A DAG partitioning-assisted list-based scheduler for homogeneous processors

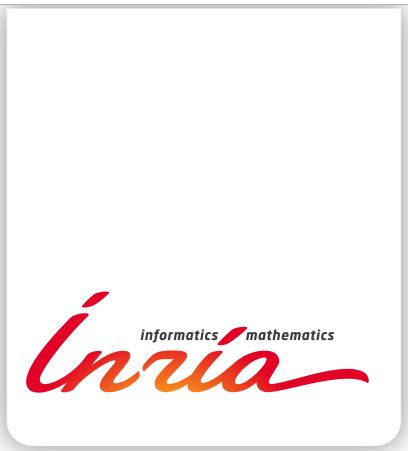
Julien Herrmann, Anne Benoit, Bora Uçar, Umit V. Catalyurek

A DAG partitioning-assisted list-based scheduler for homogeneous processors

Julien Herrmann, Anne Benoit, Bora Uçar, Ümit V. Çatalyürek

RESEARCH REPORT
N° 9185
June 2018
Project-Team ROMA

ISSN 0249-6399 ISRN INRIA/RR--9185--FR+ENG



A DAG partitioning-assisted list-based scheduler for homogeneous processors

Julien Herrmann*, Anne Benoit†, Bora Uçar‡, Ümit V. Çatalyürek*

Project-Team ROMA

Research Report n° 9185 — June 2018 — 20 pages

Abstract: When scheduling a directed acyclic graph (DAG) of tasks on computational platforms, a good trade-off between load balance and data locality is necessary. List-based scheduling techniques, such as the earliest finish time (EFT) heuristic, are commonly used greedy approaches for this problem. The downside of EFT, and other list-scheduling heuristics, is that they are incapable of making short-term sacrifices for the global efficiency of the schedule. In this work, we describe three new list-based scheduling heuristics based on clustering for homogeneous platforms. Our approach uses an acyclic partitioner for DAGs for clustering. The clustering enhances the data locality of the scheduler with a global view of the graph. Furthermore, since the partition is acyclic, we can schedule each part completely once its input tasks are ready to be executed. We present an extensive experimental evaluation showing the trade-offs between the granularity of clustering and the parallelism, and how this affects the scheduling.

Key-words: partitioning, directed acyclic graphs, data locality, concurrency.

* School of CSE, Georgia Institute of Technology, Atlanta, Georgia 30332–0250.
† ENS Lyon and LIP (UMR5668 Université de Lyon - CNRS - ENS Lyon - Inria - UCBL 1), 46, allée d'Italie, ENS Lyon, Lyon F-69364, France.
‡ CNRS and LIP (UMR5668 Université de Lyon - CNRS - ENS Lyon - Inria - UCBL 1), 46, allée d'Italie, ENS Lyon, Lyon F-69364, France.

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Un ordonnanceur de liste basé sur le partitionnement de DAGs pour des processeurs homogènes

Résumé : Lors de l'ordonnancement d'un graphe dirigé acyclique (DAG) de tâches sur une plate-forme, un bon compromis entre équilibrage de charge et localité des données est nécessaire. Les techniques d'ordonnancement de liste, comme par exemple l'heuristique EFT (earliest finish time, temps de complétion au plus tôt), sont des approches gloutonnes communément utilisées pour ce problème. Les inconvénients d'EFT, et d'autres heuristiques d'ordonnancement de liste, sont qu'elles sont incapables de faire des sacrifices à court terme pour que l'ordonnancement global soit plus efficace. Dans ces travaux, nous décrivons trois nouvelles heuristiques d'ordonnancement de liste pour des plates-formes homogènes. Notre approche se base sur un partitionnement acyclique du DAG, car les parties ainsi formées permettent d'avoir une bonne localité des données tout en conservant une vue générale du graphe. De plus, étant donné que la partition est acyclique, nous pouvons ordonnancer chaque partie entièrement une fois que ses tâches d'entrée sont prêtes à être exécutées. Nous présentons une évaluation expérimentale des algorithmes pour montrer les compromis entre la granularité des partitions et le parallélisme, et comment cela affecte l'ordonnancement.

Mots-clés : partitionnement, graphes dirigés acycliques, localité des données, concurrence.

1 Introduction

Scheduling is one of the most studied areas of computer science. A large body of research deals with scheduling applications/workflows modeled as Directed Acyclic Graphs (DAGs), where vertices represent atomic tasks, and edges represent dependencies [11]. Among others, *list-based scheduling* techniques are the most widely studied and used techniques, mainly due to the ease of implementation and explanation of the progression of the heuristics [1, 8, 13, 14, 16, 17, 19]. In list-based scheduling techniques, tasks are ordered based on some predetermined priority, and then are mapped and scheduled onto processors. Another widely used approach is clustering-based scheduling [9, 10, 15, 19, 20], where tasks are grouped into clusters and then scheduled onto processors.

Almost all of the existing clustering-based scheduling techniques are based on bottom-up clustering approaches, where clusters are constructively built from the composition of atomic tasks and existing clusters. We argue that such decisions are local, and hence cannot take into account the global structure of the graph. Recently, we have developed one of the first multi-level acyclic DAG partitioner [7]. The partitioner itself also uses bottom-up clustering in its *coarsening* phase. However, it uses multiple levels of coarsening, and then it *partitions* the graph into two parts by minimizing the *edge cut* between the two parts. Then, in an *uncoarsening* phase, it refines the partitioning while it projects the solution found in the coarsened graph to finer graphs, until it reaches to the original graph. This process can be iterated multiple times, using a constraint coarsening (where only vertices that were assigned to same part can be clustered), in order to further improve the partitioning. We hypothesize that clusters found using such a DAG partitioner are much more successful in putting together the tasks with complex dependencies, and hence in minimizing the overall inter-processor communication.

In this work, we use the realistic duplex single-port communication model, where at any point in time, each processor can, in parallel, execute a task, send one data, and receive another data. Because concurrent communications are limited within a processor, minimizing the communication volume is crucial to minimize the total execution time, or *makespan*.

We propose three DAG partitioning-assisted list-based scheduling heuristics for homogeneous platforms, aiming at minimizing the makespan when the DAG is executed on a parallel platform. In our proposed schedulers, when scheduling to a system with p processing units (or processors), the original task graph is first partitioned into K parts (clusters), where $K \geq p$. Then, a list-based scheduler is used to assign tasks (not the clusters). Our scheduler hence uses list-based scheduler, but with one major constraint: all the tasks of a cluster will be executed by same processor. This is not the same as scheduling the graph of clusters, as the decision to schedule a task can be made before scheduling all tasks in a predecessor cluster. Our intuition is that thanks to the partition that is done beforehand, the scheduler "sees" the global structure of the graph, and it uses this to "guide" the scheduling decisions. Since all the tasks in a cluster will be executed on the same processor, the execution time for the cluster can be approximated by simply the sum of the individual task's weights (actual execution time can be larger due to dependencies to tasks that might be assigned to other processors). Here, we heuristically decide that having balanced clusters helps the scheduler to achieve load-balanced execution. The choice of the number of parts K is a trade-off between data locality vs. concurrency. Large K values may yield higher concurrency, but would potentially incur more inter-processor communication. At the extreme, each task is a cluster, where we have the maximum potential concurrency. However, in this case, one has to rely on list-based scheduler's

local decisions to improve data-locality and hence reduce inter-processor communication.

Our main contribution is to develop three different variants of partitioning-assisted list-based scheduler, and to experimentally evaluate them against a baseline list-based scheduler, Earliest Finish Time (EFT), following the duplex single-port communication model. We show significant savings in terms of makespan, in particular when the communication-to-computation ratio (CCR) is large, i.e., when communications matter a lot, hence demonstrating the need for a partitioning-assisted scheduling technique.

The rest of the paper is organized as follows. First, we discuss related work in Section 2. Next, we introduce the model and formalize the optimization problem in Section 3. The proposed scheduling heuristics are described in Section 4, and they are evaluated through extensive simulations in Section 5. Finally, we conclude and give directions for future work in Section 6.

2 Related work

Task graph scheduling has been the subject of a wide literature, ranging from theoretical studies to practical ones. An excellent survey and taxonomy of task scheduling methods can be found in [11]. Moreover, some benchmarking techniques to compare these methods are discussed in [12].

DAG scheduling heuristics can be divided into two with respect to whether they allow task duplication or not [2]. Those that allow task duplication do so to avoid communication. The focus of this work is non-duplication based scheduling.

There are two main approaches taken by the non-duplication based heuristics: list scheduling and cluster-based scheduling. A recent comparative study [18] gives a catalog of list-scheduling and cluster-scheduling heuristics and compares their performance.

In the list-based scheduling approach [1, 8, 13, 14, 16, 17, 19], each task in the DAG is first assigned a priority. Then, the tasks are sorted in descending order of priorities, hence resulting in a priority list. Finally, the tasks are scheduled in topological order, with highest priorities first. The list-scheduling based heuristics have low complexity and are easy to implement and understand.

In the cluster-based scheduling approach [9, 10, 15, 18–20], the tasks are first divided into clusters, each to be scheduled on the same processor. The clusters usually consist of highly communicating tasks. Then, the clusters are scheduled onto an unlimited number of processors, which are finally combined to yield the available number of processors.

Our approach is close to cluster-based scheduling in the sense that we first cluster tasks into a number larger than the number of available processors. At this step, we enforce somewhat balanced clusters. In the next step, we schedule tasks as in the list-scheduling approach, not the clusters, since there is a degree of freedom in scheduling a task of a cluster. In other words, our approach can also be conceived as a hybrid list and cluster scheduling, where the decisions of the list-scheduling part are constrained by the cluster-scheduling decisions.

Another important characteristic of scheduling heuristics is the time at which the scheduling decisions are made [11]. If the structure of the parallel application is known a priori, scheduling can be done at compile time. This type of scheduling is called static scheduling. On the other hand, in dynamic scheduling, decisions are made during run time by using the up-to-date information about the application. We focus in this work on a static scheduling approach.

Finally, note that we consider homogeneous computing platforms, where the processing units are identical and communicate through a homogeneous network. Task graphs and scheduling approaches can also be used to model and execute workflows on grids and heterogeneous plat-

forms [4, 6], with HEFT [17], being one of the most common approach. Assessing the performance of our new scheduling strategies on heterogeneous platforms will be considered in future work.

3 Model

Let $G = (V, E)$ be a directed acyclic graph (DAG), where the vertices in the set V represent tasks, and the edges in the set E represent the precedence constraints between those tasks. Let $n = |V|$ be the total number of tasks. We use $\texttt{Pred}[v_i] = \{v_j \mid (v_j, v_i) \in E\}$ to represent the (immediate) predecessors of a vertex $v_i \in V$, and $\texttt{Succ}[v_i] = \{v_j \mid (v_i, v_j) \in E\}$ to represent the (immediate) successors of v_i in G. The immediate predecessors and successors of a vertex are called its neighbors, and are denoted with the set $\texttt{Neigh}[v_i] = \texttt{Pred}[v_i] \cup \texttt{Succ}[v_i]$. Every vertex $v_i \in V$ has a weight, denoted by w_i, and every edge $(v_i, v_j) \in E$ has a cost, denoted by $c_{i,j}$.

The computing platform is a homogeneous cluster consisting of p identical processing units, called *processors*, and denoted $P_1, \ldots, P_p$, communicating through a fully-connected homogenous network. Each task needs to be scheduled onto a processor respecting the precedence constraints, and tasks are non-preemptive and atomic: a processor executes a single task at a time. For a given mapping of the tasks onto the computing platform, let $\mu(i)$ be the index of the processor on which task v_i is mapped, i.e., v_i is executed on the processor $P_{\mu(i)}$. For every vertex $v_i \in V$, its weight w_i represents the time required to execute the task v_i on any processor. Furthermore, if there is a precedence constraint between two tasks mapped onto two different processors, i.e., $(v_i, v_j) \in E$ and $\mu(i) \neq \mu(j)$, then some data must be sent from $P_{\mu(i)}$ to $P_{\mu(j)}$, and this takes a time represented by the edge cost $c_{i,j}$.

We enforce the realistic duplex single-port communication model, where at any point in time, each processor can, in parallel, execute a task, send one data, and receive another data. Consider the DAG example in Figure 1, where all execution times are unitary, and communication times are depicted on the edges. The computing platform in the example of Figure 1 has two identical processors. There is no communication to pay when two tasks are executed on the same processor, since the output can be directly accessed in the processor memory by the next task. For the proposed schedule, note that P_1 is already performing a *send* operation when v_5 would like to initiate a communication, and hence this communication is delayed by 0.5 time unit, since it can start only once P_1 has completed the previous send from v_1 to v_2. However, P_1 can receive data from v_2 to v_3 in parallel to sending data from v_5 to v_6. In this example, the total execution time, or *makespan*, is 6.

Formally, a schedule of graph G consists of an assignment of tasks to processors (already defined as $\mu(i)$, for $1 \leq i \leq n$), and a start time for each task, $\texttt{st}(i)$, for $1 \leq i \leq n$. Furthermore, for each precedence constraint $(v_i, v_j) \in E$ such that $\mu(i) \neq \mu(j)$, we must specify the start time of the communication, $\texttt{com}(i, j)$. Several constraints must be met to have a valid schedule, in particular with respect to communications:

- (atomicity) For each processor P_k, for all tasks v_i such that $\mu(i) = k$, the intervals $[\texttt{st}(i), \texttt{st}(i) + w_i[$ are disjoint.
- (precedence constraints, same processor) For each $(v_i, v_j) \in E$ with $\mu(i) = \mu(j)$, $\texttt{st}(i) + w_i \leq \texttt{st}(j)$.
- (precedence constraints, different processors) For each $(v_i, v_j) \in E$ with $\mu(i) \neq \mu(j)$, $\texttt{st}(i) + w_i \leq \texttt{com}(i, j)$ and $\texttt{com}(i, j) + c_{i,j} \leq \texttt{st}(j)$.

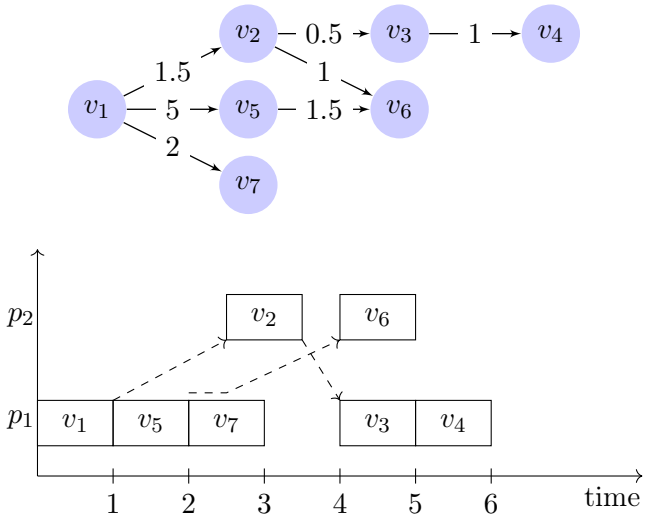


Figure 1 – Example of a small DAG with seven vertices executed on a homogeneous platform with two processors.

- (one-port, sending) For each P_k, for all $(v_i, v_j) \in E$ such that $\mu(i) = k$ and $\mu(j) \neq k$, the intervals $[\texttt{com}(i,j), \texttt{com}(i,j) + c_{i,j}[$ are disjoint.
- (one-port, receiving) For each P_k, for all $(v_i, v_j) \in E$ such that $\mu(i) \neq k$ and $\mu(j) = k$, the intervals $[\texttt{com}(i,j), \texttt{com}(i,j) + c_{i,j}[$ are disjoint.

The goal is then to minimize the makespan, that is the maximum execution time:

$$M = \max_{1 \leq i \leq n} \{\texttt{st}(i) + w_i\} \,. \tag{1}$$

We are now ready to formalize the MinMakespan optimization problem: *Given a weighted DAG $G = (V, E)$ and p identical processors, the* MinMakespan *optimization problem consists in defining μ (task mapping),* `st` *(task starting times) and* `com` *(communication starting times) so that the makespan M defined in Equation* (1) *is minimized.*

Note that this classical scheduling problem is NP-complete, even without communications, since the problem with n weighted independent tasks and $p = 2$ processors is equivalent to the 2-partition problem [5].

4 Algorithms

We propose a novel heuristic approach to solve the MinMakespan problem, building upon a directed graph partitioner that was recently developed [7]. We compare the results with a classical list scheduling heuristic, that we first describe and adapt for the duplex single-port communication model that we consider in this work (Section 4.1). Next, we introduce three variants of partition-assisted list-based scheduling heuristics in Section 4.2.

4.1 List scheduling heuristic

This simple heuristic maintains an ordered list of *ready* tasks, i.e., tasks that can be executed since all their parent tasks have already been executed. Let `Exec` be the set of tasks that have already

been executed, and let `Ready` be the set of ready tasks. Initially, $\texttt{Exec} = \emptyset$, and $\texttt{Ready} = \{v_i \in V \mid \texttt{Pred}[v_i] = \emptyset\}$. Once a task has been executed, new tasks may become ready. At any time, we have:

$$\texttt{Ready} = \{v_i \in V \mid \texttt{Pred}[v_i] = \emptyset \text{ or } \forall (v_j, v_i) \in E, v_j \in \texttt{Exec}\}. \tag{2}$$

In the first phase, tasks are assigned a priority. Most of the time, the priority of each task is designated to be its bottom level. The bottom level $\texttt{bl}(i)$ of a task $v_i \in V$ is defined as the largest weight of a path from v_i to an output vertex, including the weight w_i of v_i, and all communication costs. Formally,

$$\texttt{bl}(i) = w_i + \begin{cases} 0 & \text{if } \texttt{Succ}[v_i] = \emptyset; \\ \max\limits_{v_j \in \texttt{Succ}[v_i]} c_{i,j} + \texttt{bl}(j) & \text{otherwise.} \end{cases} \tag{3}$$

In the second phase, tasks are assigned to processors. At each iteration, the task of the `Ready` set with the highest priority is selected and scheduled on the processor that would result in the earliest finish time of that task. This finish time depends on the time when that processor becomes available, the computation cost of the task, the communication costs of its input edges, and the finish time of its predecessors. We keep track of the finish time of each processor P_k ($\texttt{comp}_k$), as well as the finish time of sending ($\texttt{send}_k$) and receiving ($\texttt{recv}_k$) operations. When we tentatively schedule a task on a processor, if several communications are needed (meaning that at least two predecessors of the task are mapped on another processor), they cannot be performed at the same time with the duplex single-port communication model. The communications from the predecessors are, then, performed as soon as possible (respecting the finish time of the predecessor and the available time of the sending and receiving ports) in the order of the finish time of the predecessors.

This heuristic is called EFT, for *Earliest Finish Time*, and is described in Algorithm 1. The `Ready` set is stored in a max-heap structure for efficiently retrieving the tasks with the highest priority, and it is initialized at lines 1-5. The computation of the bottom levels for all tasks (line 1) can easily be performed in a single traversal of the graph in $O(|V| + |E|)$ time, see for instance [11]. The main loop traverses the DAG and tentatively schedules a task with the largest bottom level on each processor, in the loop lines 11-20. The best processor is then kept, and all variables are updated on lines 24-29, where the previous ready time, t, is now $\texttt{com}(j, i) + c_{j,i}$ (hence an artificial communication time line 25 when there is in fact no communication between v_j and v_i). Finally, the list of ready tasks is updated line 31, i.e., $\texttt{Exec} \leftarrow \texttt{Exec} \cup \{v_i\}$, and new ready tasks according to Equation (2) are inserted into the max-heap.

The total complexity of Algorithm 1 is hence $O(p^2(|V| + |E|))$. The EFT heuristic will be used as a comparison basis in the rest of this paper.

4.2 Partition-based heuristics

The partition-based heuristics start by computing an acyclic partition of the DAG, using a recent DAG partitioner [7]. This acyclic DAG partitioner takes a DAG with vertex and edge weights, a number of parts K, and an allowable imbalance parameter ε as input. Its output is a partition of the vertices of G into K nonempty pairwise disjoint and collectively exhaustive parts satisfying three conditions: (i) the weight of the parts are balanced, i.e., each part has a total vertex weight of at most $(1+\epsilon)\frac{\sum_{v_i \in V} w_i}{K}$; (ii) the edge cut is reduced; (iii) the partition is acyclic; in other words the inter-part edges between the vertices from different parts should preserve an acyclic dependency structure among the parts. We use this tool to partition the task graph into $K \geq p$ parts and with

Algorithm 1: EFT algorithm

Data: Directed graph $G = (V, E)$, number of processors p
Result: For each task $v_i \in V$, allocation $\mu(i)$ and start time $\texttt{st}(i)$; For each $(v_i, v_j) \in E$, start time $\texttt{com}(i, j)$

$\texttt{bl} \leftarrow ComputeBottomLevels(G)$
$\texttt{Ready} \leftarrow EmptyHeap$
for $v_i \in V$ **do**
  if $\texttt{Pred}[v_i] = \emptyset$ **then**
    Insert v_i in Ready with key $\texttt{bl}(i)$
for $k = 1$ *to* p **do**
  $\texttt{comp}_k \leftarrow 0; \quad \texttt{send}_k \leftarrow 0; \quad \texttt{recv}_k \leftarrow 0;$
while $\texttt{Ready} \neq EmptyHeap$ **do**
  $v_i \leftarrow \texttt{extractMax}(\texttt{Ready})$
  Sort $\texttt{Pred}[v_i]$ in a non-decreasing order of the finish times
  for $k = 1$ *to* p **do**
    for $m = 1$ *to* p **do**
      $\texttt{send}'_m \leftarrow \texttt{send}_m; \quad \texttt{recv}'_m \leftarrow \texttt{recv}_m$
    $\texttt{begin}_k \leftarrow \texttt{comp}_k$
    for $v_j \in \texttt{Pred}[v_i]$ **do**
      if $\mu(j) = k$ **then** $t \leftarrow \texttt{st}(j) + w_j$
      else
        $t \leftarrow c_{j,i} + \max\{\texttt{st}(j) + w_j, \texttt{send}'_{\mu(j)}, \texttt{recv}'_k\}$
        $\texttt{send}'_{\mu(j)} \leftarrow \texttt{recv}'_k \leftarrow t$
      $\texttt{begin}_k \leftarrow \max\{\texttt{begin}_k, t\}$
  $k^* \leftarrow \text{argmin}_k\{\texttt{begin}_k\}$ // Best Processor
  $\mu(i) \leftarrow k^*$
  $\texttt{st}(i) \leftarrow \texttt{comp}_{k^*}$
  for $v_j \in \texttt{Pred}[v_i]$ **do**
    if $\mu(j) = k^*$ **then** $\texttt{com}(j, i) \leftarrow \texttt{st}(j) + w_j - c_{j,i}$
    else
      $\texttt{com}(j, i) \leftarrow \max\{\texttt{st}(j) + w_j, \texttt{send}_{\mu(j)}, \texttt{recv}_{k^*}\}$
      $\texttt{send}_{\mu(j)} \leftarrow \texttt{recv}_{k^*} \leftarrow \texttt{com}(j, i) + c_{j,i}$
    $\texttt{st}(i) \leftarrow \max\{\texttt{st}(i), \texttt{com}(j, i) + c_{j,i}\}$
  $\texttt{comp}_{k^*} \leftarrow \texttt{st}(i) + w_i$
  Update(Ready)

a relatively large imbalance parameter of $\varepsilon = 1.5$. In this paper, we use the recommended version of the approach in [7], namely `CoHyb_CIP`.

Given K parts $V_1, \ldots, V_K$ forming a partition of the DAG, we propose three scheduling heuristics.

EFT-Part The first heuristic, EFT-Part, performs a list scheduling heuristic similar to EFT described in Algorithm 1, but with the additional constraint that two tasks that belong to the same part must be mapped on the same processor. This means that once a task of a part has been mapped, we enforce that other tasks of the same part share the same processors, and hence do not incur any communication among the tasks of the same part. This algorithm is described in Algorithm 2, and its complexity is the same as Algorithm 1.

EFT-Busy One drawback of EFT-Part is that it may happen that the next ready task is in a part that we are just starting (say V_ℓ), while some other parts have not been entirely scheduled. For instance, if processor P_j has already started processing a part $V_{\ell'}$ but has not scheduled all of the tasks of $V_{\ell'}$ yet, EFT-Part may decide to schedule the new task from V_ℓ onto the same processor if it will finish at the earliest time. This may overload the processor and delay other tasks from both $V_{\ell'}$ and V_ℓ.

The second heuristic EFT-Busy checks whether a processor is already busy with an on-going part, and it will allocate a ready task from another part to a processor not already busy. If all processors are busy, EFT-Busy will behave similarly to EFT-Part. This algorithm is described in Algorithm 3, and its complexity is the same as Algorithm 1.

EFT-Macro The last heuristic, EFT-Macro, further focuses on the parts, and schedules a whole part before moving to the next one, so as to avoid problems discussed earlier. This heuristic relies on the fact that the partitioning is acyclic, and hence it is possible to process parts one after another in a topological order.

We extend the definition of ready tasks to parts. A part is ready if all its predecessor parts have already been processed. We also extend the definition of bottom level to parts, by taking the maximum bottom level of tasks in the part.

EFT-Macro selects the ready part with the maximum bottom level (using a max-heap for ready parts, `ReadyParts`), and tentatively schedules the whole part on each processor (lines 15-26). Tasks within the part are scheduled in a topological order and respect dependencies. Incoming communications are scheduled at that time to ensure the one-port model, and outgoing communications are left for later. The processor that minimizes the finish time is selected, and the part is assigned to this processor. The finish times for computation, sending, and receiving are updated. Once a part has been scheduled entirely, the list of ready parts is updated, and the next ready part with the largest bottom level is selected. This heuristic is detailed in Algorithm 4, and its complexity is the same as Algorithm 1.

5 Simulation results

We first describe the simulation setup in Section 5.1, and then present detailed results in Section 5.2.

Algorithm 2: EFT-PART algorithm

Data: Directed graph $G = (V, E)$, number of processors p, acyclic partition of G: $V_1, \ldots, V_K$
Result: For each task $v_i \in V$, allocation $\mu(i)$ and start time $\texttt{st}(i)$; For each $(v_i, v_j) \in E$, start time $\texttt{com}(i, j)$

$\texttt{bl} \leftarrow \mathit{ComputeBottomLevels}(G)$
$\texttt{Ready} \leftarrow \mathit{EmptyHeap}$
for $v_i \in V$ **do**
  if $\texttt{Pred}[v_i] = \emptyset$ **then**
    Insert v_i in Ready with key $\texttt{bl}(i)$
for $k = 1$ *to* p **do**
  $\texttt{comp}_k \leftarrow 0; \quad \texttt{send}_k \leftarrow 0; \quad \texttt{recv}_k \leftarrow 0;$
for $k = 1$ *to* K **do**
  $\texttt{mapPart}_k \leftarrow 0;$
while $\texttt{Ready} \neq \mathit{EmptyHeap}$ **do**
  $v_i \leftarrow \texttt{extractMax}(\texttt{Ready})$
  $\ell \leftarrow$ index of the part of v_i
  Sort $\texttt{Pred}[v_i]$ in a non-decreasing order of the finish times
  if $\texttt{mapPart}_\ell \neq 0$ **then** $k^* \leftarrow \texttt{mapPart}_\ell$
  else
    for $k = 1$ *to* p **do**
      for $m = 1$ *to* p **do**
        $\texttt{send}'_m \leftarrow \texttt{send}_m; \quad \texttt{recv}'_m \leftarrow \texttt{recv}_m;$
      $\texttt{begin}_k \leftarrow \texttt{comp}_k$
      for $v_j \in \texttt{Pred}[v_i]$ **do**
        if $\mu(j) = k$ **then** $t \leftarrow \texttt{st}(j) + w_j$
        else
          $t \leftarrow c_{j,i} + \max\{\texttt{st}(j) + w_j, \texttt{send}'_{\mu(j)}, \texttt{recv}'_k\}$
          $\texttt{send}'_{\mu(j)} \leftarrow \texttt{recv}'_k \leftarrow t$
        $\texttt{begin}_k \leftarrow \max\{\texttt{begin}_k, t\}$
    $k^* \leftarrow \text{argmin}_k\{\texttt{begin}_k\}$ // Best Processor
  $\mu(i) \leftarrow k^*$
  $\texttt{mapPart}_\ell \leftarrow k^*$
  $\texttt{st}(i) \leftarrow \texttt{comp}_{k^*}$
  for $v_j \in \texttt{Pred}[v_i]$ **do**
    if $\mu(j) = k^*$ **then** $\texttt{com}(j, i) \leftarrow \texttt{st}(j) + w_j$
    else
      $\texttt{com}(j, i) \leftarrow \max\{\texttt{st}(j) + w_j, \texttt{send}_{\mu(j)}, \texttt{recv}_{k^*}\}$
      $\texttt{send}_{\mu(j)} \leftarrow \texttt{recv}_{k^*} \leftarrow \texttt{com}(j, i) + c_{j,i}$
    $\texttt{st}(i) \leftarrow \max\{\texttt{st}(i), \texttt{com}(j, i) + c_{j,i}\}$
  $\texttt{comp}_{k^*} \leftarrow \texttt{st}(i) + w_i$
  Update(Ready)

Algorithm 3: EFT-Busy algorithm

Data: Directed graph $G = (V, E)$, number of processors p, acyclic partition of G: $V_1, \ldots, V_K$
Result: For each task $v_i \in V$, allocation $\mu(i)$ and start time $\texttt{st}(i)$; For each $(v_i, v_j) \in E$, start time $\texttt{com}(i, j)$

$\texttt{bl} \leftarrow \mathit{ComputeBottomLevels}(G)$
$\texttt{Ready} \leftarrow \mathit{EmptyHeap}$
for $v_i \in V$ **do**
  if $\texttt{Pred}[v_i] = \emptyset$ **then**
    Insert v_i in Ready with key $\texttt{bl}(i)$
for $k = 1$ *to* p **do**
  $\texttt{comp}_k \leftarrow 0$; $\texttt{send}_k \leftarrow 0$; $\texttt{recv}_k \leftarrow 0$; $\texttt{busy}_k \leftarrow 0$;
for $k = 1$ *to* K **do**
  $\texttt{mapPart}_k \leftarrow 0$;
while $\texttt{Ready} \neq \mathit{EmptyHeap}$ **do**
  $v_i \leftarrow \texttt{extractMax}(\texttt{Ready})$
  $\ell \leftarrow$ index of the part of v_i
  Sort $\texttt{Pred}[v_i]$ in a non-decreasing order of the finish times
  if $\texttt{mapPart}_\ell \neq 0$ **then** $k^* \leftarrow \texttt{mapPart}_\ell$
  else
    $\mathit{allBusy} \leftarrow \mathit{True}$
    for $k = 1$ *to* p **do**
      if $\texttt{busy}_k = 0$ **then** $\mathit{allBusy} \leftarrow \mathit{False}$
    for $k = 1$ *to* p **do**
      if $\texttt{busy}_k > 0$ *and* $\mathit{allBusy} = \mathit{False}$ **then** continue
      for $m = 1$ *to* p **do**
        $\texttt{send}'_m \leftarrow \texttt{send}_m$; $\texttt{recv}'_m \leftarrow \texttt{recv}_m$;
      $\texttt{begin}_k \leftarrow \texttt{comp}_k$
      for $v_j \in \texttt{Pred}[v_i]$ **do**
        if $\mu(j) = k$ **then** $t \leftarrow \texttt{st}(j) + w_j$
        else
          $t \leftarrow c_{j,i} + \max\{\texttt{st}(j) + w_j, \texttt{send}'_{\mu(j)}, \texttt{recv}'_k\}$
          $\texttt{send}'_{\mu(j)} \leftarrow \texttt{recv}'_k \leftarrow t$
        $\texttt{begin}_k \leftarrow \max\{\texttt{begin}_k, t\}$
    $k^* \leftarrow \mathrm{argmin}_k\{\texttt{begin}_k\}$ // Best Processor
  $\mu(i) \leftarrow k^*$
  if $\texttt{mapPart}_\ell = 0$ **then** $\texttt{busy}_{k^*} \leftarrow \texttt{busy}_{k^*} + |V_\ell|$
  $\texttt{busy}_{k^*} \leftarrow \texttt{busy}_{k^*} - 1$
  $\texttt{mapPart}_\ell \leftarrow k^*$
  $\texttt{st}(i) \leftarrow \texttt{comp}_{k^*}$
  for $v_j \in \texttt{Pred}[v_i]$ **do**
    if $\mu(j) = k^*$ **then** $\texttt{com}(j, i) \leftarrow \texttt{st}(j) + w_j$
    else
      $\texttt{com}(j, i) \leftarrow \max\{\texttt{st}(j) + w_j, \texttt{send}_{\mu(j)}, \texttt{recv}_{k^*}\}$
      $\texttt{send}_{\mu(j)} \leftarrow \texttt{recv}_{k^*} \leftarrow \texttt{com}(j, i) + c_{j,i}$
    $\texttt{st}(i) \leftarrow \max\{\texttt{st}(i), \texttt{com}(j, i) + c_{j,i}\}$
  $\texttt{comp}_{k^*} \leftarrow \texttt{st}(i) + w_i$
  Update(Ready)

Algorithm 4: EFT-Macro algorithm

Data: Directed graph $G = (V, E)$, number of processors p, acyclic partition of G: $V_1, \ldots, V_K$
Result: For each task $v_i \in V$, allocation $\mu(i)$ and start time $\texttt{st}(i)$; For each $(v_i, v_j) \in E$, start time $\texttt{com}(i, j)$

$\texttt{bl} \leftarrow \mathit{ComputeBottomLevels}(G)$
for $\ell = 1$ *to* K **do**
  $\texttt{blPart}_\ell \leftarrow \max\{\texttt{bl}(i) \mid v_i \in V_\ell\}$;
$\texttt{ReadyParts} \leftarrow \mathit{EmptyHeap}$
for $\ell = 1$ *to* K **do**
  Sort V_ℓ in the non-decreasing order of $\texttt{bl}$
  if $\forall v_i \in V_\ell,\ \texttt{Pred}[v_i] \backslash V_\ell = \emptyset$ **then**
    Insert V_ℓ in $\texttt{ReadyParts}$ with key $\texttt{blPart}_\ell$
for $k = 1$ *to* p **do**
  $\texttt{comp}_k \leftarrow 0; \quad \texttt{send}_k \leftarrow 0; \quad \texttt{recv}_k \leftarrow 0;$
while $\texttt{ReadyParts} \neq \mathit{EmptyHeap}$ **do**
  $V_\ell \leftarrow \texttt{extractMax}(\texttt{ReadyParts})$
  for $v_i \in V_\ell$ **do**
    Sort $\texttt{Pred}[v_i]$ in a non-decreasing order of the finish times
  for $k = 1$ *to* p **do**
    for $m = 1$ *to* p **do**
      $\texttt{send}'_m \leftarrow \texttt{send}_m; \quad \texttt{recv}'_m \leftarrow \texttt{recv}_m; \quad \texttt{comp}'_m \leftarrow \texttt{comp}_m;$
    for $v_i \in V_\ell$ **do**
      $\texttt{begin}_k \leftarrow \texttt{comp}'_k$
      for $v_j \in \texttt{Pred}[v_i]$ **do**
        if $\mu(j) = k$ **then** $t \leftarrow \texttt{st}(j) + w_j$
        else
          $t \leftarrow c_{j,i} + \max\{\texttt{st}(j) + w_j, \texttt{send}'_{\mu(j)}, \texttt{recv}'_k\}$
          $\texttt{send}'_{\mu(j)} \leftarrow \texttt{recv}'_k \leftarrow t$
        $\texttt{begin}_k \leftarrow \max\{\texttt{begin}_k, t\}$
      $\texttt{comp}'_k \leftarrow \texttt{begin}_k + w_i$
  $k^* \leftarrow \mathrm{argmin}_k\{\texttt{comp}'_k\}$
  for $v_i \in V_\ell$ **do**
    $\mu(i) \leftarrow k^*$
    $\texttt{st}(i) \leftarrow \texttt{comp}_{k^*}$
    for $v_j \in \texttt{Pred}[v_i]$ **do**
      if $\mu(j) = k^*$ **then** $\texttt{com}(j, i) \leftarrow \texttt{st}(j) + w_j$
      else
        $\texttt{com}(j, i) \leftarrow \max\{\texttt{st}(j) + w_j, \texttt{send}_{\mu(j)}, \texttt{recv}_{k^*}\}$
        $\texttt{send}_{\mu(j)} \leftarrow \texttt{recv}_{k^*} \leftarrow \texttt{com}(j, i) + c_{j,i}$
      $\texttt{st}(i) \leftarrow \max\{\texttt{st}(i), \texttt{com}(j, i) + c_{j,i}\}$
    $\texttt{comp}_{k^*} \leftarrow \texttt{st}(i) + w_i$
  Update($\texttt{ReadyParts}$)

Graph	$\|V\|$	$\|E\|$	Degree max.	Degree avg.	#source	#target
598a	110,971	741,934	26	13.38	6,485	8,344
caidaRouterLev.	192,244	609,066	1,071	6.34	7,791	87,577
delaunay-n17	131,072	393,176	17	6.00	17,111	10,082
email-EuAll	265,214	305,539	7,630	2.30	260,513	56,419
fe-ocean	143,437	409,593	6	5.78	40	861
ford2	100,196	222,246	29	4.44	6,276	7,822
luxembourg-osm	114,599	119,666	6	4.16	3,721	9,171
rgg-n-2-17-s0	131,072	728,753	28	5.56	598	615
usroads	129,164	165,435	7	2.56	6,173	6,040
vsp-mod2-pgp2.	101,364	389,368	1,901	7.68	21,748	44,896

Table 1 – Instances from the UFL Collection [3].

Graph	$\|V\|$	$\|E\|$	Degree max.	Degree avg.	#source	#target
cholesky	1,030,204	1,206,952	5,051	2.34	333,302	505,003
fibonacci	1,258,198	1,865,158	206	3.96	2	296,742
quicksort	1,970,281	2,758,390	5	2.80	197,030	3
RSBench	766,520	1,502,976	3,074	3.96	4	5
Smith-water.	58,406	83,842	7	2.88	164	6,885
UTS	781,831	2,061,099	9,727	5.28	2	25
XSBench	898,843	1,760,829	6,801	3.92	5	5

Table 2 – Instances from OCR [21].

5.1 Simulation setup

The experiments were conducted on computers equipped with dual 2.1 GHz Xeon E5-2683 processors and 512GB memory. We have performed an extensive evaluation of the proposed cluster-based scheduling heuristics on DAG instances coming from two sources.

The first set of instances are obtained from the matrices available in the SuiteSparse Matrix Collection (formerly known as the University of Florida Sparse Matrix Collection) [3]. From this collection, we picked ten matrices satisfying the following properties: listed as binary, square, and has at least 100000 rows and at most 2^{26} nonzeros. For each such matrix, we took the strict upper triangular part as the associated DAG instance, whenever this part has more nonzeros than the lower triangular part; otherwise we took the lower triangular part. The ten graphs from the UFL dataset and their characteristics are listed in Table 1.

The second set of instances are from the Open Community Runtime (OCR), an open source asynchronous many-task runtime that supports point-to-point synchronization and disjoint data blocks [21]. We use seven benchmarks from the OCR repository[1]. These benchmarks are either scientific computing programs or mini-apps from real-world applications. The seven graphs from the OCR dataset and their characteristics are listed in Table 2.

To cover a variety of applications, and to compare different heuristics at various regimes, we ran all heuristics on these graphs with random edge costs and random vertex weights using different communication to computation ratios (CCRs). For a graph $G = (V, E)$, the CCR is formally defined as

$$CCR = \frac{\sum_{(v_i,v_j)\in E} c_{i,j}}{\sum_{v_i\in V} w_i} .$$

In order to create instances with a target CCR, we proceed in two steps: (i) we first randomly

[1] `https://xstack.exascale-tech.com/git/public/apps.git`

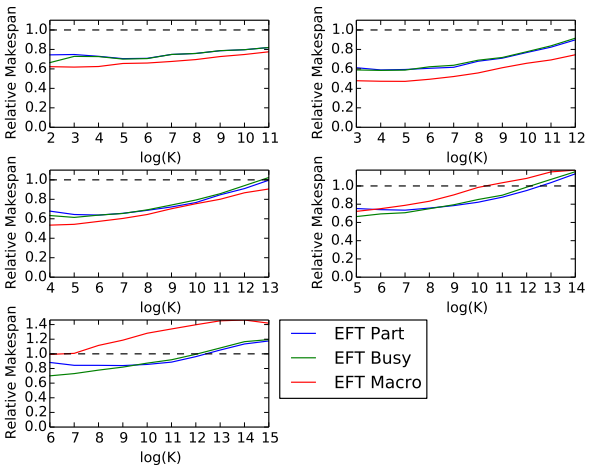


Figure 2 – Relative makespan of the proposed heuristics compared to the standard EFT heuristic on the whole dataset, as a function of the number of parts, with $CCR = 10$ and with 2 (top left), 4 (top right), 8 (middle left), 16 (middle right), and 32 (bottom left) processors.

assign chosen costs and weights between 1 and 10 to each edge and vertex, and then (ii) we scale the edge costs appropriately to yield the desired CCR.

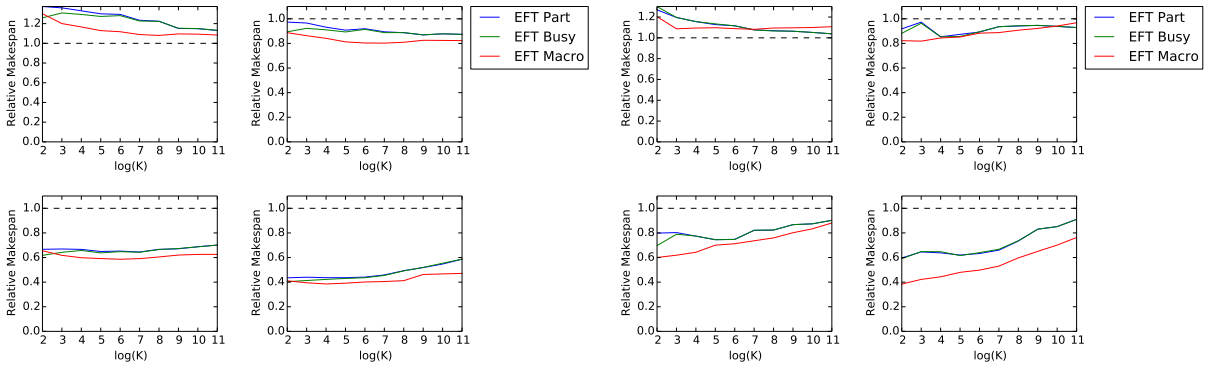
5.2 Results

In all simulations, the running times of EFT, EFT-Part, EFT-Busy, and EFT-Macro are equivalent and negligible compared to the running time of the partitioning algorithm, which is in the order of seconds. We focus here on the makespan obtained by each heuristic.

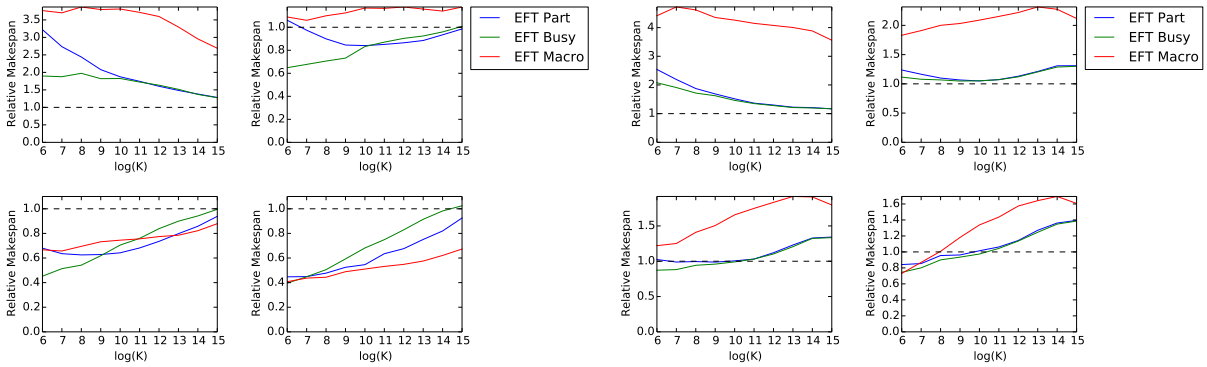
For a given graph and a given architecture, the performance of our partitioning-assisted list-based schedulers will depend on the number of parts K in which we partition the graph. Figure 2 depicts the relative performance of EFT-Part, EFT-Busy, and EFT-Macro compared to EFT on the whole dataset, as a function of K, when $CCR = 10$ and for a number p of processors in $\{2, 4, 8, 16, 32\}$. We can see that when K is too large, the performance of the three heuristics is decreasing. However for all $p \in \{2, 4, 8, 16, 32\}$, there is at least one K value for which the three heuristics outperform the baseline.

This is not the case for all CCR values. Figure 3 provides the relative makespan of EFT-Part, EFT-Busy, and EFT-Macro, for CCR in $\{1, 5, 10, 20\}$. Figures 3a and 3c depict these results for OCR graphs (number of processors equal to 2 and 32), while Figures 3b and 3d depict these results for UFL graphs (number of processors equal to 2 and 32). The evolution of the performance when K increases depends of the number of processors and on the value for CCR. From these figures, we observe that the large values of K provide better results for small values of CCR (for 1 and 5), while small values of K provide better results when the CCR is large. In the rest of this section, we will only consider the best value of K for the proposed heuristics and compare the results with the standard EFT heuristic.

Table 3 displays the detailed results on the whole dataset, when the number of processors is 2, for CCR in $\{1, 5, 10, 20\}$. On average, EFT-Busy provides slightly better results than EFT-Part. When $CCR = 1$, the heuristics often return a makespan that is slightly larger than the one



(a) OCR, $p = 2$ (b) UFL, $p = 2$



(c) OCR, $p = 32$ (d) UFL, $p = 32$

Figure 3 – Relative makespan of the proposed heuristics compared to the standard EFT heuristic on OCR and UFL datasets, as a function of the number of parts with 2 and 32 processors. In each subfigure, top left is $CCR = 1$, top right is $CCR = 5$, bottom left is $CCR = 10$, and bottom right is $CCR = 20$.

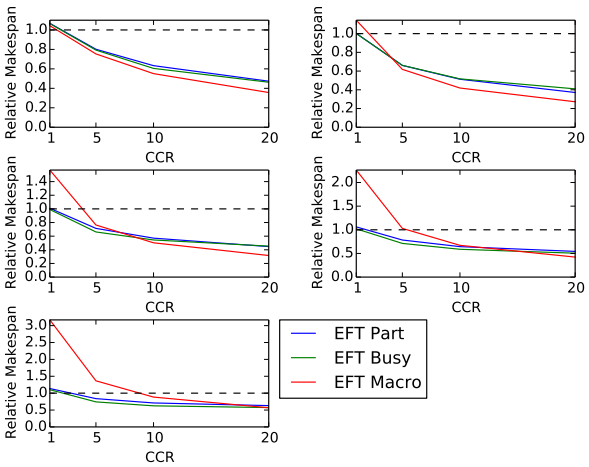


Figure 4 – Relative makespan of the proposed heuristics compared to the standard EFT heuristic on the whole dataset, as a function of the CCR, with 2 (top left), 4 (top right), 8 (middle left), 16 (middle right), and 32 (bottom left) processors.

from EFT, on average by 4 or 6%. However, when the value of CCR is increasing, it is more and more necessary to handle communications correctly. We observe that the proposed three heuristics perform better than the baseline as the CCR increases. When $CCR = 5$, EFT-Part, EFT-Busy, and EFT-Macro provide a 21%, 22%, and 26% improvement compared to the baseline, on average on the whole dataset, when considering an architecture with 2 processors. When $CCR = 20$, these values become respectively 56%, 57%, and 66%.

Figure 4 depicts the relative makespan of the proposed three heuristics compared to EFT as a function of CCR and for an architecture of 2, 4, 8, 16, and 32 processors. Comparing the relative performance of EFT-Part and EFT-Busy across the sub-figures, one observes that EFT-Part and EFT-Busy have more or less stable performance with the increasing number of processors. On the other hand, EFT-Macro's performance drops as p increases. We can notice that the performance of EFT-Part and EFT-Busy mostly depends on the value of CCR, but remains the same when the number of processors varies. EFT-Macro performs worse than the other two heuristics for small values of CCR. This phenomenon is amplified when the number of processors increases. However, for p tested, EFT-Macro performs better and better when the CCR increases, to finally outperform every other heuristics on average when the CCR is large enough.

6 Conclusion

We proposed three new list-based scheduling techniques based on an acyclic partition of the DAGs: EFT-Part, EFT-Busy, and EFT-Macro. The acyclicity of the partition ensures that we can schedule a part of the partition in its entirety as soon as its input nodes are available. Hence, we have been able to design specific list-based scheduling techniques that would not have been possible without an acyclic partition of the DAG. We compared our scheduling techniques with the widely used EFT heuristic.

Our experiments suggest that the relative performance of EFT-Part and EFT-Busy compared to the baseline does not depend on the number of processors, which means that these heuristics scale

	$CCR = 1$				$CCR = 5$			
Graph	EFT	EFT-Part	EFT-Busy	EFT-Macro	EFT	EFT-Part	EFT-Busy	EFT-Macro
598a	3058476	1.03	1.03	1.01	5857127	0.64	0.64	0.58
caidaRouterLevel	5337718	1.00	1.00	0.99	8937548	0.92	0.92	0.76
delaunay-n17	3606092	1.01	1.01	1.03	5431960	0.69	0.69	0.69
email-EuAll	7711619	0.98	0.98	1.19	18123055	0.57	0.57	0.61
fe-ocean	3949464	1.15	1.15	1.01	5185419	0.88	0.88	0.78
ford2	2781775	1.00	1.00	1.00	4024990	0.71	0.71	0.70
luxembourg-osm	3152973	1.00	1.00	1.00	3506686	0.90	0.90	0.90
rgg-n-2-17-s0	3601079	1.05	1.05	1.05	4585262	0.84	0.84	0.84
usroads	3550396	1.05	1.05	1.09	4097201	0.91	0.91	0.92
vsp-mod2-pgp2-slptsk	2794636	1.00	1.00	1.00	5509790	0.67	0.67	0.65
cholesky	30603433	1.09	1.04	1.09	49102625	0.79	0.65	0.76
fibonacci	34601228	1.07	1.07	1.31	44109081	0.84	0.84	0.99
quicksort	54162227	1.00	1.01	1.02	71605033	0.76	0.76	0.77
RSBench	26941941	1.04	1.04	0.82	45191117	0.66	0.66	0.50
Smith-waterman	1661676	1.22	1.23	1.10	2196692	1.06	1.06	0.90
UTS	31904401	1.32	1.32	1.09	51957000	0.83	0.83	0.67
XSBench	26941941	1.04	1.04	0.82	49993817	0.97	0.97	0.79
Geomean	**1.00**	**1.06**	**1.06**	**1.04**	**1.00**	**0.79**	**0.78**	**0.74**

	$CCR = 10$				$CCR = 20$			
Graph	EFT	EFT-Part	EFT-Busy	EFT-Macro	EFT	EFT-Part	EFT-Busy	EFT-Macro
598a	9669102	0.42	0.42	0.39	17038485	0.48	0.48	0.22
caidaRouterLevel	14638583	0.92	0.92	0.60	26745328	0.96	0.97	0.36
delaunay-n17	9216833	0.41	0.41	0.42	17567627	0.22	0.22	0.23
email-EuAll	32997285	0.49	0.49	0.36	67066585	0.48	0.48	0.23
fe-ocean	7202636	0.65	0.65	0.56	11573357	0.45	0.45	0.35
ford2	6068545	0.49	0.49	0.47	10538479	0.28	0.28	0.27
luxembourg-osm	3941446	0.80	0.80	0.80	4801062	0.67	0.66	0.66
rgg-n-2-17-s0	5892674	0.68	0.68	0.65	9094485	0.47	0.47	0.44
usroads	5327111	0.72	0.72	0.72	8428888	0.47	0.47	0.47
vsp-mod2-pgp2-slptsk	9460442	0.86	0.53	0.51	19887584	0.66	0.65	0.25
cholesky	75676369	0.52	0.42	0.52	130153391	0.30	0.24	0.29
fibonacci	64454756	0.59	0.59	0.68	110167490	0.36	0.35	0.39
quicksort	104478680	0.52	0.52	0.53	173055640	0.32	0.32	0.31
RSBench	67674107	0.49	0.49	0.35	109245784	0.35	0.33	0.23
Smith-waterman	3408415	0.81	0.75	0.69	5694549	0.53	0.45	0.49
UTS	74335883	0.58	0.59	0.49	117598932	0.42	0.40	0.36
XSBench	59646365	0.83	0.82	0.63	77257208	0.64	0.64	0.51
Geomean	**1.00**	**0.61**	**0.59**	**0.54**	**1.00**	**0.44**	**0.43**	**0.34**

Table 3 – The makespan of EFT in absolute numbers, and those of EFT-Part, EFT-Busy, and EFT-Macro relative to EFT on UFL and OCR graphs, when the number of processors p is 2, and for CCR in $\{1, 5, 10, 20\}$.

well. They provide a steady improvement over the classic EFT heuristic and their performance are even better when the ratio between communication and computation is large. EFT-Macro seems to not scale when the number of processors increases. Nevertheless, when the ratio between communication and computation is large, it usually outperforms all the other heuristics, especially with a small number of parts.

There are two immediate lines of research that we will pursue. First, a thorough comparison with the proposed heuristics with some existing cluster-based heuristics implemented by Wang and Sinnen [18] (the implementations were not available at the time of our experimentation, but we hope to obtain them). Second, an adaptation of the proposed heuristics to the heterogeneous processing systems will be needed. A difficulty arises in addressing the communication cost, which requires updating the partitio

References

[1] T. L. Adam, K. M. Chandy, and J. Dickson. A comparison of list schedules for parallel processing systems. *Communications of the ACM*, 17(12):685–690, 1974.

[2] I. Ahmad and Y.-K. Kwok. On exploiting task duplication in parallel program scheduling. *IEEE Trans. Parallel Distrib. Syst.*, 9(9):872–892, 1998.

[3] T. A. Davis and Y. Hu. The University of Florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, 2011. ISSN 0098-3500.

[4] I. T. Foster, M. Fidler, A. Roy, V. Sander, and L. Winkler. End-to-end quality of service for high-end applications. *Computer Communications*, 27(14):1375–1388, 2004.

[5] M. R. Garey and D. S. Johnson. *Computers and Intractability, a Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, 1979.

[6] T. Glatard, J. Montagnat, D. Lingrand, and X. Pennec. Flexible and efficient workflow deployement of data-intensive applications on grids with MOTEUR. *Int. Journal of High Performance Computing and Applications*, 2008.

[7] J. Herrmann, M. Y. Özkaya, B. Uçar, K. Kaya, and Ü. V. Çatalyürek. Acyclic partitioning of large directed acyclic graphs. Research Report RR-9163, Inria - Research Centre Grenoble – Rhône-Alpes, Mar 2018. URL `https://hal.inria.fr/hal-01744603`.

[8] J.-J. Hwang, Y.-C. Chow, F. D. Anger, and C.-Y. Lee. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM Journal on Computing*, 18(2):244–257, 1989.

[9] H. Kanemitsu, M. Hanada, and H. Nakazato. Clustering-based task scheduling in a large number of heterogeneous processors. *IEEE Transactions on Parallel and Distributed Systems*, 27(11):3144–3157, Nov 2016. ISSN 1045-9219. doi: 10.1109/TPDS.2016.2526682.

[10] Y.-K. Kwok and I. Ahmad. Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors. *IEEE transactions on parallel and distributed systems*, 7(5):506–521, 1996.

[11] Y.-K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Survey*, 31(4):406–471, 1999. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/344588.344618.

[12] Y.-K. Kwok and I. Ahmad. Benchmarking and comparison of the task graph scheduling algorithms. *Journal of Parallel and Distributed Computing*, 59(3):381–422, 1999.

[13] S. Mingsheng, S. Shixin, and W. Qingxian. An efficient parallel scheduling algorithm of dependent task graphs. In *Parallel and Distributed Computing, Applications and Technologies, 2003. PDCAT'2003. Proceedings of the Fourth International Conference on*, pages 595–598. IEEE, 2003.

[14] A. Radulescu and A. J. Van Gemund. Low-cost task scheduling for distributed-memory machines. *IEEE Transactions on Parallel and Distributed Systems*, 13(6):648–658, 2002.

[15] V. Sarkar. Partitioning and scheduling parallel programs for execution on multiprocessors. Technical report, Stanford Univ., CA (USA), 1987.

[16] 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.

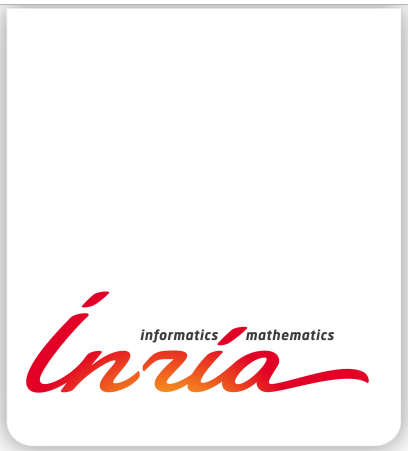
[17] H. Topcuoglu, S. Hariri, and M. Y. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Trans. Parallel Distributed Systems*, 13(3):260–274, 2002.

[18] H. Wang and O. Sinnen. List-scheduling vs. cluster-scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 2018. ISSN 1045-9219. doi: 10.1109/TPDS.2018.2808959. in press.

[19] M.-Y. Wu and D. D. Gajski. Hypertool: A programming aid for message-passing systems. *IEEE transactions on parallel and distributed systems*, 1(3):330–343, 1990.

[20] T. Yang and A. Gerasoulis. Dsc: Scheduling parallel tasks on an unbounded number of processors. *IEEE Transactions on Parallel and Distributed Systems*, 5(9):951–967, 1994.

[21] L. Yu and V. Sarkar. GT-Race: Graph traversal based data race detection for asynchronous many-task runtimes. In *European Conference on Parallel Processing*. Springer, 2018. Accepted.



RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée

655 avenue de l'Europe Montbonnot

38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399