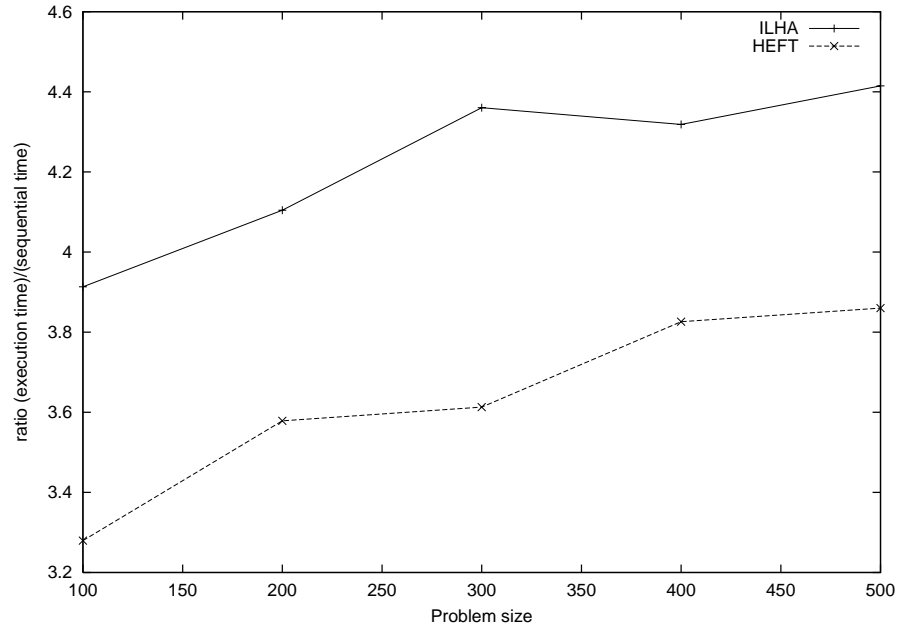

Figure 10: Comparison of HEFT and ILHA for the DOOLITTLE problem, with 10 processors and a communication cost equal to 10.
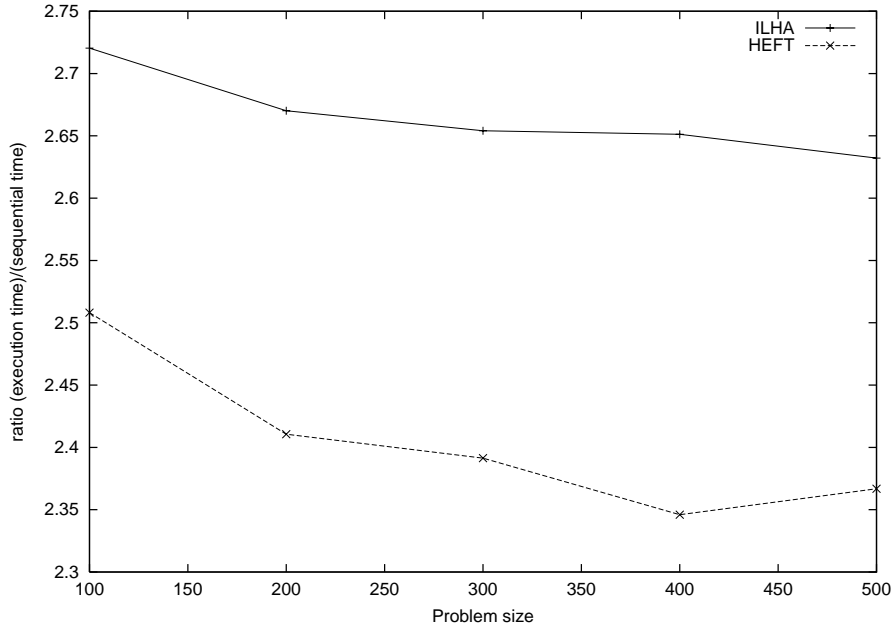
Figure 11: Comparison of HEFT and ILHA for the STENCIL problem, with 10 processors and a communication cost equal to 10.

heuristic was already an extension of critical path scheduling to heterogeneous computing resources, and we showed how to serialize communications in accordance to the one-port constraint.

We have also introduced a new heuristic, ILHA, whose design is motivated by (i) the search for a better load-balance and (ii) the generation of fewer communications. These goals are achieved by scheduling a chunk of ready tasks simultaneously, which enables for a global view of the potential communications. Preliminary results conducted on six classical testbeds demonstrate very promising results. Still, there is room for further analysis and improvements of the ILHA heuristic, as well as more extensive experimental validation and comparisons.

7 Appendix

In this section, we prove that scheduling the communications in a bipartite graph, after having allocated the tasks to the processors, is a NP-complete problem. The link with ILHA is obvious: one subset of the nodes represent the B ready tasks which are currently under examination, and the other subset represents their parents: communication links are directed from the latter to the former.

Definition 2 *COMM-SCHED(G,P,T): Given a bipartite graph G of task nodes $V = V_1 \cup V_2$, a finite set P of same-speed processors connected through a fully homogeneous network, such that each task node is assigned a processor number in P, and edges from V_1 to V_2 are assigned a communication cost, and given a time-bound T, is there a valid schedule σ whose makespan is not greater than T?*

Theorem 2 *The COMM-SCHED(G,P,T) decision problem is NP-complete.*

Proof As for the FORK-SCHED problem, we use a reduction from 2-PARTITION. We start with an arbitrary instance of 2-PARTITION, i.e. a set $\mathcal{A} = \{a_1, \ldots, a_n\}$ of n integers. We have to polynomially transform this instance into an instance of the COMM-SCHED problem which has a solution iff the original instance of 2-PARTITION has a solution, i.e. iff there exists a partition of $\{1, \ldots, n\}$ into two subsets $\mathcal{A}_1$
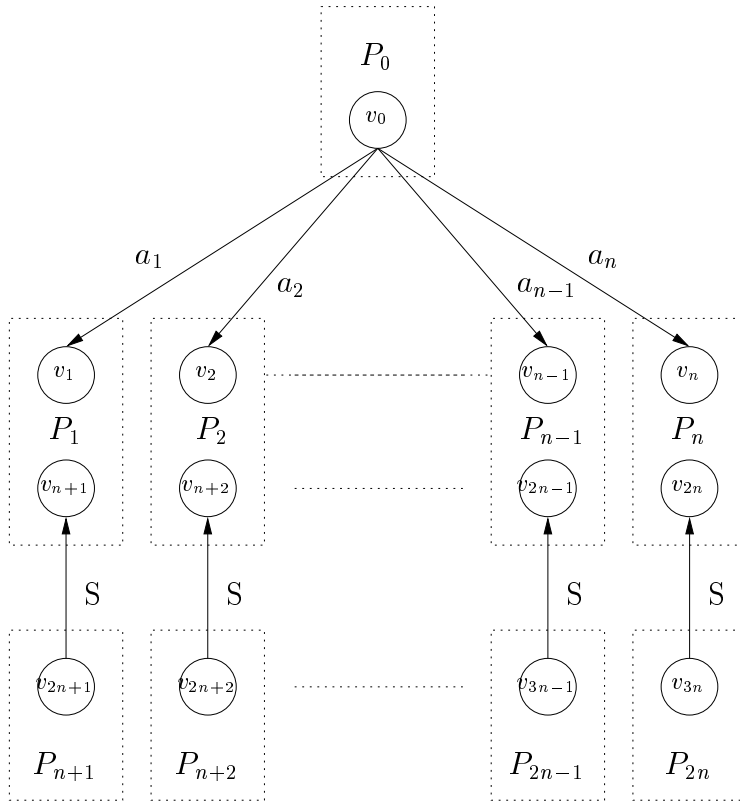
Figure 12: The instance of COMM-SCHED.

and $\mathcal{A}_2$ such that

$$\sum_{i \in \mathcal{A}_1} a_i = \sum_{i \in \mathcal{A}_2} a_i$$

We let $2S = \sum_{i=1}^{n} a_i$ (if the sum is odd, there is no solution to the instance of 2-PARTITION). We construct the following instance of COMM-SCHED:

- there are $3n+1$ tasks: a fork-graph with parent v_0 and children $v_1, v_2, \ldots, v_n$, and n separated pairs of tasks $(v_{2n+1}, v_{n+1}), (v_{2n+2}, v_{n+2}), \ldots, (v_{3n}, v_{2n})$; each pair has an edge $v_{2n+i} \longrightarrow v_{n+i}$: see Figure 12).

- there are $2n+1$ processors $P_0, P_1, \ldots, P_n$ of same speed: $t_i = 1, 0 \le i \le n$

- task v_0 is assigned to P_0 and for $1 \le i \le n$, tasks v_i and v_{n+1} are assigned to processor P_i

- for $1 \le i \le n$, task v_{2n+i} is assigned to processor P_{n+i}

- all computation times are equal to zero: $w(v_i) = 0, 0 \le i \le 3n$

- communication times: for $1 \le i \le n$, $d_i = data(v_0, v_i) = a_i$ and $data(v_{2n+i}, v_{n+i}) = S$.

- homogeneous network: $link(P_i, P_j) = 1$ if $i \ne j$

- time bound: $T = S$

Clearly, the size of the constructed instance of COMM-SCHED is polynomial (even linear) in the size of the original instance of 2-PARTITION.

Assume that the original instance of 2-PARTITION admits a solution: let $\mathcal{A}_1$ and $\mathcal{A}_2$ be a partition of $\{1, \ldots, n\}$ such that $\sum_{i \in \mathcal{A}_1} a_i = \sum_{i \in \mathcal{A}_2} a_i = S$. We derive a scheduling for the instance of COMM-SCHED as follows:

16

- At time-step $t = 0$, processor P_{n+i} sends its message to processor P_i, $1 \leq i \leq n$

- Processor P_0 sends messages to nodes v_i such that $i \in \mathcal{A}_1$ (in any order); then, at time-step S, it sends messages to sends messages to nodes v_i such that $i \in \mathcal{A}_2$ (in any order)

- If $i \in \mathcal{A}_1$, processor P_i first executes v_i, as soon as it has received the data from P_0. Then, no later than at time-step $S = \sum_{i \in \mathcal{A}_2} d_i$, it executes task v_{n+i}

- If $i \in \mathcal{A}_2$, processor P_i first executes task v_{n+i}; then it executes task v_i, as soon as it has received the data from P_0

Therefore, we have derived a valid scheduling that matches the time-bound, hence a solution to the COMM-SCHED instance.

Reciprocally, assume that the COMM-SCHED instance admits a solution, i.e. a valid scheduling σ that matches the time-bound T. Then P_0 sends all its n messages without any idle-time. If at time-step S processor P_0 is in the middle of an emission, say that of message number j, then processor P_j has not enough time to receive data from P_{n+j} for task v_{n+i}, either before or after receiving the message from P_0. Hence at time-step S P_0 is just completing the sending of one message, hence the solution to 2-PARTITION.

For the reader that would be worried with execution times equal to 0, it is not difficult to modify the construction to derive an instance with execution times equal to 1 (but this slightly complicates the proof, hence our choice). ■

References

[1] B.A.Shirazi, A.R. Hurson, and K.M. Kavi. *Scheduling and load balancing in parallel and distributed systems*. IEEE Computer Science Press, 1995.

[2] Vincent Boudet, Fabrice Rastello, and Yves Robert. A proposal for a heterogeneous cluster ScaLAPACK (dense linear solvers). In Hamid R. Arabnia, editor, *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*. CSREA Press, 1999. Extended version available as LIP Technical Report RR-99-17.

[3] Vincent Boudet and Yves Robert. Scheduling heuristics for heterogeneous processors. In *2001 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2001)*, pages 2109–2115. CSREA Press, 2001. Extended version available (on the Web) as Technical Report 2001-22, LIP, ENS Lyon.

[4] P. Chrétienne, E.G. Coffman Jr., J.K. Lenstra, and Z. Liu, editors. *Scheduling Theory and its Applications*. John Wiley and Sons, 1995.

[5] D. E. Culler and J. P. Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, San Francisco, CA, 1999.

[6] A. Darte, Y. Robert, and F.Vivien. *Scheduling and Automatic Parallelization*. Birkhaüser, 2000.

[7] H. El-Rewini, H.H. Ali, and T.G. Lewis. Task scheduling in multiprocessing systems. *Computer*, 28(12):27–37, 1995.

[8] Michael R. Garey and Davis S. Johnson. *Computers and Intractability, a Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1991.

[9] Apostolos Gerasoulis and Tao Yang. A comparison of clustering heuristics for scheduling DAGs on multiprocessors. *J. Parallel and Distributed computing*, 16(4):276–291, December 1992.

[10] Apostolos Gerasoulis and Tao Yang. On the granularity and clustering of directed acyclic task graphs. *IEEE Trans. Parallel and Distributed Systems*, 4(6):686–701, 1993.

[11] Gene H. Golub and Charles F. Van Loan. *Matrix computations*. Johns Hopkins, 2 edition, 1989.

[12] L. Hollermann, T.S. Hsu, D.R. Lopez, and K. Vertanen. Scheduling problems in a practial allocation model. *J. Combinatorial Optimization*, 1(2):129–149, 1997.

[13] T.S. Hsu, J. C. Lee, D.R. Lopez, and W.A. Royce. Task allocation on a network of processors. *IEEE Trans. Computers*, 49(12):1339–1353, 2000.

[14] K. Hwang and Z. Xu. *Scalable Parallel Computing*. McGraw-Hill, 1998.

[15] M. Maheswaran and H. J. Siegel. A dynamic matching and scheduling algorithm for heterogeneous computing systems. In *Seventh Heterogeneous Computing Workshop*. IEEE Computer Society Press, 1998.

[16] M.G. Norman and P. Thanisch. Models of machines and computation for mapping in multicomputers. *ACM Computing Surveys*, 25(3):103–117, 1993.

[17] Hyunok Oh and Soonhoi Ha. A static scheduling heuristic for heterogeneous processors. In *Proceedings of Europar'96*, volume 1123 of *LNCS*, Lyon, France, August 1996. Springer Verlag.

[18] J.M. Orduna, F. Silla, and J. Duato. A new task mapping technique for communication-aware scheduling strategies. In T.M. Pinkston, editor, *Workshop for Scheduling and Resource Management for Cluster Computing (ICPP'01)*, pages 349–354. IEEE Computer Society, 2001.

[19] Yves Robert, Michel Cosnard, Mounir Marrakchi, and Denis Trystram. Parallel Gaussian elimination on a MIMD computer. *Parallel Computing*, 6:275–296, 1988.

[20] C. Roig, A. Ripoll, M.A. Senar, F. Guirado, and E. Luque. Improving static scheduling using inter-task concurrency measures. In T.M. Pinkston, editor, *Workshop for Scheduling and Resource Management for Cluster Computing (ICPP'01)*, pages 375–381. IEEE Computer Society, 2001.

[21] G.C. Sih and E.A. Lee. A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. *IEEE Transactions on Parallel and Distributed Systems*, 4(2):175–187, 1993.

[22] O. Sinnen and L. Sousa. Comparison of contention-aware list scheduling heuristics for cluster computing. In T.M. Pinkston, editor, *Workshop for Scheduling and Resource Management for Cluster Computing (ICPP'01)*, pages 382–387. IEEE Computer Society, 2001.

[23] O. Sinnen and L. Sousa. Exploiting unused time-slots in list scheduling considering communication contention. In R. Sakellariou, J. Keane, J. Gurd, and L. Freeman, editors, *EuroPar'2001 Parallel Processing*, pages 166–170. Springer-Verlag LNCS 2150, 2001.

[24] Oliver Sinnen and Leonel Sousa. Scheduling task graphs on arbitrary processor architectures considering contention. In *High Performance Computing and Networking*, pages 373–382. Springer-Verlag LNCS 2110, 2001.

[25] M. Tan, H.J. Siegel, J.K. Antonio, and Y.A. Li. Minimizing the aplication execution time through scheduling of subtasks and communication traffic in a heterogeneous computing system. *IEEE Transactions on Parallel and Distributed Systems*, 8(4):857–1871, 1997.

[26] H. Topcuoglu, S. Hariri, and M.-Y. Wu. Task scheduling algorithms for heterogeneous processors. In *Eighth Heterogeneous Computing Workshop*. IEEE Computer Society Press, 1999.

[27] Tao Yang and Apostolos Gerasoulis. DSC: Scheduling parallel tasks on an unbounded number of processors. *IEEE Trans. Parallel and Distributed Systems*, 5(9):951–967, 1994.