A scalable clustering-based task scheduler for homogeneous processors using DAG partitioning

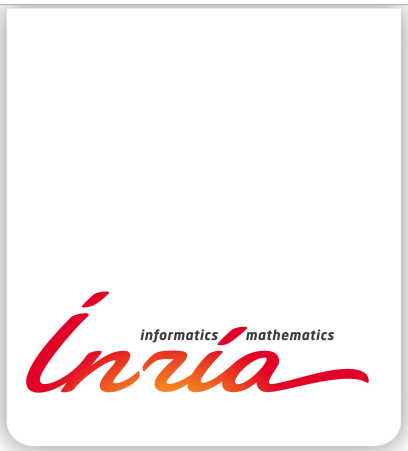
Yusuf M. Özkaya, Anne Benoit, Bora Uçar, Julien Herrmann, Ümit V. Çatalyürek

A scalable clustering-based task scheduler for homogeneous processors using DAG partitioning

M. Yusuf Özkaya, Anne Benoit, Bora Uçar, Julien Herrmann, Ümit V. Çatalyürek

RESEARCH REPORT
N° 9185
January 2019
Project-Team ROMA

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



A scalable clustering-based task scheduler for homogeneous processors using DAG partitioning

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

Project-Team ROMA

Research Report n° 9185 — January 2019 — 34 pages

* 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

Abstract: When scheduling a directed acyclic graph (DAG) of tasks with computational costs on computational platforms, a good trade-off between load balance and data locality is necessary. List-based scheduling techniques are commonly used greedy approaches for this problem. The downside of 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 new list-based scheduling heuristics based on clustering for homogeneous platforms, under the realistic duplex single-port communication model. 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. Furthermore, we compare our heuristics to the best state-of-the-art list-scheduling and clustering heuristics, and obtain more than three times better makespan in cases with many communications.

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

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 sont des approches gloutonnes communément utilisées pour ce problème. Les inconvénients de telles heuristiques 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 de nouvelles heuristiques d'ordonnancement de liste pour des plates-formes homogènes, avec un modèle de communications duplexe un-port réaliste. 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. De plus, nous comparons nos heuristiques aux meilleurs compétiteurs de la littérature, et nous obtenons un temps d'exécution total plus de trois fois meilleur dans des cas avec de nombreuses communications.

Mots-clés : ordonnancement de liste, clustering, 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 [17] with associated communication costs [17, 23]. The classical objective function is to minimize the total execution time, or *makespan*, and this problem is denoted as $P|prec, c_{i,j}|C_{max}$ in the scheduling literature. 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, 12, 18, 20, 22, 25, 26, 27]. 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 [14, 15, 21, 26, 27, 28], 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 partitioners [11]. 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 the *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, and we confirm this hypothesis in our experiments.

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 minimizing the total execution time, or *makespan*.

We propose several 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, since the partition 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 tasks' 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 (meta-heuristics) of partitioning-assisted list-based scheduler, taking different decisions about how to schedule tasks within a part. These variants run on top of two classical list-based schedulers: (1) BL-EST chooses the task with largest bottom-level first (BL), and assigns the task on the processor with the earliest start time (EST), while (2) ETF tries all ready tasks on all processors and picks the combination with the earliest EST first (hence with a higher complexity). The proposed meta-heuristics can be used with any other list scheduler and DAG partitioner, hence they provide a flexible solution to DAG scheduling. Also, we experimentally evaluate the new algorithms against the two baseline list-based schedulers (BL-EST and ETF) and one baseline cluster-based scheduler (DSC-GLB-ETF), since ETF and DSC-GLB-ETF are the winners of the recent comparison done by Wang and Sinnen [26]. However, unlike [26], we follow the realistic 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.

In other words, we propose a novel algorithmic framework for DAG scheduling, building upon a multi-level *acyclic* DAG partitioner for the clustering phase. Furthermore, we consider a realistic communication model, contrarily to most theoretical work on scheduling. Thus, our algorithms lend themselves as efficient heuristics with no lower bounds or performance guarantees. However, as demonstrated in the results section, they drastically outperform state-of-the-art schedulers under more realistic scenarios, such as single-port communication model and when communications are more costly than computations. For example, one of the datasets we experimented includes several DAGs corresponding to high-performance computing (HPC) applications that use Open Community Runtime (OCR) framework [29], on which we achieve more than three times better makespan than the state-of-the-art heuristic with large CCRs.

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. Kwok and Ahmad [17] give an excellent survey and taxonomy of task scheduling methods and some benchmarking techniques to compare these methods [16].

On the theoretical side, a related problem of minimizing the makespan of a DAG on identical processors without communication costs ($P|prec|C_{max}$) has been extensively studied. Graham's seminal list-scheduling algorithm [9] has been known for a long time to be a $(2-\frac{1}{p})$-approximation algorithm, where p is the number of processors. It has then been shown that it is NP-hard to improve upon this approximation ratio, assuming a new variant of the unique games conjecture [24]. Several works further focus on unit execution times to derive theoretical results (lower bounds, complexity results), see for instance [13].

On the practical side, communication costs cannot be neglected, and it becomes much harder

to derive theoretical guarantees. Even the problem with unit execution time and unit communication time (UET-UCT) is NP-hard [19]. Hence, the $P|prec, c_{i,j}|C_{max}$ problem is usually tackled through heuristics. For coarse-grain graphs, a guaranteed heuristic based on a linear programming formulation of the problem was proposed [10], and it was proven that there always exists a linear optimal clustering [7].

DAG scheduling heuristics can be divided into two groups 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 [26] gives a catalog of list-scheduling and cluster-scheduling heuristics and compares their performance. These algorithms take the entire task graph as input, similar to our approach.

In the list-based scheduling approach [1, 12, 18, 20, 22, 25, 27], each task in the DAG is first assigned a priority. Then, the tasks are sorted in descending order of priorities, thereby resulting in a priority list. Finally, the tasks are scheduled in topological order, with the highest priorities first. There are also two variants of list-scheduling based on how priorities are computed: *static* and *dynamic*. In the static list-scheduling, priorities are pre-computed and do not change during the algorithm. In the dynamic list-scheduling, task priorities are updated as the predecessor tasks are scheduled. The list-scheduling based heuristics usually have low complexity and are easy to implement and understand. In general, the static list-scheduling algorithms also have low computational complexity, whereas dynamic list-scheduling algorithms have higher complexity, due to priority updates.

In the cluster-based scheduling approach [14, 15, 21, 26, 27, 28], 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 partition tasks into $K \geq p$ clusters, where p is 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. Hence, 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.

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 platforms [5, 8]; HEFT [25] is a common approach for this purpose. 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. Vertices without any predecessors are called *source* nodes, and the ones without any successors are called *target* nodes. Every vertex $v_i \in V$ has a weight, denoted by w_i,

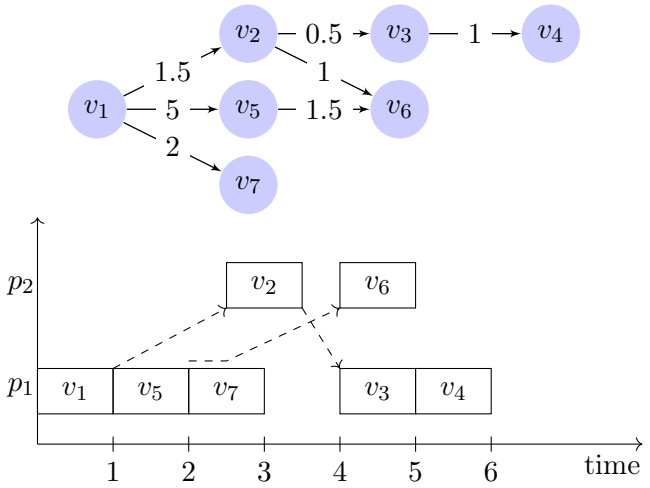


Figure 1 – Example of a small DAG with seven vertices executed on a homogeneous platform with two processors.

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 cost 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, 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 after 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 $\texttt{com}(i, j) + c_{i,j} \leq \texttt{st}(j)$.
- (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 [6].

4 Algorithms

We propose novel heuristic approaches to solve the MinMakespan problem, using a recent directed graph partitioner [11]. We compare the results with classical list-based and clustering heuristics, that we first describe and adapt for the duplex single-port communication model (Section 4.1). Next, we introduce three variants of partition-assisted list-based scheduling heuristics in Section 4.2.

For convenience, Table 1 summarizes acronyms used in the paper, in particular in the heuristic names, and Table 2 summarizes the main features of all considered approaches.

Notation	Meaning
`DAG`	Directed Acyclic Graph
`CCR`	Communication-to-Computation Ratio
`BL`	Bottom-Level
`TL`	Top-Level
`EST`	Earliest Start Time
`ETF`	Earliest EST First
`DSC`	Dominant Sequence Clustering
`GLB`	Guided Load Balancing

Table 1 – Acronyms.

		(Cluster-based only)			task priority type	(partition priority) task priority	placement
		clustering approach	load balancing	produced clusters			
List-based	BL-EST				static	BL	EST-processor
	ETF				dynamic	EST	
Clustering-based	DSC-GLB-ETF	cyclic cluster graph limited refinement	Guided Load Balancing	Cyclic, non-convex graph	dynamic	TL+BL	EST task within cluster (processor)
Proposed Partitioning-based	*-PART	acyclic cluster graph better refinement		Directed Acyclic Graph	priority type of * (static or dynamic)	priority of * (BL or EST)	EST-Processor
	*-BUSY						EST-idle Processor
	*-MACRO						Earliest (Part) Finish Time-Processor

Table 2 – Heuristic approaches to solve MINMAKESPAN.

4.1 State-of-the-art scheduling heuristics

We first consider the best alternatives from the list-based and cluster-based scheduling heuristics presented by Wang and Sinnen [26]. We consider one static list-scheduling heuristic (BL-EST), the best dynamic priority list-based scheduling heuristic for real application graphs (ETF), and the best cluster-based scheduling heuristic (DSC-GLB-ETF).

BL-EST. This simple heuristic maintains an ordered list of *ready* tasks, i.e., tasks that can be executed since all their predecessors have already been executed. Let $\texttt{Ex}$ be the set of tasks that have already been executed, and let $\texttt{Ready}$ be the set of ready tasks. Initially, $\texttt{Ex} = \emptyset$, 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:

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

In the first phase, tasks are assigned a *priority*, which is designated to be its bottom level (hence the name BL). 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 a target node (vertex without successors), 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 $\texttt{Ready}$ set with the highest priority is selected and scheduled on the processor that would result in the earliest start time of that task. The start time depends on the time when that processor becomes available, 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 other processors), 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 BL-EST, for *Bottom-Level Earliest-Start-Time*, and is described in Algorithm 1. The $\texttt{Ready}$ 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 [17]. 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 processor with the earliest start time is then saved, and all variables are updated on lines 24-29. When updating $\texttt{com}(j,i)$, if v_i and its predecessor v_j are mapped on the same processor, communication start time is artificially set to $\texttt{st}(j) + w_j - c_{j,i}$ in line 25, so that $\texttt{st}(i)$ can be computed correctly in line 29. Finally, the list of ready tasks is updated line 31, i.e., $\texttt{Ex} \leftarrow \texttt{Ex} \cup \{v_i\}$, and new ready tasks according to Equation (2) are inserted into the max-heap.

The total complexity of Algorithm 1 is hence $O(p^2|V|+|V|\log|V|+p|E|)$: $p^2|V|$ for lines 11-13, $|V|\log|V|$ for the heap operations (we perform $|V|$ times the extraction of the maximum, and the insertion of new ready tasks into the heap), and $p|E|$ for lines 15-20. The BL-EST heuristic will be used as a comparison basis in the rest of this paper. The space complexity is O(p + |V | + |E|).

ETF. We also consider a dynamic priority list scheduler, ETF. For each ready task, this algorithm computes the earliest start time (EST) of the task. Then, it schedules the ready task with the earliest EST, hence the name ETF, for *Earliest EST First*. Since we tentatively schedule each ready task, the time complexity of ETF is higher than BL-EST; it becomes $O(p^2|V|^2 + p|V||E|)$. The space complexity is the same as BL-EST, i.e., $O(p + |V| + |E|)$.

DSC-GLB-ETF. The clustering scheduling algorithm used as a basis for comparison is one of the best ones identified by Wang and Sinnen [26], namely, the DSC-GLB-ETF algorithm. It uses dominant sequence clustering (DSC), then merges clusters with guided load balancing (GLB), and finally orders tasks using earliest EST first (ETF). We refer the reader to [26] for more details about this algorithm.

Algorithm 1: BL-EST 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
      $\texttt{com}(j, i) \leftarrow \max\{\texttt{st}(j) + w_j, \texttt{send}_{\mu(j)}, \texttt{recv}_{k^*}\}$
      $\texttt{send}_{\mu(j)} \leftarrow \texttt{recv}_{k^*} \leftarrow \texttt{com}(j, i) + c_{j,i}$
    $\texttt{st}(i) \leftarrow \max\{\texttt{st}(i), \texttt{com}(j, i) + c_{j,i}\}$
  $\texttt{comp}_{k^*} \leftarrow \texttt{st}(i) + w_i$
  Insert new ready tasks into Ready

4.2 Partition-based heuristics

The partition-based heuristics start by computing an acyclic partition of the DAG, using a recent DAG partitioner [11]. 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 minimized; (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 $K = \alpha \times p$ parts, hence $\alpha \geq 1$ can be interpreted as the average number of clusters per processor. We choose an imbalance parameter of $\varepsilon = 1.1$ to have relatively balanced clusters; other values of ε led to similar results. In this paper, we use the recommended version of the approach in [11], namely `CoHyb_CIP`. It may not always be possible to find a feasible partition with the given constraints, especially for small graphs and large α and K values. However, since our main goal is to achieve good clustering, not perfect balance, we will continue with whatever partitioning found by our tool, even if it is not balanced (which only happened very rarely in our experiments).

Given K parts $V_1, \ldots, V_K$ forming a partition of the DAG, we propose three variants of scheduling heuristics. Note that the variants are designed on top of BL-EST and ETF, but they can easily be adapted to any other list-based scheduling algorithm since, in essence, these heuristics are capturing a hybrid approach between cluster-based and list-based scheduling algorithms using DAG partitioning.

***-Part.** The first variant, denoted *-Part, is used in this paper on top of BL-EST or ETF. The BL-EST-Part heuristic (resp. ETF-Part) performs a list scheduling heuristic similar to BL-EST described in Algorithm 1 (resp. similar to ETF), 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 processor, and hence do not incur any communication cost among the tasks of the same part. algorithm. This variant is denoted *-Part (BL-EST-Part or ETF-Part depending on the corresponding baseline algorithm) and has the same complexity as the corresponding baseline The pseudo-code of *-Part can be found in Algorithm 2. An instantiation of this algorithm, BL-EST-Part algorithm is described in Algorithm 3.

***-Busy.** One drawback of the *-Part heuristics 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, *-Part may decide to schedule the new task from V_ℓ onto the same processor if it will start at the earliest time. This may overload the processor and delay other tasks from both $V_{\ell'}$ and V_ℓ.

The second variant, *-Busy, checks whether a processor is already busy with an on-going part, and it does not allocate a ready task from another part to a busy processor, unless all processors are busy. In this latter case, *-Busy behaves similarly to *-Part. This algorithm is described in Algorithm 4, and the complexity of this variant is the same as the list scheduling heuristic on top

Algorithm 2: *-PART algorithm

Data: Directed graph $G = (V, E)$, number of processors p, acyclic partition of G: $V_1, \ldots, V_K$, a task priority algorithm TP
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 $\texttt{com}(i, j)$

```
Initialize Ready with all v_i without any predecessors
Initialize mapPart array of size K to Null
while Ready is not empty do
    v_i ← TP(Ready)
                                  // TP() returns highest priority task from the Ready list.
    ℓ ← part id of v_i
    if mapPart[ℓ] = Null then
        Sort Pred[v_i] in a non-decreasing order of the finish times
        for k = 1 to p do
            begin_k ← comp_k
            for v_j ∈ Pred[v_i] do
                Update earliest possible begin time begin_k with the latest finishing predecessor
                 communication
        k* ← argmin_k{begin_k}                                   // Best Processor
    else
        k* ← mapPart[ℓ]
    μ(i) ← k*
    mapPart[ℓ] ← k*
    Assign communication times (in Pred[v_i] order) and update computation times
    Insert new ready tasks into Ready
```

of which the the variant is run, in our case BL-EST or ETF. The BL-EST-BUSY algorithm, as an instantiation of *-BUSY, is described in Algorithm 5.

***-MACRO.** The last variant, *-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 strongly 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. Hence, when a part is ready, all predecessors of tasks from that part have already been scheduled. We also extend the definition of bottom level to parts, by taking the maximum bottom level of tasks in the part.

Algorithm 3: BL-EST-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
      $\texttt{com}(j, i) \leftarrow \max\{\texttt{st}(j) + w_j, \texttt{send}_{\mu(j)}, \texttt{recv}_{k^*}\}$
      $\texttt{send}_{\mu(j)} \leftarrow \texttt{recv}_{k^*} \leftarrow \texttt{com}(j, i) + c_{j,i}$
    $\texttt{st}(i) \leftarrow \max\{\texttt{st}(i), \texttt{com}(j, i) + c_{j,i}\}$
  $\texttt{comp}_{k^*} \leftarrow \texttt{st}(i) + w_i$
  Insert new ready tasks into Ready

The generic *-MACRO is detailed in Algorithm 6. The algorithm relies on two priority algorithms, one for selecting parts, and one for selecting tasks. These priorities can be static, such as BL (selects parts or tasks with maximum bottom level), or dynamic, such as earliest start time as in ETF. Once a part has been selected, the algorithm tentatively schedules the whole part on each processor (lines 4-14). Tasks within the part are selected with the second priority algorithm. Incoming communications are scheduled at that time to ensure the single-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, since we aim at finishing a part as soon as possible to minimize the makespan. 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 highest priority is selected.

In ETF-MACRO, similarly to heuristic ETF, we tentatively schedule each ready part, and then each task, and at each step, we keep the best schedule. The time complexity of these variants are slightly different than the list scheduling heuristics on top of which the variant is run, because of part-by-part scheduling. For ETF-MACRO, the complexity is $O(p^4 + p^3|V| + p^2|V|^2 + p|V||E|)$.

BL-MACRO is detailed in Algorithm 7. The algorithm 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 by non-increasing $\texttt{bl}(i)$'s, hence respecting dependencies. Incoming communications are scheduled at that time to ensure the single-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 has the same complexity as Algorithm 1.

Algorithm 4: *-BUSY algorithm

Data: Directed graph $G = (V, E)$, number of processors p, acyclic partition of G: $V_1, \ldots, V_K$, a task priority algorithm TP

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 $\texttt{com}(i, j)$

```
1  Initialize Ready with all v_i without any predecessors
2  Initialize mapPart array of size K to Null
3  Initialize busy array of size K to zero
4  while Ready is not empty do
5      v_i ← TP(Ready)
                                  // TP() returns highest priority task from the Ready list.
6      ℓ ← part id of v_i
7      if mapPart[ℓ] = Null then
8          allBusy ← whether all processors are busy or not
9          Sort Pred[v_i] in a non-decreasing order of the finish times
10         for k = 1 to p do
11             if busy_k > 0  and  allBusy = False then continue;
12             begin_k ← comp_k
13             for v_j ∈ Pred[v_i] do
14                 Update earliest possible begin time begin_k with the latest finishing predecessor
                   communication.
15         k* ← argmin_k{begin_k}                                   // Best Processor
16     else
17         k* ← mapPart[ℓ]
18     μ(i) ← k*
19     if mapPart_ℓ = Null then busy_k* ← busy_k* + |V_ℓ|
20     mapPart[ℓ] ← k*
21     busy_k* ← busy_k* − 1
22     st(i) ← comp_k*
23     Assign communication times (in Pred[v_i] order) and update computation times
24     Insert new ready tasks into Ready
```

Algorithm 5: BL-EST-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
      $\texttt{com}(j, i) \leftarrow \max\{\texttt{st}(j) + w_j, \texttt{send}_{\mu(j)}, \texttt{recv}_{k^*}\}$
      $\texttt{send}_{\mu(j)} \leftarrow \texttt{recv}_{k^*} \leftarrow \texttt{com}(j, i) + c_{j,i}$
    $\texttt{st}(i) \leftarrow \max\{\texttt{st}(i), \texttt{com}(j, i) + c_{j,i}\}$
  $\texttt{comp}_{k^*} \leftarrow \texttt{st}(i) + w_i$
  Insert new ready tasks into Ready

Algorithm 6: *-MACRO algorithm

Data: Directed graph $G = (V, E)$, number of processors p, acyclic partition of G: $V_1, \ldots, V_K$, a partition priority algorithm PP, a task priority algorithm TP

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 $\texttt{com}(i, j)$

1 Initialize `ReadyParts` with all V_i without any predecessors
2 **while** `ReadyParts` *is not empty* **do**
3 $V_i \leftarrow PP(\texttt{ReadyParts})$
 `// PP() returns highest priority part from the ReadyParts list.`
4 **for** $k = 1$ *to* p **do**
5 $\texttt{end}_k \leftarrow \texttt{comp}_k$
6 Initialize `Ready` with all tasks from V_i with no unscheduled predecessors
7 **while** `Ready` *is not empty* **do**
8 $v_x \leftarrow TP(\texttt{Ready})$
 `// TP() returns highest priority task from the Ready list.`
9 Sort $\texttt{Pred}[v_x]$ in a non-decreasing order of the finish times
10 Assign communication times (in $\texttt{Pred}[v_x]$ order) and update computation times
11 $\mu(i) \leftarrow k$
12 Update $st(x)$, `com` and `comp`
13 Update $\texttt{end}_k$ with the latest finishing task
14 Insert new ready tasks from same part into `Ready`
15 $k^* \leftarrow \text{argmin}_k\{\texttt{end}_k\}$ `// Best Processor`
16 Schedule part V_i to processor k^* (with the same procedure as in lines 6-14)
17 Insert new ready parts into `ReadyParts`

Algorithm 7: BL-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 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 non-decreasing order of 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 $v_i \in V_\ell$ *in non-increasing* $\texttt{bl}(i)$ *order* **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 $v_i \in V_\ell$ *in non-increasing* $\texttt{bl}(i)$ *order* **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
        $\texttt{com}(j, i) \leftarrow \max\{\texttt{st}(j) + w_j, \texttt{send}_{\mu(j)}, \texttt{recv}_{k^*}\}$
        $\texttt{send}_{\mu(j)} \leftarrow \texttt{recv}_{k^*} \leftarrow \texttt{com}(j, i) + c_{j,i}$
      $\texttt{st}(i) \leftarrow \max\{\texttt{st}(i), \texttt{com}(j, i) + c_{j,i}\}$
    $\texttt{comp}_{k^*} \leftarrow \texttt{st}(i) + w_i$
  Insert new ready parts into $\texttt{ReadyParts}$

5 Simulation results

We first describe the simulation setup in Section 5.1, in particular, the different instances that we use in the simulations. Next, we compare the baseline algorithms under different communication models (communication-delay model vs. realistic model) in Section 5.2. Section 5.3 shows the impact of the number of parts used by the partitioner, the communication-to-computation ratio (CCR), the number of processors, and the edge cut. Finally, we present detailed simulation results in Section 5.4 and summarize these results in Section 5.5.

5.1 Simulation setup

The experiments were conducted on a computer 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 instances coming from three sources.

The first set of instances is from Wang and Sinnen's work [26]. This set contains roughly 1600 instances of graphs, each having 50 to 1151 nodes. All graphs have three versions for CCRs 0.1, 1, and 10. The dataset includes a wide range of real world, regular structure, and random structure graphs; more details about them are available in the original paper [26]. Since the graphs are up to 1151 nodes, we refer to this dataset as the *small* dataset.

The second set of instances is 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 3.

The third set of instances is from the Open Community Runtime (OCR), an open source asynchronous many-task runtime that supports point-to-point synchronization and disjoint data blocks [29]. We use seven benchmarks from the OCR repository[1]. These benchmarks are either scientific computing programs or mini-apps from real-world applications whose graphs' characteristics

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

			Degree			
Graph	$\|V\|$	$\|E\|$	max.	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 3 – 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 4 – Instances from OCR [29].

are listed in Table 4.

To cover a variety of applications, we consider UFL and OCR instances 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

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

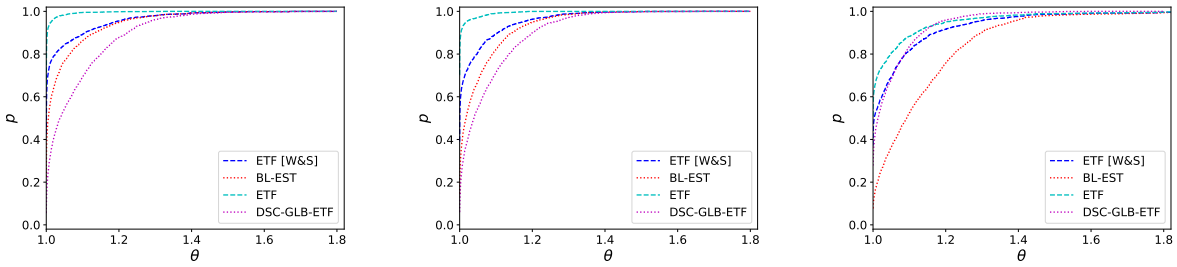
In order to create instances with a target CCR, we proceed in two steps: (i) we first randomly assign costs and weights between 1 and 10 to each edge and vertex, and then (ii) we scale the edge costs appropriately to yield the desired CCR.

Since the ETF algorithms have a complexity in $O(p^2|V|^2 + p|V||E|)$, they are not suited to million-node graphs that are included in the OCR and UFL datasets. Hence, we have selected a subset of OCR and UFL graphs, namely, graphs with 10k to 150k nodes, denoted as the *medium* dataset. The *big* dataset contains all graphs from Tables 3 and 4.

5.2 Communication-delay model vs. realistic model

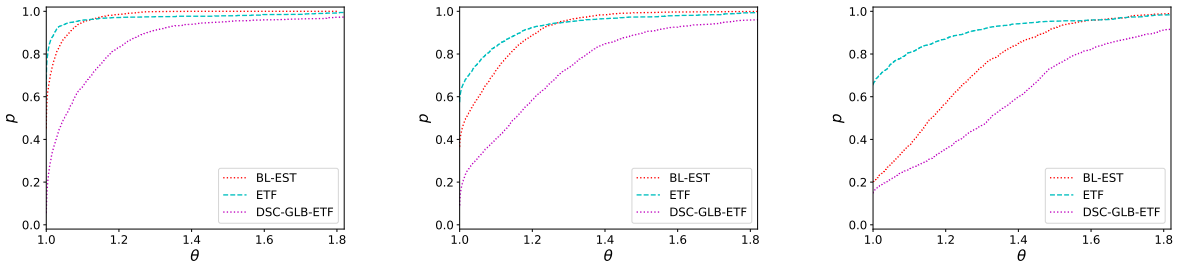
Our goal is to compare the new heuristics with the best competitors from the literature [26]. We call them the baseline heuristics, as they represent the current state-of-the-art. We have access to executables of the original implementation [26]. However, these heuristics assume a pure communication delay model, where communications can all happen at the same time, given that the task initiating the communications has finished its computation. Hence, there is no need to schedule the communications in this model.

In our work, we have assumed a more realistic, duplex single-port communication model. Thus, we cannot directly compare the new heuristics with the executables of the baseline heuristics. We have, therefore, implemented our own version of the baseline algorithms (BL-EST, ETF as best list-based and DSC-GLB-ETF as best cluster-based scheduler) with the communication delay model, and compared the resulting makespans with those of Wang and Sinnen's implementation, denoted as "ETF [W&S]", in an attempt to validate our implementations. We show the performance profiles in Figure 2 for this comparison. In the performance profiles, we plot the percentages of the instances in which a scheduling heuristic obtains a makespan on an instance that is no larger than θ times the best makespan found by any heuristic for that instance [4]. Therefore, the higher a profile at a given θ, the better a heuristic is. Results on Figure 2 confirm those presented by Wang and Sinnen: with low CCR (CCR=0.1 or CCR=1), DSC-GLB-ETF is worse than ETF (the higher the



(a) CCR=0.1 (b) CCR=1 (c) CCR=10

Figure 2 – Performance profiles comparing our implementation of baseline heuristics with Wang and Sinnen's implementation of ETF, on the *small* data set, with the communication-delay model.



(a) CCR=0.1 (b) CCR=1 (c) CCR=10

Figure 3 – Performance profiles comparing baselines on the *small* dataset, with the duplex single-port communication model.

better). However, when the CCR increases, the performance of DSC-GLB-ETF also increases, and it surpasses ETF for CCR=10 at the end [26].

Figure 2 also shows that our implementation of ETF performs better than ETF [W&S]. This may be due to tie-breaking in case of equal ordering condition, that we could not verify in detail since we had only the executables. Our implementation ETF is, thus, a fair competitor since it turns out to be better than the existing implementation.

Next, we converted our implementation of these algorithms into duplex single-port model, as explained in Section 4, in order to establish the baseline to compare the proposed heuristics. Figure 3 shows the performance profiles of our three baseline heuristics on the *small* dataset. From these results, we see that DSC-GLB-ETF is not well suited for the realistic communication model, since it performs pretty badly in comparison to ETF. BL-EST is also slightly worse than ETF, but it has a lower theoretical complexity.

5.3 Impact of number of parts, processors, CCR, and edge cut

In this section, we evaluate the impact of number of parts in the partitioning phase, number of processors, and CCR of datasets, and edge cut of the partitioner on the quality of the proposed heuristics.

Figure 4 depicts the relative performance of BL-EST-PART, BL-EST-BUSY, and BL-MACRO

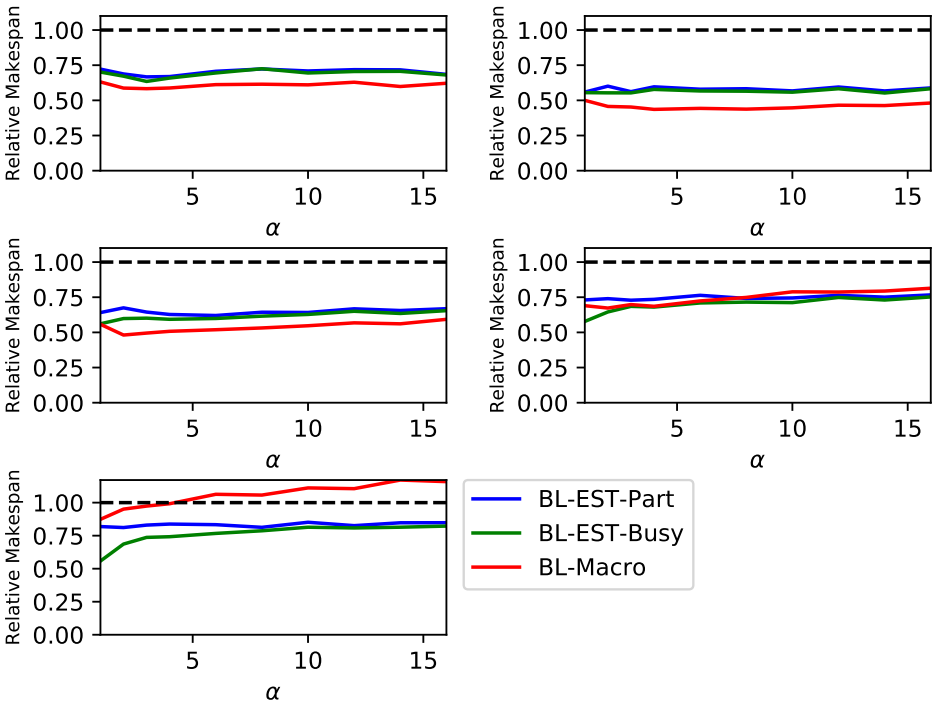


Figure 4 – Relative makespan compared to BL-EST on the *big* dataset, as a function of the number of parts, with CCR=10 and with 2 (top left), 4 (top right), 8 (middle left), 16 (middle right), and 32 (bottom left) processors.

compared to BL-EST on the *big* dataset as a function of α for different number of processors, $p = \{2, 4, 8, 16, 32\}$, and CCR=10. We set the number of parts $K = \alpha \times p$ and we have $\alpha = \{1, 2, 3, 4, 6, 8, 10, 12, 14, 16\}$. As seen in the figure, except BL-Macro on $p = 32$ processors, the new algorithms perform better than the baseline BL-EST for all values of α that we tested. Even for the worst case, that is, on 32 processors, BL-Macro performs better or comparable to BL-EST, when $\alpha \leq 4$. Therefore, we recommend to select $\alpha \leq 4$.

As shown in the previous studies (e.g., [26]), performance of the scheduling algorithms vary significantly with different CCRs, and in particular, clustering-based schedulers perform better for high CCRs, i.e., when communications are more costly than computations. Figure 5 shows the performance of the heuristics on the *big* dataset with varying CCR, i.e., for CCR=$\{1, 5, 10, 20\}$ and for $p = \{2, 4, 8, 16, 32\}$. The results are the average of all input instances using the best α value, for α=$\{1, 2, 3, 4\}$, for that instance.

As expected, similar to existing clustering-based schedulers, the proposed heuristics give significantly better results than the BL-EST baseline. For instance, when CCR=20, for all numbers of processors in the figure, all partitioning-based heuristics give at least 50% better makespans.

Comparing the relative performance of BL-EST-Part and BL-EST-Busy across the sub-figures, one observes that BL-EST-Part and BL-EST-Busy have more or less stable performance with the increasing number of processors. Note that the performance of BL-EST-Part and BL-EST-Busy mostly depends on the value of CCR, but remains the same when the number of processors varies. BL-Macro performs worse than the other two heuristics for small values of CCR with an increasing

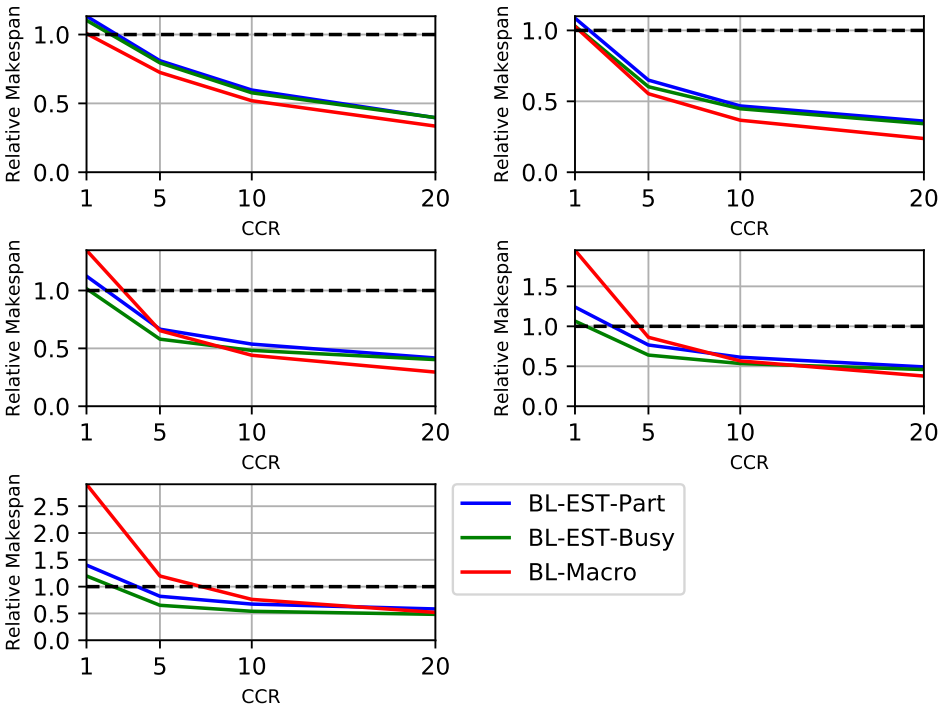


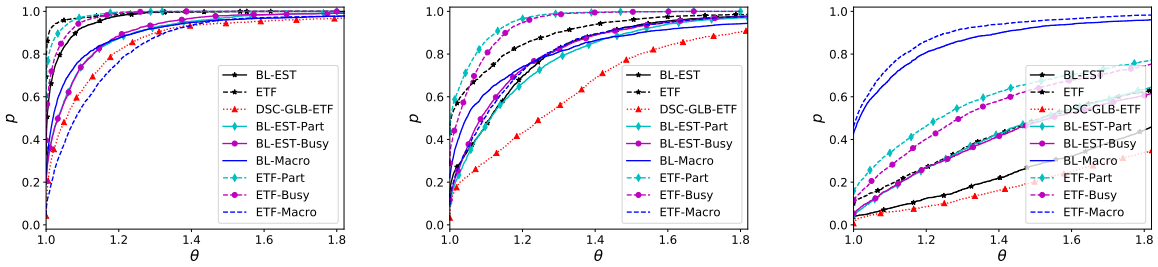
Figure 5 – Relative makespan compared to BL-EST on the *big* dataset, as a function of the CCR, with 2 (top left), 4 (top right), 8 (middle left), 16 (middle right), and 32 (bottom left) processors.

number of processors. However, for tested values of p, the performance of BL-MACRO improves as the CCR increases, and finally it outperforms all other heuristics on average when the CCR is large enough.

We have carried out thorough experiments to see the effects of the edge cut of DAG partitioning in the final makespan. We observed that having a smaller edge cut in DAG partitioning yields a better makespan more than 82% of the time for all proposed heuristics, when the communication-to-computation ratio (CCR) is 10. Indeed, on the *small* dataset, we counted the instances where a better edge cut in partitioning gave a better makespan. Out of 9045 instances, there were 7494 such instances for *-MACRO, 7519 for *-PART, and 7477 for *-BUSY, hence ranging between 82.6% and 83.1%.

5.4 Runtime comparison and performance results

We present the results on the *small*, *medium* and *big* datasets. We focus only on the BL-EST algorithm for the *big* dataset, since ETF does not scale well (due to quadratic time complexity on the number of vertices), and DSC-GLB-ETF shows poor results with the realistic communication model and smaller datasets. Let us consider `XSBench` graph as an example of how long it takes to run ETF on one of the *big* graphs. When we schedule this graph on two processors, the DAG partitioning algorithm runs in 9.5 seconds on average, and BL-EST-PART, BL-EST-BUSY, and BL-MACRO heuristics run under half a second to totalize approximately 10 seconds. However, ETF algorithm takes 4759 seconds. On four processors, it goes up to 7507 seconds.



(a) CCR=0.1 (b) CCR=1 (c) CCR=10

Figure 6 – Performance profiles comparing all the algorithms on the *small* dataset with duplex single-port model.

Small dataset. Figure 6 shows the comparison of all heuristics on the *small* dataset for CCR=$\{0.1, 1, 10\}$. While ETF remains the best with a small CCR=0.1, ETF-PART becomes better as soon as CCR=1. Finally, the performance of BL-MACRO and ETF-MACRO is striking for CCR=10, where the *-MACRO variant clearly outperforms all other heuristics.

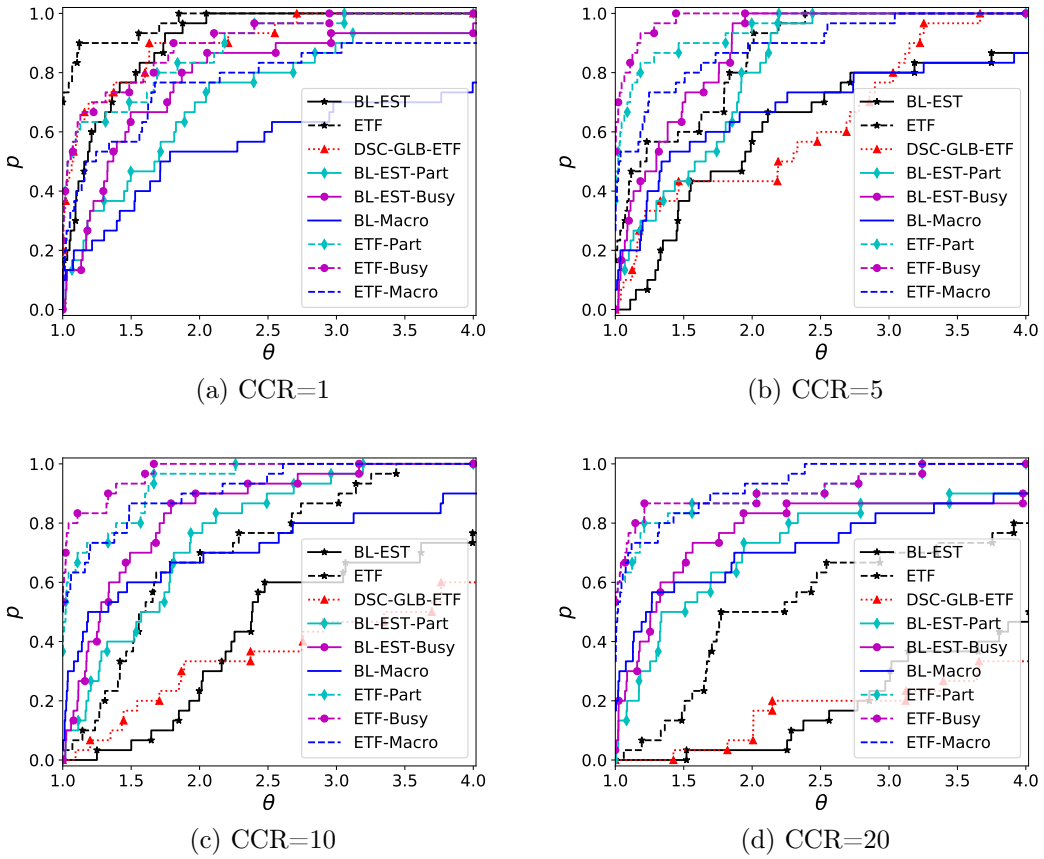
As seen before, DSC-GLB-ETF performs poorly in this case, since it is not designed for realistic duplex single-port communication model.

Medium dataset. Figure 7 shows the performance profiles on the *medium* dataset for CCR=$\{1, 5, 10, 20\}$. As expected, dynamic scheduling technique ETF and our ETF-based heuristics perform better than their BL-EST counterparts, as for the *small dataset*. Note that our heuristics perform better than the original versions they are built upon, except for CCR=1.

ETF and ETF-based algorithms' quality comes with a downside of high theoretical complexity and consequently, slower algorithms due to their dynamic nature. Figure 8 shows runtime performance profiles. It is therefore the fraction of instances in which an algorithm gave a runtime no worse than the fastest algorithm, hence the higher the better. As expected, the static BL-EST approach runs much faster than dynamic approaches. DAG-partitioning introduces an overhead to proposed heuristics, but this is still negligible compared to the theoretical complexity of the algorithms. BL-EST-PART, BL-EST-BUSY, and BL-MACRO heuristics also perform comparably fast even with partitioning time overhead. ETF and ETF-based heuristics run two to three orders of magnitude slower, making them infeasible to run them on bigger graphs.

Big dataset. Table 5 displays the detailed results on the *big* dataset, with two processors, for CCR in $\{1, 5, 10, 20\}$. On average, BL-EST-BUSY provides slightly better results than BL-EST-PART. When CCR=1, the heuristics often return a makespan that is slightly larger than the one from BL-EST, on average by 13%, 11%, and 2%, respectively. When CCR=5, BL-EST-PART, BL-EST-BUSY, and BL-MACRO provide 20%, 22%, and 29% improvement compared to BL-EST, on average on the whole *big* dataset when considering an architecture with two processors. When $CCR = 10$, these values become respectively 42%, 44%, and 49% (1.7, 1.8, and 2 times smaller, respectively). For CCR=20, BL-EST-PART and BL-EST-BUSY obtain a makespan 2.8 times smaller than the baseline, and it goes up to 3.1 times smaller for BL-MACRO.

Figure 9 shows the performance profile of these four algorithms for CCR=$\{1, 5, 10, 20\}$. When



(a) CCR=1 (b) CCR=5

(c) CCR=10 (d) CCR=20

Figure 7 – Performance profiles on *medium* dataset, with CCR=$\{1, 5, 10, 20\}$.

CCR=1, BL-EST performs best but BL-EST-BUSY performs very close to it. However, when the value of CCR is increasing, it is more and more important to handle communications correctly. We observe that the proposed three heuristics outperform the baseline (BL-EST) as the CCR increases. When CCR=5, in about 90% of all cases BL-EST-BUSY's makespan is within 1.5× of the best result, whereas this fraction is only 40% for BL-EST. Starting with CCR=10, the proposed heuristics completely dominate BL-EST algorithm. For all values of CCRs, BL-EST-BUSY outperforms BL-EST-PART. BL-MACRO, on the other hand, performs worse than BL-EST-PART and BL-EST-BUSY when CCR=1, and gradually outperforms the other two as the CCR increases.

To understand the nature of datasets where the proposed heuristics and the baseline behave differently, we divided the *big* dataset into two subsets, the graphs consisting of more than 10% of the nodes as sources, and the ones with less than 10%. Figures 10 and 11 show how the algorithms' quality differ for these subsets. With a lot of sources (Figure 11), BL-EST baseline performs badly while BL-MACRO performs better than with fewer sources. This is due to the inherent nature of DAG-partitioning followed by cluster-by-cluster scheduling. Consider a DAG of clusters with one source cluster. BL-MACRO would need to schedule all of the nodes in this cluster in one processor to start utilizing any other processor available. When the number of source clusters is high, this heuristic can start efficiently using more processors right from the start.

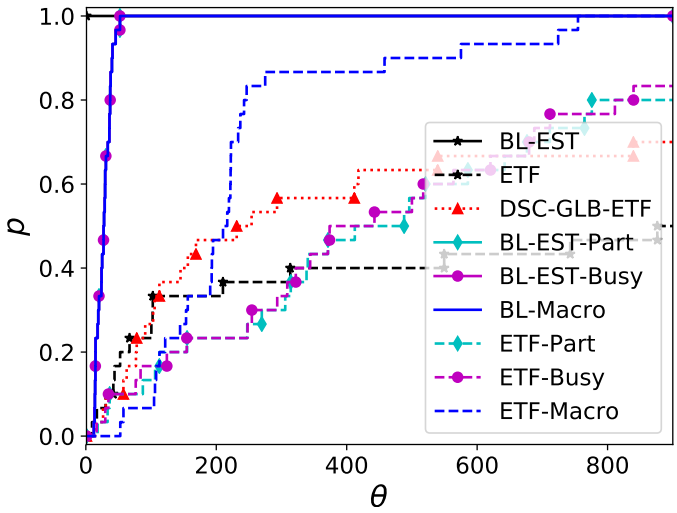


Figure 8 – Performance profiles of the runtime on *medium* dataset, with CCR=10.

In all simulations, the running times of BL-EST, BL-EST-PART, BL-EST-BUSY, and BL-MACRO are equivalent and negligible compared to the running time of the partitioning algorithm, which is in the order of seconds.

5.5 Summary

Overall, the proposed meta-heuristics significantly improve the makespan found by the baseline heuristics they are applied on, as empirically shown with a wide range of graph instances. The results confirm the correlation between the edge cut found during the partitioning phase and the makespan at the end. The benefit of a good partitioning with minimum edge cut objective shows itself clearly, especially when the CCR is high.

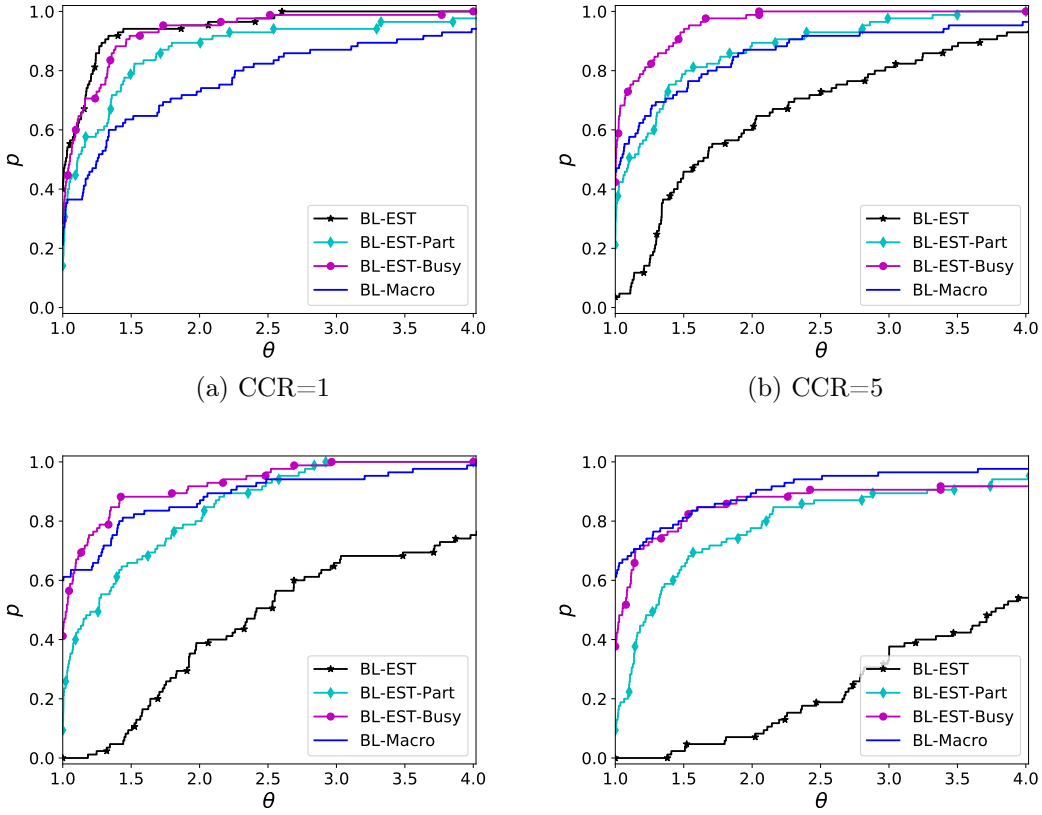
The results show that *-PART and *-BUSY behave consistently, and that they provide a steady improvement over the baselines. Furthermore, their relative performance (compared to the baseline) does not depend on the number of processors, which means that these heuristics scale well. They perform even better when the ratio between communication and computation is large.

The *-MACRO's performance has a higher variance. This meta-heuristic tries to have more of a "global" view during scheduling, by tentatively scheduling whole parts instead of deciding the mapping when it is only at the first node of the part and dictating the rest (as done by *-PART and *-BUSY). It 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. Also, the experiments show that when the input instance to be scheduled has higher percentage of sources (source parts), *-MACRO is even more likely to outperform other heuristics.

	CCR=1				CCR=5			
Graph	BL-EST	BL-EST-PART	BL-EST-BUSY	BL-MACRO	BL-EST	BL-EST-PART	BL-EST-BUSY	BL-MACRO
598a	3058476	1.14	1.14	1.04	5857127	0.62	0.62	0.57
caidaRouterLevel	5337718	1.02	1.02	1.00	8937548	0.95	0.95	0.80
delaunay-n17	3606092	1.02	1.03	1.00	5431960	0.69	0.69	0.67
email-EuAll	7711619	1.00	1.00	0.98	18123055	0.44	0.44	0.44
fe-ocean	3949464	1.12	1.12	1.02	5185419	0.86	0.86	0.78
ford2	2781775	1.03	1.03	0.99	4024990	0.70	0.70	0.69
luxembourg-osm	3152973	1.01	1.01	1.00	3506686	0.90	0.90	0.90
rgg-n-2-17-s0	3601079	1.23	1.23	1.06	4585262	0.91	0.91	0.83
usroads	3550396	1.02	1.02	1.02	4097201	0.97	0.91	0.88
vsp-mod2-pgp2-slptsk	2794636	1.04	1.04	1.00	5509790	0.67	0.67	0.64
cholesky	30603433	1.28	1.03	0.95	49102625	0.82	0.65	0.60
fibonacci	34601228	1.11	1.10	1.03	44109081	0.89	0.89	0.81
quicksort	54162227	1.01	1.01	1.00	71605033	0.76	0.76	0.76
RSBench	26941941	1.38	1.25	0.88	45191117	0.84	0.78	0.53
Smith-waterman	1661676	1.46	1.41	1.02	2196692	1.11	1.02	0.78
UTS	31904401	1.34	1.34	1.34	51957000	0.83	0.83	0.83
XSBench	41794985	1.15	1.15	1.02	49993817	0.97	0.97	0.87
Geomean	**1.00**	**1.13**	**1.11**	**1.02**	**1.00**	**0.80**	**0.78**	**0.71**

	CCR=10				CCR=20			
Graph	BL-EST	BL-EST-PART	BL-EST-BUSY	BL-MACRO	BL-EST	BL-EST-PART	BL-EST-BUSY	BL-MACRO
598a	9669102	0.38	0.38	0.38	17038485	0.22	0.22	0.22
caidaRouterLevel	14638583	0.85	0.85	0.58	26745328	0.56	0.56	0.35
delaunay-n17	9216833	0.40	0.40	0.39	17567627	0.22	0.22	0.21
email-EuAll	32997285	0.34	0.34	0.34	67066585	0.19	0.19	0.19
fe-ocean	7202636	0.62	0.62	0.56	11573357	0.39	0.39	0.35
ford2	6068545	0.47	0.47	0.45	10538479	0.27	0.27	0.26
luxembourg-osm	3941446	0.81	0.81	0.80	4801062	0.66	0.66	0.66
rgg-n-2-17-s0	5892674	0.73	0.73	0.66	9094485	0.48	0.48	0.43
usroads	5327111	0.67	0.67	0.67	8428888	0.43	0.43	0.42
vsp-mod2-pgp2-slptsk	9460442	0.59	0.49	0.45	19887584	0.28	0.46	0.23
cholesky	75676369	0.53	0.49	0.39	130153391	0.31	0.24	0.23
fibonacci	64454756	0.61	0.61	0.55	110167490	0.36	0.35	0.32
quicksort	104478680	0.52	0.52	0.52	173055640	0.32	0.32	0.31
RSBench	67674107	0.59	0.47	0.36	109245784	0.38	0.30	0.24
Smith-waterman	3408415	0.79	0.71	0.50	5694549	0.53	0.44	0.33
UTS	74335883	0.58	0.58	0.58	117598932	0.40	0.41	0.37
XSBench	59646365	0.83	0.82	0.75	77257208	0.64	0.63	0.60
Geomean	**1.00**	**0.58**	**0.56**	**0.51**	**1.00**	**0.36**	**0.36**	**0.32**

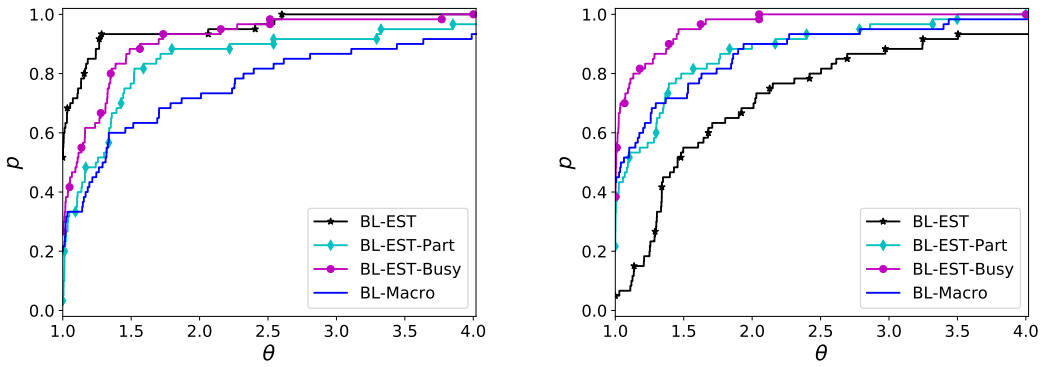
Table 5 – The makespan of BL-EST in absolute numbers, and those of BL-EST-PART, BL-EST-BUSY, and BL-MACRO relative to BL-EST on *big* dataset, when the number of processors p is 2, and for CCR in $\{1, 5, 10, 20\}$.



(a) CCR=1 (b) CCR=5

(c) CCR=10 (d) CCR=20

Figure 9 – Performance profiles on *big* dataset, with CCR=$\{1, 5, 10, 20\}$.



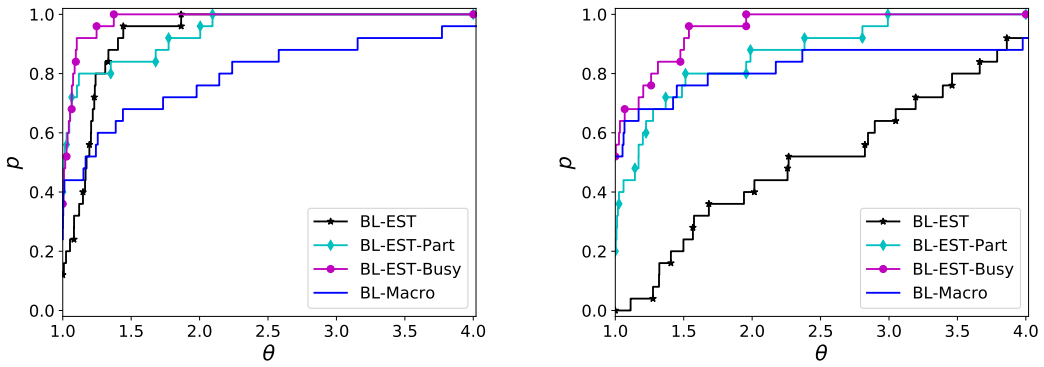
(a) CCR=1

(b) CCR=5

(c) CCR=10

(d) CCR=20

Figure 10 – Performance profiles on *big* dataset when less than 10% of nodes are sources, with CCR=$\{1, 5, 10, 20\}$.



(a) CCR=1 (b) CCR=5

(c) CCR=10 (d) CCR=20

Figure 11 – Performance profiles on *big* dataset when more than 10% of nodes are sources, with CCR=$\{1, 5, 10, 20\}$.

6 Conclusion

We proposed three new partitioning-assisted list-based scheduling techniques (or meta-heuristics) based on an acyclic partition of the DAGs: *-Part, *-Busy, and *-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.

To the best of our knowledge, this is the first partitioning-assisted list-scheduler using a multi-level *directed* DAG partitioner for the clustering phase. The acyclicity is well suited to identify data locality in the DAG, and it allows the design of specific allocation strategies, such as *-Macro. The proposed meta-heuristics are generic and can be combined with any classical list-scheduling heuristic, and used with any acyclic partitioner.

We compared our scheduling techniques with the widely used bl-est, etf, and dsc-glb-etf heuristics, adapted to the realistic duplex single-port communication model. The results are striking, with the new heuristics consistently improving the makespan. Even though *-Macro does not seem to scale well with the number of processors, it delivers the best results in several cases, while *-Part and *-Busy are consistently good. For instance, the proposed *-Part (resp. *-Busy and *-Macro) algorithms achieve a makespan 2.6 (resp. 3.1 and 3.3) times smaller than bl-est when considering the *big* dataset with $CCR = 20$, averaging over all processor numbers. Furthermore, if we pick the best of the three heuristics for each instance, it is four times better.

Our experiments suggest that the relative performance of bl-est-Part and bl-est-Busy (etf-Part and etf-Busy) compared to the baseline bl-est (etf) does not depend on the number of processors, which means that these heuristics scale well. They provide a steady improvement over the classic bl-est (etf) heuristic and they perform even better when the ratio between communication and computation is large. bl-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.

As future work, we plan to consider *convex* partitioning instead of acyclic partitioning, which we believe will enable more parallelism. The existing clustering techniques in the scheduling area can also be viewed as local algorithms for convex partitioning. To the best of our knowledge, there is no top-down convex partitioning technique available, which we plan to investigate. Also, an adaptation of the proposed heuristics to heterogeneous processing systems would be needed. A difficulty arises in addressing the communication cost, which also requires updating the partitioner.

Acknowledgment

We thank Oliver Sinnen for providing us the Java binaries of their implementation and the datasets they used in their studies [26], and the referees for valuable feedback.

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 T. Parall. Distr.*, 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.

[4] E. D. Dolan and J. J. Moré. Benchmarking optimization software with performance profiles. *Mathematical programming*, 91(2):201–213, 2002.

[5] 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.

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

[7] A. Gerasoulis and T. Yang. On the granularity and clustering of directed acyclic task graphs. *IEEE T. Parall. Distr.*, 4(6):686–701, June 1993.

[8] 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.

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

[10] C. Hanen and A. Munier. An approximation algorithm for scheduling dependent tasks on m processors with small communication delays. In *Proceedings 1995 INRIA/IEEE Symposium on Emerging Technologies and Factory Automation. ETFA'95*, volume 1, pages 167–189 vol.1, Oct 1995.

[11] 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.

[12] 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.

[13] K. Jansen, F. Land, and M. Kaluza. Precedence scheduling with unit execution time is equivalent to parametrized biclique. In R. M. Freivalds, G. Engels, and B. Catania, editors, *SOFSEM 2016: Theory and Practice of Computer Science*, pages 329–343, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[14] H. Kanemitsu, M. Hanada, and H. Nakazato. Clustering-based task scheduling in a large number of heterogeneous processors. *IEEE T. Parall. Distr.*, 27(11):3144–3157, Nov 2016.

[15] Y.-K. Kwok and I. Ahmad. Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors. *IEEE T. Parall. Distr.*, 7(5):506–521, 1996.

[16] 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.

[17] Y.-K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Comput. Surv.*, 31(4):406–471, 1999.

[18] S. Mingsheng, S. Shixin, and W. Qingxian. An efficient parallel scheduling algorithm of dependent task graphs. In *Proc. of 4th Int. Conf. on Parallel and Distributed Computing, Applications and Technologies, PDCAT*, pages 595–598. IEEE, 2003.

[19] C. Picouleau. New complexity results on scheduling with small communication delays. *Discrete Applied Mathematics*, 60(1):331 – 342, 1995.

[20] A. Radulescu and A. J. Van Gemund. Low-cost task scheduling for distributed-memory machines. *IEEE T. Parall. Distr.*, 13(6):648–658, 2002.

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

[22] G. C. Sih and E. A. Lee. A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. *IEEE T. Parall. Distr.*, 4(2):175–187, 1993.

[23] O. Sinnen. *Task Scheduling for Parallel Systems.* Wiley Series on Par.and Distr. Computing, Wiley-Interscience, New York, NY, USA, 2007.

[24] O. Svensson. Conditional hardness of precedence constrained scheduling on identical machines. In *Proceedings of the Forty-second ACM Symposium on Theory of Computing*, STOC'10, pages 745–754, New York, NY, USA, 2010. ACM.

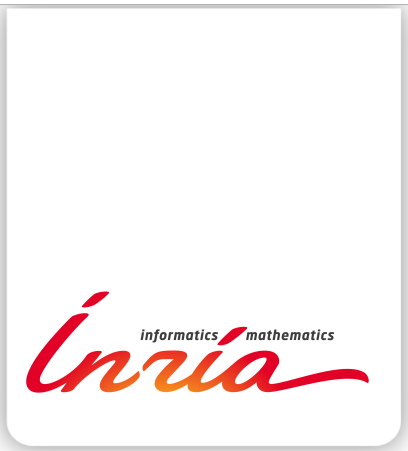
[25] H. Topcuoglu, S. Hariri, and M. Y. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE T. Parall. Distr.*, 13(3):260–274, 2002.

[26] H. Wang and O. Sinnen. List-scheduling vs. cluster-scheduling. *IEEE T. Parall. Distr.*, 2018. in press.

[27] M.-Y. Wu and D. D. Gajski. Hypertool: A programming aid for message-passing systems. *IEEE T. Parall. Distr.*, 1(3):330–343, 1990.

[28] T. Yang and A. Gerasoulis. DSC: Scheduling parallel tasks on an unbounded number of processors. *IEEE T. Parall. Distr.*, 5(9):951–967, 1994.

[29] L. Yu and V. Sarkar. GT-Race: Graph traversal based data race detection for asynchronous many-task runtimes. In *Euro-Par 2018: Parallel Processing.* Springer, 2018.



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