

Efficient Multi-Processor Scheduling in Increasingly Realistic Models

Pál András Papp

pal.andras.papp@huawei.com
Computing Systems Lab
Huawei Zurich Research Center
Zurich, Switzerland

Aikaterini Karanasiou

aikaterini.karanasiou@huawei.com
Computing Systems Lab
Huawei Zurich Research Center
Zurich, Switzerland

Georg Anegg

georg.anegg@huawei.com
Computing Systems Lab
Huawei Zurich Research Center
Zurich, Switzerland

Albert-Jan N. Yzelman

albertjan.yzelman@huawei.com
Computing Systems Lab
Huawei Zurich Research Center
Zurich, Switzerland

ABSTRACT

We study the problem of efficiently scheduling a computational DAG on multiple processors. The majority of previous works have developed and compared algorithms for this problem in relatively simple models; in contrast to this, we analyze this problem in a more realistic model that captures many real-world aspects, such as communication costs, synchronization costs, and the hierarchical structure of modern processing architectures. For this we extend the well-established BSP model of parallel computing with non-uniform memory access (NUMA) effects. We then develop a range of new scheduling algorithms to minimize the scheduling cost in this more complex setting: several initialization heuristics, a hill-climbing local search method, and several approaches that formulate (and solve) the scheduling problem as an Integer Linear Program (ILP). We combine these algorithms into a single framework, and conduct experiments on a diverse set of real-world computational DAGs to show that the resulting scheduler significantly outperforms both academic and practical baselines. In particular, even without NUMA effects, our scheduler finds solutions of 24% – 44% smaller cost on average than the baselines, and in case of NUMA effects, it achieves up to a factor 2.5× improvement compared to the baselines. Finally, we also develop a multilevel scheduling algorithm, which provides up to almost a factor 5× improvement in the special case when the problem is dominated by very high communication costs.

CCS CONCEPTS

• **Theory of computation** → **Scheduling algorithms**; Parallel computing models; Integer programming.

KEYWORDS

DAG scheduling; BSP model; Integer Linear Programming; Multi-level scheduling

1 INTRODUCTION

The optimal scheduling of complex computational workloads on multiple processors has been extensively studied since the early days of computing. As the different subtasks in a computation usually have precedence constraints between them (one subtask can only be started after another one has been finished), the computations in these applications are often modeled as Directed Acyclic Graphs (DAGs), with the directed edges representing these dependencies.

Besides the structure of the DAG representing the computation, the other main ingredient of a scheduling problem is the machine model assumed. DAG scheduling has been studied in a very wide range of models in the past decades, differing in e.g. the numbers and types of processors, the modelling of communication costs, or other properties of the system such as synchronization or preemption. However, for arbitrary computational workloads (i.e. general DAGs), finding an optimal or close-to-optimal schedule is already very challenging in the simplest possible models, e.g. with uniform processors and very simple models of communication (or even no communication costs at all).

Due to this, previous research has mostly focused on variants of the scheduling problem where either the set of input DAGs or the scheduling model is heavily restricted or simplified. Theoretical work has almost exclusively studied very restricted models, where it is still manageable to analyze approximation algorithms or hardness results; however, these models are usually far from realistic. Experimental research has developed several scheduling heuristics based on natural ideas such as locality, that generally deliver good empirical results in simpler models; however, these heuristics are completely impervious to the parameters of real-world systems, and hence it is unclear how their performance carries over to more complex models. On the practical side, the parallel computing community has developed more sophisticated models, such as the Bulk Synchronous Parallel (BSP) or the LogP model, that capture many aspects of a real-world system; however, due to their complexity, these models have mostly been used for developing and analyzing parallel implementations of concrete computations (i.e. specific DAGs), instead of general computational tasks. As such, while each of these directions offers valuable insight, we still lack a proper

©Pál András Papp, Georg Anegg, Aikaterini Karanasiou and Albert-Jan N. Yzelman, 2024. This is the author's full version of the work, posted here for personal use. Not for redistribution. The definitive version (extended abstract) was published in the 36th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2024), <https://doi.org/10.1145/3626183.3659972>.

understanding of how to devise efficient scheduling algorithms for arbitrary DAGs in a realistic machine model.

Our goal in this paper is to develop and analyze scheduling algorithms for this more general case: we consider the scheduling problem for general DAGs, in a model that captures many of the most important aspects of real computations. In particular, we begin from the BSP model which already accounts for communication volume and latency, and we extend this with a simple model of non-uniform memory access (NUMA) effects, which also have a defining role in modern manycore architectures. We then develop a framework of scheduling algorithms in this more realistic model, and compare this against several state-of-the-art scheduling algorithms from previous research and from applications, on a set of DAGs representing a wide range of actual computational tasks.

Our first contribution in the paper is a database of computational DAGs, which offers a diverse benchmark to evaluate different scheduling algorithms in the future. Then in terms of algorithms, we first present some heuristic methods to develop initial schedules (similar to heuristics from previous works, but tuned towards our realistic model). We then develop more advanced scheduling algorithms that take a different approach: instead of making heuristic decisions based on e.g. locality, they consider the actual cost of each solution in our complex model (with all its parameters), and directly optimize for minimizing this cost. In particular, we develop (i) a local search algorithm that iteratively improves a solution by making small modifications to a schedule as long as this decreases its total cost, and (ii) several different ways to model the problem as an Integer Linear Program (ILP), and use an ILP solver to find (sub)schedules of small cost. Finally, we also present a multilevel scheduling approach that is superior in the case when our scheduling problem is substantially dominated by communication costs.

Unsurprisingly, our more complex algorithms require notably more running time than lightweight heuristics, and hence their study is not viable on very large graphs. On the other hand, in the domain where they are applicable, our results show that they significantly outperform the baseline heuristics. In particular, even without NUMA effects, our scheduler achieves a cost reduction of 32%–51% compared to a simpler, and 13%–40% compared to a more sophisticated baseline, depending on the number of processors and the parameters of the model, on DAGs up to 10 000 nodes. In case of NUMA effects, the improvement is even larger: 48%–71% to the simpler, and 27%–58% to the stronger baseline, or with the multilevel algorithm, even up to 79% (i.e. almost a factor 5× improvement) to the stronger baseline in some cases. This shows that in realistic scheduling models, our approach can indeed return drastically better solutions than the baseline algorithms.

2 RELATED WORK

Different variants of the DAG scheduling problem have been studied thoroughly since the 1960s. In the first decades, research has mostly focused on very simple settings that either do not consider communication costs (motivated by the PRAM model), or only capture it as a fixed delay, independently of communication volume. In terms of theoretical results, scheduling is known to be already NP-hard in these simple models, and on special subclasses of DAGs

[12, 18, 30, 35, 38]. On the other hand, the topic of developing approximation algorithms in these models is still extensively studied in recent years [16, 20–22].

On the experimental side, researchers have developed a range of different scheduling heuristics, and evaluated their performance empirically. Recent surveys [41] have classified these heuristics into two groups: list-based [2, 13, 26, 32, 36] and cluster-based [14, 17, 42] methods. To our knowledge, there is very little previous work on evaluating these heuristics in more realistic models (with the exception of [27], which is discussed separately below).

On the more practical side, BSP is one of the prominent models of parallel computing, and has been studied extensively from various perspectives [4, 23, 34, 39, 40]. The BSP model accounts for several important real-world aspects such as communication volume and latency, but it conveniently captures all these in a relatively simple cost function; as such, it is also used in various applications via the BSPlib library and its implementations [11, 43, 44]. However, most of the previous works on BSP (and on similar models with more parameters, such as LogP [6]) focus on the scheduling of specific DAGs that describe a concrete computation [8, 24, 25]. One exception to this is a recent theoretical work that studies the computational complexity of BSP scheduling for given subclasses of DAGs, and also presents a naive ILP formulation to capture the BSP scheduling problem on general DAGs [28].

The impact of NUMA in an algorithmic context has also been studied before on e.g. different variants of the (hyper)graph partitioning problem [9, 29, 33].

Numerous other variants of the scheduling problem have also been studied extensively, e.g. with separate deadlines for nodes or with node duplication [3, 19].

The closest result our work is the work of Özkaya et al. [27], who conduct a similar experimental study with the same goal of analyzing sophisticated scheduling algorithms in a more realistic model. Our work differs from their approach in several aspects. Firstly, the work of [27] introduces a custom model (named duplex single-port model) that only extends classical scheduling models with communication volume; in contrast to this, our work considers BSP, a well-established parallel computing model in both theory and practice which also captures further aspects such as latency, and we further extend this with NUMA effects. Moreover, while the main idea of [27] is to extend modern list schedulers with a acyclic partitioner, we take an entirely different approach, and focus on methods that aim to directly minimize the cost function (such as local search and ILP representations).

3 PROBLEM DEFINITION AND BACKGROUND

3.1 Preliminaries

We assume that our computation is represented by Directed Acyclic Graph (DAG) $G(V, E)$. The nodes of the DAG correspond to subtasks or operations we need to execute, and the directed edges correspond to dependencies between these operations: an edge from node u to node v implies that the execution of u has to be finished before the execution of v begins, because the output of u is required as an input for v . We denote the number of nodes by n .

Besides the structure of the DAG, our computation is described by two parameters for each node v : the computation weight $w(v)$

(also called *work weight*) is the amount of time required to execute operation v on a processor, and the *communication weight* $c(v)$ is the amount of communication required to send the output of v to another processor (e.g. its size in bytes). These weights can differ significantly between the different nodes, so they are both crucial to include in our model. Note that we consider the communication weight $c(v)$ to be a property of each node of the DAG (in contrast to e.g. [27], where it is assigned to the edges). For simplicity, we assume that node weights are integers.

3.2 The BSP model

The BSP model assumes that the computational steps are organized into so-called *supersteps*. Each superstep consists of the following two phases:

- (1) *Computation phase*: each processor can execute an arbitrary amount of computation, but no communication is allowed.
- (2) *Communication phase*: processors can communicate any number of values to each other, but no computation happens.

Intuitively, supersteps correspond to larger batches of computations that are executed consecutively on a single processor, without any interruption for communication. Dividing the computations into such batches is often beneficial in practice, since the communication often comes with a significant overhead that is independent of the number of communicated values, due to e.g. synchronization or network initialization.

Our computing architecture in BSP is described by three parameters: the number P of processors available, the time cost g of sending a single unit of data between processors, and a fixed overhead cost ℓ (called the *latency*) incurred by each superstep.

When applying the BSP model to DAG scheduling, our schedule must respect the dependencies described by the edges of the DAG. This means that a node v can only be computed on processor p in superstep s if the output values from all its direct predecessors are already present on p : that is, they were either computed on p in an earlier (or the same) superstep, or they were sent to p by another processor before superstep s . In the communication phases, a processor p can send the output value of any node v if it is already present on p , to any other processor(s).

Formally, a BSP schedule of a DAG consists of (i) an assignment of nodes to processors $\pi : V \rightarrow \{1, \dots, P\}$ and supersteps $\tau : V \rightarrow \mathbb{N}$, and (ii) a communication schedule Γ , i.e. a set of 4-tuples (v, p_1, p_2, s) , indicating that the output of node v is sent from processor p_1 to processor p_2 in the communication phase of superstep s . A valid BSP schedule then must satisfy the conditions discussed above, i.e.

- For each edge (u, v) of G , in case of $\pi(u) = \pi(v)$, we must have $\tau(u) \leq \tau(v)$, and in case of $\pi(u) \neq \pi(v)$, we must have an entry $(u, p_1, \pi(v), s) \in \Gamma$ for some processor p_1 and some superstep $s < \tau(v)$.
- For each $(v, p_1, p_2, s) \in \Gamma$, we must either have $\pi(v) = p_1$ and $\tau(v) \leq s$, or we must have another $(v, p', p_1, s') \in \Gamma$ with $s' < s$.

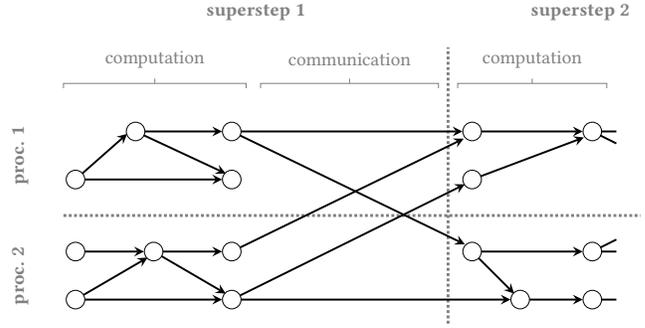


Figure 1: Example BSP scheduling of a DAG.

An example for a BSP scheduling of a DAG is illustrated in Figure 1. In the computation phase, processors 1 and 2 execute 4 and 5 operations (nodes), respectively; then in the communication phase, processor 1 needs to send one value to processor 2, while processor 2 needs to send two values to processor 1 (to make these available on the given processors for superstep 2). After this, the computation phase of superstep 2 can begin.

3.3 Cost in BSP

Another advantage of BSP is that in contrast to classical models, BSP assigns a simple cost metric to each superstep, and the total time required (i.e. total cost of a schedule) can be obtained by simply summing up the costs of the individual supersteps.

In the computation phase of each superstep, the subtasks assigned to each processor are executed by the processors simultaneously; as such, the cost of a computation phase is defined as the maximum work assigned to any processor in the superstep, i.e.

$$C_{work}(s) = \max_{p \in \{1, \dots, P\}} \sum_{\substack{\pi(v)=p \\ \tau(v)=s}} w(v).$$

On the other hand, communication costs are measured by the so-called *h-relation* metric, where the cost of a communication phase is determined by the maximum amount of data sent or received by any processor. That is, the send cost and receive cost, respectively, of processor p in superstep s is

$$C_{send}(p, s) = \sum_{\substack{(v, p_1, p_2, s_1) \in \Gamma \\ p_1=p \\ s_1=s}} c(v),$$

$$C_{rec}(p, s) = \sum_{\substack{(v, p_1, p_2, s_1) \in \Gamma \\ p_2=p \\ s_1=s}} c(v),$$

and the communication cost of superstep s is

$$C_{comm}(s) = \max_{p \in \{1, \dots, P\}} \max(C_{send}(p, s), C_{rec}(p, s)).$$

The total cost of superstep s is then the sum of the work, communication, and latency costs, defined as

$$C(s) = C_{work}(s) + g \cdot C_{comm}(s) + \ell,$$

and the cost of the whole schedule is obtained by simply summing up $C(s)$ for all the supersteps.

For more details on the properties and the different variants of this model, we refer the reader to [28].

3.4 NUMA effects

BSP already captures communications more realistically than most classical models. However, one significant drawback of BSP is that it assumes a uniform communication cost between any pair of processors. In contrast to this, today’s computing architectures very often exhibit a hierarchical structure: each processor has multiple cores, each machine has multiple processors, and maybe multiple machines are connected over a network. This hierarchical structure results in a non-uniform memory access setting, where the cost of sending a piece of data heavily depends on the concrete pair of processing units that communicate: sending a value between two cores on the same processor is relatively low, whereas sending the same data over the highest level of the hierarchy (e.g. over the network) is drastically higher.

This asymmetry between the processors is often a defining aspect of the scheduling problem in practice. Due to this, we also extend the BSP model which such NUMA effects, and further analyze the impact of this in our experiments. The BSP definition above is straightforward to extend to this setting: if the cost λ_{p_1, p_2} of communicating a single unit of data is known for each pair of processors (p_1, p_2) , then we can simply add λ_{p_1, p_2} as a further factor in the formulas defining $C_{send}(p, s)$ and $C_{rec}(p, s)$ above. The values λ_{p_1, p_2} then become further input parameters of the problem; we can specify them either directly for each pair of processors, or implicitly through a hierarchy. Note that the default case of uniform communication costs corresponds to a choice of $\lambda_{p_1, p_2} = 1$ for $p_1 \neq p_2$, and $\lambda_{p_1, p_2} = 0$ for $p_1 = p_2$.

3.5 Problem definition

Altogether, the input of our problem is (i) a DAG with node weights $w(v)$ and $c(v)$, and (ii) a machine description with the parameters P , g and ℓ , and possibly λ_{p_1, p_2} for all pairs of processors p_1, p_2 . Our goal is to find the best valid schedule in the BSP model, minimizing the total cost.

We sometimes also discuss the *communication scheduling* subproblem of optimizing Γ when π and τ are already fixed, i.e. sorting the necessary communication steps into the given communication phases while minimizing the total cost of h -relations. With the assignment to processors and supersteps fixed, this subproblem has a much smaller degree of freedom; its theoretical complexity was studied separately in [28].

4 SCHEDULING ALGORITHMS

We now describe our scheduling algorithms in the BSP model. Due to space constraints, we only outline the main ideas behind each algorithm; we discuss them in more detail in Appendix A. The pipeline combining the algorithms is discussed later in Section 6.

4.1 Baselines

In order to evaluate our approach, we use the following baseline schedulers for comparison:

- **Cilk**: this is a simple and yet efficient scheduling heuristic [5]; different variants of the same work-stealing approach are widely used in many of today’s prominent parallel programming libraries and frameworks. While Cilk was originally not defined on DAGs, it is easy to adapt to this case. Intuitively, Cilk maintains a stack of ready tasks for each processor to work on, and if a processor is idle (its stack is empty), it “steals” a subtask from the bottom of a (randomly chosen) other processor’s stack. We use Cilk to represent the baseline from the practical/application side.
- **BL-EST and ETF**: recent comparison studies have found that the best scheduling algorithms are list-based schedulers [27, 41]. In particular, the so-called BL-EST and ETF schedulers were found to be most efficient in earlier experiments; they have also already been adapted to a setting with communication volume [27]. When scheduling the next node, both BL-EST and ETF assign the node to the processor that offers the earliest start time, based on previous assignments and necessary communication steps. While BL-EST always selects the next node based on the longest outgoing path, ETF selects the node with the earliest starting time.
- **HDagg**: a more recent and more advanced state-of-the-art scheduler is the HDagg algorithm of Zarebavani et. al. [46]. HDagg is presented and analyzed in [46] for the specific purpose of speeding up SpTRSV computations; however, it is in fact a scheduling algorithm that can be applied to any computational DAG. Moreover, HDagg considers a very similar scheduling model to ours, sorting the nodes of the DAG into so-called wavefronts (essentially equivalent to supersteps), and minimizing the amount of communication between these wavefronts. Our experiments also show that HDagg consistently outperforms both BL-EST and ETF. Due to this, HDagg is a very fitting baseline for our work from the academic side.

Besides list-based schedulers, clustering is also a prominent method of designing state-of-the-art heuristics; however, previous work has found that this approach is consistently outperformed by BL-EST and ETF in models with communication cost [27].

Note that Cilk, BL-EST and ETF return a “classical” schedule where nodes are assigned to concrete time steps; such a schedule can be naturally adapted to BSP and organized into supersteps, by adding a superstep barrier (closing the current computation phase) whenever communication is required. In contrast to this, schedules returned by HDagg are already in the appropriate format.

4.2 Initialization heuristics

In order to develop an initial BSP schedule, we use the following heuristic methods:

- **BSPg**: A BSP-tailored greedy algorithm that consecutively assigns nodes to processors when processors become idle. Generally, we only allow assigning a node v to processor p if this is possible without ending the computational phase of the current superstep s , i.e. if we can ensure that all of v ’s predecessors are already available on p by superstep

s (that is, they were either computed on processor p or in an earlier superstep). In case of multiple possible node assignments to a processor, tie-breaking is done with a heuristic that aims to minimize communication costs in the future. Once we cannot assign further nodes to at least half of the processors without a communication requirement, the computation phase of the current superstep is closed, and a next superstep is started.

- **Source:** A different greedy approach that in each step forms a new superstep from the next layer of source nodes in the DAG. As a preprocessing step in the beginning, the algorithm uses a simple rule to cluster the original source nodes of the DAG. Then in each superstep, it applies a round-robin-based approach to assign the current source nodes to processors, considering the nodes in a decreasing order according to $w(v)$ to ensure the load balancing of work costs between processors. Besides the current source nodes, the algorithm occasionally also adds some of their successors to the current superstep, if this requires no extra communication.
- **ILPinit:** We also apply an ILP-based initialization heuristic that divides the nodes into smaller batches according to a topological order, and consecutively finds a schedule for each batch separately (given the already-selected partial schedules on previous batches), using an ILP-formulation of the subproblem. This approach also provides good initial schedules, but it requires drastically more running time than the heuristics above.

Note that the above heuristics only assign the nodes to processors and supersteps, i.e. they only define π and τ . The required communication steps Γ are then derived from these separately afterwards, simply following a lazy communication schedule where every required value is sent in the last possible communication phase, immediately before it is needed.

4.3 Local search algorithm

Another ingredient of our framework is a hill climbing local search method (denoted HC) that begins from an initial solution (that is, an already found, valid BSP schedule), and attempts to make small improvements to this solution as long as this is possible, i.e. until either a local minimum is found where none of the potential modification steps result in an improvement, or until a predefined time limit is reached.

For each local improvement step, given a node v that is currently assigned to processor p and superstep s , we consider all the alternative schedules where v is assigned to any other processor $p' \neq p$ in superstep s , or v is assigned to any processor in supersteps $(s - 1)$ or $(s + 1)$, with the assignments of all the other nodes unchanged. We consider all such modification steps (for every node v), provided that they yield a valid BSP schedule with smaller total cost.

Note this algorithm needs to be aware of the cost of the modified solution after each potential improvement step. Recalculating this entirely for each option would be very time-consuming; as such, we apply a range of sophisticated data structures to store the current schedule, and use these to efficiently query (and update) the

cost change incurred by each potential improvement step, without having to consider nodes and supersteps that are unaffected by this modification.

Besides this algorithm, we also apply a separate, similar hill climbing method for the communication scheduling subproblem (denoted HCCs), which only tries to modify the communication schedule; that is, it checks whether any communication step $(v, p_1, p_2, s) \in \Gamma$ can be replaced by some other (v, p_1, p_2, s') such that the schedule is still valid and has lower cost.

4.4 ILP-based approach

As our most sophisticated approach, we represent the BSP scheduling task as an ILP problem, and apply a state-of-the-art open-source ILP solver (CBC, see [7]) to find a low-cost schedule. We use several techniques to express (parts of) the scheduling problem as an ILP:

- **ILPfull:** A naive representation of BSP scheduling as an ILP problem has already been described before in the work of [28], but only analyzed from a theoretical perspective. This formulation captures the entire problem as a single ILP, and hence it requires a very high number of variables. Due to this, even with sophisticated ILP solvers, this approach is only feasible for very small DAGs in practice.
- **ILPpart:** In order to handle larger DAGs, we develop a partial ILP formulation as a more advanced iterative improvement method. In particular, given a starting BSP schedule and two superstep indices $s_1 \leq s_2$, we define a partial ILP that only reorganizes the supersteps between s_1 and s_2 ; that is, we only consider nodes v that are currently assigned to one of the supersteps in the interval $[s_1, s_2]$, and try to reassign these differently to any processor and any superstep in $[s_1, s_2]$, with the rest of the schedule unchanged. This gives a significantly smaller ILP that essentially only scales with the number of supersteps between s_1 and s_2 and the number of nodes assigned to these supersteps. Given this partial ILP formulation, we can then divide the range of supersteps into disjoint intervals, and then repeatedly use this approach to further polish each part of our schedule.
- **ILPcs:** We also devise and implement a separate ILP representation for the communication scheduling subproblem, i.e. the scheduling of communication steps Γ when π and τ are already fixed. This problem has a significantly smaller degree of freedom, hence the ILP solver can often return reasonably good solutions for it on the entire DAG, even for DAGs of larger size. If we already have a BSP schedule with a specific π and τ , we can use this ILP formulation to find a more optimal scheduling of the communication steps, hence resulting in a lower total cost.

When using an ILP solver in practice, we can achieve much better results by starting the solver from a good initial solution; this is provided by the initialization and local search algorithms discussed before.

4.5 Multilevel approach

While our algorithms above perform well in general, we have found that they are often unable to find good solutions in problems dominated by communication costs, e.g. when the weights c_v or some of the parameters g , ℓ or λ_{p_1, p_2} are excessively large. Intuitively speaking, in this case, a reasonable solution always needs to assign a well-connected cluster of nodes to the same processor, in order to avoid too much communication. In contrast to this, both our initialization heuristics and our local search algorithms attempt to (re)schedule single nodes separately, so they do not perform well in this case.

In order to address this problem, we also present a *multilevel* scheduling algorithm, inspired by the multilevel approach that consistently provides state-of-the-art results for hypergraph partitioning and other problems [15, 31, 33, 37]. When adapted to our scheduling problem, the three main steps of the multilevel paradigm are as follows:

- (1) **Coarsening** the DAG iteratively into smaller and smaller DAGs, which (ideally) retain most of the structure of the original DAG. In our case, we do this by repeatedly contracting a selected edge (u, v) of the DAG into a single node. The contracted edges are chosen in each step such that (i) the graph still remains a DAG after the contraction, and (ii) we prefer edges (u, v) where the total work weight $w(u) + w(v)$ is small, but the communication weight $c(u)$ is large.
- (2) **Solving** the BSP scheduling problem in our coarsened DAG with the algorithms discussed before. The coarsened DAG is not only much smaller (and hence more viable for our algorithms), but also naturally ensures that larger clusters of nodes are assigned to the same processor and superstep.
- (3) **Uncoarsening and refining** this schedule iteratively. That is, in each step, we undo the next few contraction steps in a reverse order, thus obtaining a slightly larger DAG, and we extend our current assignment π and τ to the newly uncontracted nodes. We then execute a few iterative improvement steps (using our local search algorithm) to ensure that the schedule is further refined to fit the more delicate structure of the more uncoarsened DAG.

Note that this approach is somewhat similar to the idea of combining an acyclic DAG partitioner with list schedulers in the work of [27]. However, while most of the experiments in [27] focus on creating a few (at most $4 \cdot P$) larger partitions, our coarsening phase sorts the DAG into a larger number of smaller clusters. Even more importantly, instead of applying a partitioning directly, our algorithm coarsens the DAG in a step-by-step fashion, since the gradual refinement steps in the uncoarsening phase are the key element of the multilevel approach.

5 COMPUTATIONAL DAG DATABASE

Besides the algorithms, we also present a collection of DAGs that represent a diverse set of real-world computational tasks, and hence offer a large and realistic benchmark to evaluate our scheduling algorithms. Our computational DAG database is available at

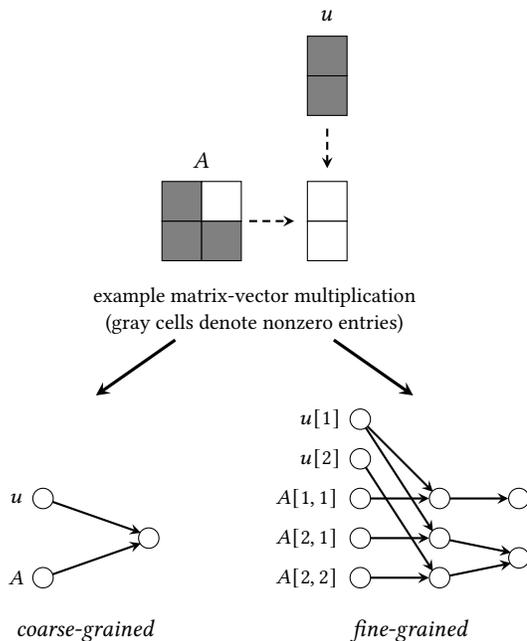


Figure 2: Coarse-grained and fine-grained DAG representation of a simple matrix-vector multiplication.

https://github.com/Algebraic-Programming/HyperDAG_DB, and its details are discussed in Appendix B.

Most of the DAGs in our database correspond to computations in some algebraic form: they are derived either directly from algebraic computations, or from computations in various other areas that can naturally be expressed in algebraic form. Since these kinds of computations typically work with matrices and vectors, there are two natural ways to represent them as computational DAGs. In the *coarse-grained* representation, each matrix or vector corresponds to (the output of) a single node in our DAG. In the *fine-grained* representation, we also attempt to capture the internal structure of matrices/vectors: in this case, each nonzero entry in a matrix/vector is represented by (the output of) a separate node in our DAG. An example for the two representations is shown in Figure 2.

To obtain coarse-grained DAGs, we extended the C++ GraphBLAS framework from [45] with a so-called *hyperDAG backend*, which automatically extracts the structure of a given computation while the computation is running. This allows us to conveniently obtain a DAG representation for the wide variety of algorithms implemented in GraphBLAS, such as the conjugate gradient and biconjugate gradient stabilized methods, the pagerank algorithm, label propagation, k -nearest neighbors, and many more. Since many of these algorithms are iterative methods, we can obtain different computational DAGs with this technique if we run the given algorithms for a predefined number of iterations, or until convergence.

To obtain fine-grained DAGs, the extension of GraphBLAS would be significantly more technical, so we follow a different approach: we produce a simple tool that synthetically generates the computational DAG corresponding to four concrete algorithms (conjugate gradient, k -nearest neighbors, sparse matrix-vector multiplication

and iterated matrix-vector multiplication) for a given pattern of nonzeros in the input matrices/vectors. While this only captures a few algorithms, it also has the advantage that we can conveniently generate a large number of computational DAGs (of any desired size) by running this generator for different matrices of a specific size and density of nonzeros.

As a technical detail, we note that in line with some recent works [29, 31], the DAGs in our database are represented in a hypergraph format (with hyperedges containing a node and all of its direct successors), but these are easy to convert back into a DAG format.

6 EXPERIMENTAL SETUP

For our experiments, we implemented the algorithms described above. Since the CBC ILP solver has a convenient Python interface, we implemented the ILP-based methods in Python, and the rest of the algorithms in C++. Our experiments were conducted on a workstation equipped with Intel Core i9-10900 CPU at 2.80GHz and 62.5 GiB of RAM.

The up-to-date version of our scheduling framework is available at <https://github.com/Algebraic-Programming/OneStopParallel>. We note that the current version of this scheduling tool has been significantly extended and upgraded since our experiments. For reproducibility, the version of the implementation used for the experiments, as well as the computational DAG database and the data from our experiments, are available in the repository at [1].

In order to tune our algorithms, we first ran some preliminary experiments on a small training set of 10 fine-grained computational DAGs, with their sizes ranging from $n = 15$ to $n = 2000$, and with BSP parameter combinations from $P \in \{4, 8, 16\}$ and $g \in \{1, 3, 5\}$, and a fixed $\ell = 5$. These initial experiments (discussed in Appendix C) showed that `ILPinit` is only competitive when we have very few processors, while it is rather time-consuming; as such, we chose to only apply `ILPinit` for $P = 4$.

We also observed that the CBC solver can usually return relatively good solutions whenever the number of variables in the ILP representation is below approximately 4 000; we have used this as our guiding principle in `ILPpart`, choosing to extend the superstep interval $[s_1, s_2]$ until the number of variables in the resulting ILP exceeds 4 000. Besides this, we also noted that the ILP solver rarely returns any reasonable solution above 20 000 variables: in this case, even the preprocessing heuristics of CBC can exceed a 1-hour time limit. Due to this, we only attempt the naive `ILPfull` approach from [28] in very small DAGs, where the estimated number of ILP variables is below 20 000. We point out that this limitation is partially due to the fact that our work uses an open-source ILP solver; with today’s more powerful commercial ILP solvers, the same approach could be applied to a significantly higher number of variables.

For our main experiments, we construct 4 different test sets of DAGs, labeled `tiny`, `small`, `medium` and `large`, with n in the ranges [40, 80], [250, 500], [1000, 2000] and [5000, 10000], respectively. We generated a set of fine-grained instances with different properties (varying matrix sizes and iterations to produce some “wider” and some “deeper” DAGs), covering these intervals. This resulted in 12 DAGs in the `tiny` dataset, and 21 in the remaining datasets (the `tiny` set is smaller since only a few parameter options can generate

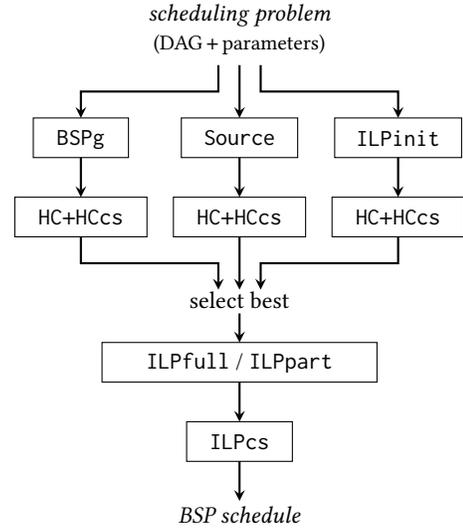


Figure 3: Summary of our scheduling framework.

so small DAGs). We further added to each of these datasets all the coarse-grained instances in the DAG database which has n in the same interval.

Besides the machine parameters in the training set, we experiment with different values of ℓ , and also with different NUMA parameters. Our NUMA settings each correspond to a binary tree hierarchy over the P leaf nodes, with the communication cost increasing by a specific factor Δ over each new level, with choices of $\Delta \in \{2, 3, 4\}$. For example, in case we have $P = 8$ and $\Delta = 3$, then the communication costs from the first processor are $\lambda_{1,2} = 1$, then $\lambda_{1,p} = 3$ for $p \in \{3, 4\}$, and $\lambda_{1,p} = 9$ for $p \in \{5, 6, 7, 8\}$.

Finally, we also create a smaller dataset with DAGs of size $n \in [50000, 100000]$, called the huge dataset, in order to observe how our simpler algorithms scale to even larger DAGs. We do not study the ILP-based methods here, since they would be far too time-consuming in this case.

For the experiments, we combine our algorithms in the pipeline shown in Figure 3. We begin by running the initialization heuristics to obtain different initial schedules. We then improve each of these separately with the local search methods (since running `HC+HCcs` is rather inexpensive), and then select the best schedule obtained this way. We then use the ILP-based methods: we begin with `ILPfull` in the few cases where the number of variables is below 20 000. If `ILPfull` was not applicable, or its solution was not shown to be optimal by the ILP solver, we apply the further ILP-based methods `ILPpart` and `ILPcs`.

When `ILPfull` is applicable, we assign a time limit of 1 hour to this. We allow 5 minutes for `HC+HCcs` and 5 minutes for `ILPcs` (although these algorithms rarely time out), and 3 minutes for each iteration of `ILPpart`.

When using the multilevel approach, the pipeline begins with a coarsening step (see Figure 4). We then apply the scheduling framework shown in Figure 3, and finally refine the DAG in the end. Since the coarsened versions of the DAGs only provide an imprecise estimation of the real amount of communication required in the

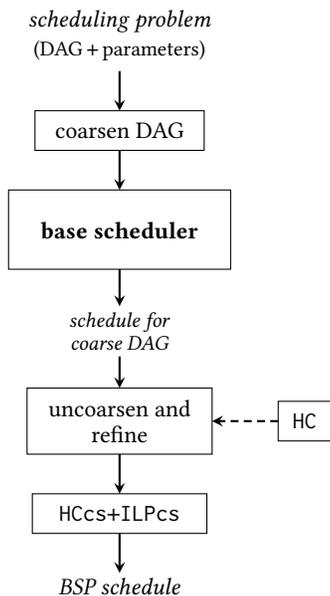


Figure 4: Summary of our multilevel framework. Base scheduler refers to the pipeline in Figure 3 (without ILPcs).

original DAG, we explicitly run the communication scheduling algorithms (HCcs and ILPcs) after the uncoarsening phase.

7 EMPIRICAL RESULTS

On any given problem instance, we evaluate the scheduling algorithms by the ratio between the cost returned by our algorithm and the baselines. In each dataset/experiment, we then aggregate these ratios through a geometric mean (this is more accurate for ratios than the average), to obtain a metric of the overall improvement achieved by our algorithms (with respect to the baselines) on the dataset. The results in our figures are normalized with respect to the cost of the Cilk baseline.

As for our remaining baselines, we found that ETF and BL-EST are consistently outperformed by HDagg, so out of these, we only include HDagg in our figures and tables. The schedules achieved by ETF and BL-EST are briefly discussed in Appendix C.

For simplicity, besides the baselines, our diagrams only show the best initialization method (labelled *Init*), the cost after applying HC+HCcs (labelled HCcs), and the final cost after all the ILP methods (labelled ILP). More details on the results (e.g. improvements by each separate ILP method) are also provided in Appendix C.

7.1 Without NUMA

We first run our schedulers on the tiny, small, medium and large datasets, for $P \in \{4, 8, 16\}$, $g \in \{1, 3, 5\}$ and $\ell = 5$, without NUMA. Over the combination of all parameters and datasets, the mean cost ratio between Cilk and our scheduler is 0.56, while the cost ratio between HDagg and our scheduler is 0.76. This means that our approach indeed returns schedules with significantly lower cost than the baselines: it achieves a 44% cost reduction compared to Cilk, and a 24% reduction compared to HDagg.

A detailed analysis also shows that this improvement factor depends on the choice of parameters and dataset. In particular, Table 1 shows the cost reduction with respect to the baselines, separated according to g and P (left), and g and the dataset (right). The table shows that the improvement ranges from 32% to 51% compared to Cilk, and from 13% to 40% compared to HDagg. We can observe that the difference between our scheduler and the baselines consistently grows larger with higher g ; this is because communication costs become more dominant, but these are not (accurately) considered by the baselines. The same holds for larger P values, since this often result in more data that needs to be communicated in a schedule. As for the size of the dataset, the tables suggest that the improvement compared to Cilk is mostly unaffected, while the improvement compared to HDagg becomes smaller for large datasets. This is likely because our schedulers come with a higher time complexity, and hence they do not scale as well as HDagg; improving this property is a strong candidate for our future work. However, we note that even in these cases, our schedules are still consistently better than those returned by HDagg.

In order to understand the role of each of our different algorithmic ingredients, we also show the cost ratios achieved by the algorithms (compared to Cilk), separated for different values of g , in Figure 5. The figure shows that while the initialization heuristics are already much more efficient than the baselines, the local search and ILP methods both achieve some further improvement. In Appendix C, we also discuss the role of the different algorithms in more detail; for instance, the ILP-based methods tend to result in a more significant improvement on the smaller datasets, and only a minor improvement for larger DAGs.

We also considered the effect of the latency parameter ℓ on our schedules separately (see Appendix C for details). Our experiments show that the difference between our scheduler and the baselines also grows for larger values of ℓ , although not as rapidly and as consistently as for the parameter g . This is in line with our previous observations, since the latency ℓ is essentially a different kind of communication cost, which is once again not considered by the baseline methods.

For our preliminary experiments on the huge dataset, we only applied the non-ILP methods from our algorithmic framework: BSPg and Source, followed by HC+HCcs with a larger time limit of 30 minutes. Even for this larger dataset, our scheduler achieves an improvement ranging from 15% to 41% compared to Cilk, and a more marginal improvement from 6% to 13% compared to HDagg (details again in Appendix C). This demonstrates that our general approach also scales reasonably well to computational DAGs with up to 100 000 nodes.

7.2 With NUMA effects

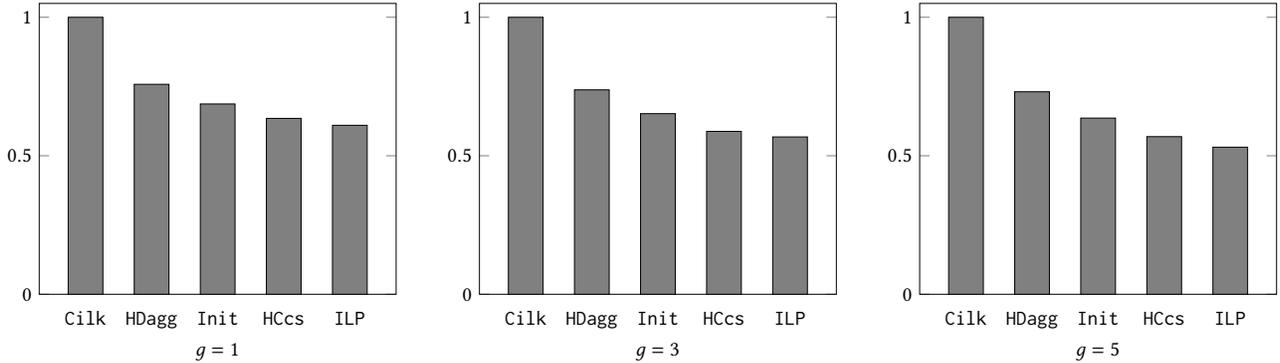
When we extend the BSP model with NUMA effects, the improvements achieved by our scheduler are even larger. On average over all the parameters $P \in \{8, 16\}$ and $\Delta \in \{2, 3, 4\}$, our method provides a 60% improvement compared to Cilk and a 43% improvement compared to HDagg.

The results in this setting show an even stronger dependency on P and the NUMA multiplier Δ : the improvement consistently grows in both of these parameters, as also illustrated in Figure 6

Table 1: Results achieved by our scheduler (in a NUMA-free setting), restricted to given values of g , P and given datasets. The two numbers in each cell show the reduction in cost compared to Cilk and HDagg, respectively.

| | $g = 1$ | $g = 3$ | $g = 5$ |
|----------|-----------|-----------|-----------|
| $P = 4$ | 32% / 20% | 40% / 24% | 44% / 26% |
| $P = 8$ | 40% / 21% | 45% / 25% | 45% / 25% |
| $P = 16$ | 43% / 18% | 49% / 26% | 51% / 30% |

| | $g = 1$ | $g = 3$ | $g = 5$ |
|--------|-----------|-----------|-----------|
| tiny | 32% / 26% | 40% / 35% | 43% / 40% |
| small | 38% / 22% | 44% / 30% | 46% / 32% |
| medium | 43% / 17% | 47% / 21% | 49% / 23% |
| large | 41% / 13% | 46% / 14% | 48% / 15% |

**Figure 5: Performance comparison of Cilk, HDagg and our scheduling algorithms without NUMA effects, for values $g \in \{1, 3, 5\}$.**

(let us ignore the ML column in the figure for now). In particular, for the case of $P = 8$ and $\Delta = 2$, the improvement is only 48% compared to Cilk and only 27% compared to HDagg. On the other hand, when we have $P = 16$ and $\Delta = 4$, the schedule returned by our algorithm has a 71% lower cost than Cilk, and a 58% lower cost than HDagg. We also show the concrete improvements for each case in a numerical format in Table 2.

Note that the case of $P = 16$ and $\Delta = 4$ amounts to a very significant improvement in scheduling cost: more than a $3\times$ factor for Cilk, and almost a $2.5\times$ factor for HDagg. Moreover, since Δ is often indeed large in practice, one might argue that this is the most realistic among our parameters for capturing today’s computing architectures. Altogether, this suggests that our approach may indeed be able to provide significantly better schedules for many relevant applications.

We also point out that the local search and ILP methods also become much more important ingredients in this NUMA setting, achieving a notably larger further improvement than in the case without NUMA.

7.3 Multilevel scheduling

Finally, we analyze our multilevel scheduling method, which was designed particularly for the case when communication costs are very high. For instance, in our previous setting for the highest choice of $\Delta = 4$, we have seen that our scheduler is notably better than the baselines. However, this is only their relative performance; in fact, both our scheduler and the baselines perform rather poorly in this setting, and even the solutions returned by our scheduler

Table 2: Cost reduction achieved by our base scheduler in a setting with NUMA effects, for different values of P and Δ , compared to Cilk and HDagg, respectively.

| | $\Delta = 2$ | $\Delta = 3$ | $\Delta = 4$ |
|----------|--------------|--------------|--------------|
| $P = 8$ | 48% / 27% | 55% / 35% | 61% / 42% |
| $P = 16$ | 57% / 36% | 67% / 51% | 71% / 58% |

are often barely better (or in several cases, in fact, even worse) than the trivial solution of assigning all nodes to the same processor and superstep. Our multilevel algorithm, on the other hand, is able to find good solutions even in this rather challenging setting, consistently beating this trivial baseline.

The cost ratio of the multilevel method to the other schedulers is also shown in Figure 6 for different values of P and Δ . For completeness, we also show the improvement values in numerical format in Table 3. The figure shows that when communication costs are high (e.g. if we have $\Delta = 4$, or if we only have $\Delta = 3$ but $P = 16$, and hence there are NUMA coefficients as high as $\lambda_{1,16} = \Delta^{\log_2 P - 1} = 27$ in our binary hierarchy), then the multilevel method is able to considerably outperform our base scheduler. For the highest choice of P and Δ , the multilevel algorithm finds solutions that on average provide a further factor $2\times$ improvement to our base scheduler. Altogether, this amounts to a very significant, almost a factor $5\times$ cost reduction, even compared to HDagg.

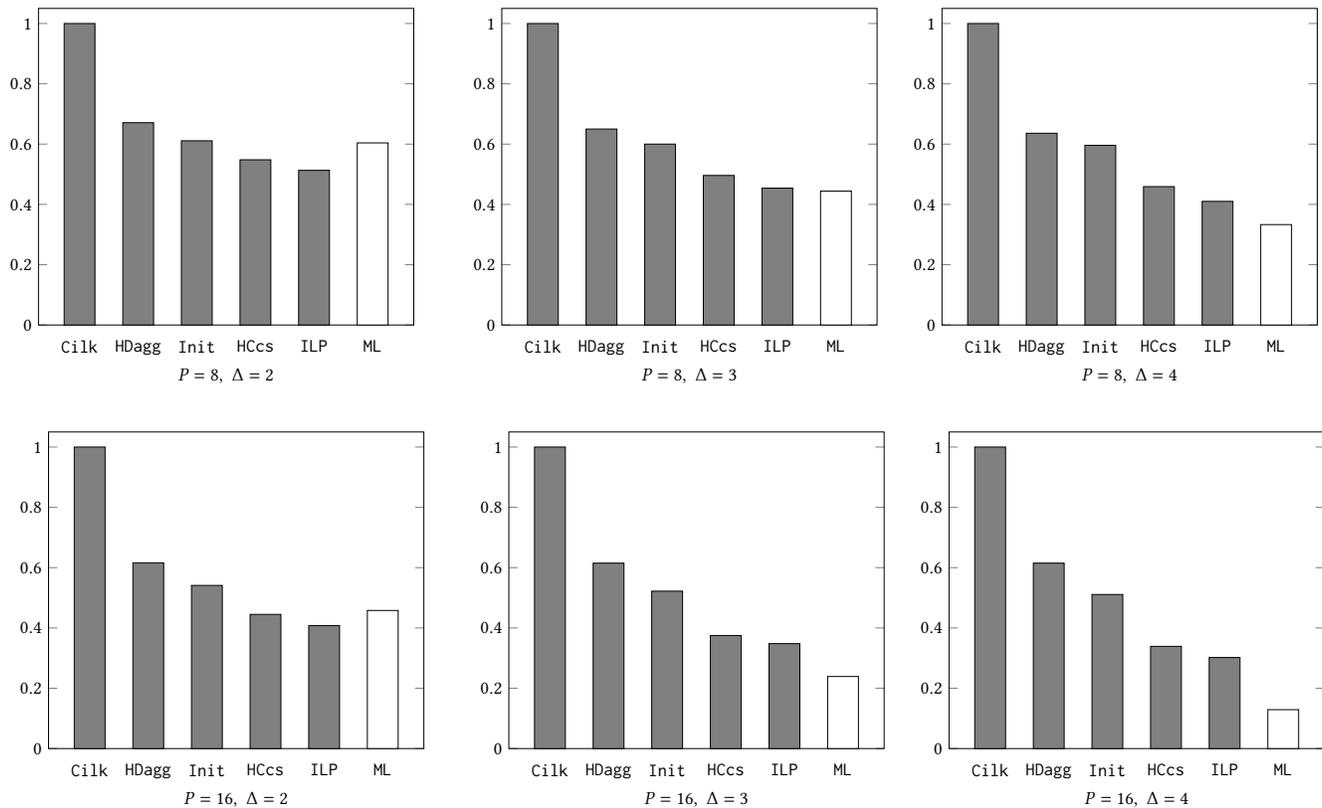


Figure 6: Performance comparison of Cilk, HDagg and our scheduling algorithms with NUMA effects, for different $P \in \{8, 16\}$ and different NUMA increase factors $\Delta \in \{2, 3, 4\}$. The multilevel approach (ML) is shown separately at the end. This specific figure only covers the small, medium and large datasets, since tiny is too small to coarsify with ML; however, in general, our improvement factors in Section 7.2 and Appendix C also include tiny.

Table 3: Cost reduction achieved by the multilevel algorithm with NUMA effects, for different P and Δ , compared to Cilk and HDagg.

| | $\Delta = 2$ | $\Delta = 3$ | $\Delta = 4$ |
|----------|--------------|--------------|--------------|
| $P = 8$ | 40% / 10% | 56% / 32% | 67% / 48% |
| $P = 16$ | 54% / 26% | 76% / 61% | 87% / 79% |

However, while the multilevel algorithm achieves a large further improvement for high communication costs (large Δ and/or P), on the other hand, for the case of $\Delta = 2$, the figure shows that ML is clearly inferior to our base scheduler. Similarly, if we revisit the setting of Section 7.1 without NUMA effects, the multilevel approach provides notably weaker solutions in general than our base scheduler (see Appendix C for details). This suggests that our multilevel method is indeed a specialized tool, which is useful mostly when our problem is dominated by high communication costs. However, we note that our algorithm is a relatively simple implementation of this multilevel scheduling idea; we believe that with some further

work, the same approach could become an efficient tool for any choice of parameters.

8 CONCLUSION

Our results show that if we consider a more detailed and more realistic model of the scheduling problem, then applying a range of advanced algorithms can significantly outperform classical scheduling heuristics like Cilk or HDagg.

Unsurprisingly, the main drawback of these more advanced algorithms is that they come with a notably higher running time. More specifically, BSPg and Source take a similarly small amount of time as the baselines, but the remaining algorithms are more time-consuming: HC+HCcs and Multi typically take between 1-2 seconds to 1-2 minutes on our DAGs, and the ILP-based methods take even longer, hence each individual ILP is capped at a few minutes. While the running times of these initial implementations can still be reduced significantly, in general, it is inevitable that this more complex direct optimization approach requires much more time than lightweight heuristics.

However, while this running time is certainly a limitation of our work, the benefits from the higher-quality schedules can still make our approach useful in various applications, e.g.:

- when the same computation needs to be executed many times, possibly with different inputs,
- when the computation can be captured in a coarse-grained way by relatively small DAGs, but possibly still with large node weights,
- in areas such as machine learning, where this scheduling time may still be negligible compared to the execution of the computation itself (e.g. the training of a neural network).

Furthermore, even in areas where our approach is not directly applicable in practice, our algorithms still provide crucial insight on the performance of the baseline heuristics: our solutions with much smaller cost prove that the schedules returned by these baselines are often very far from the optimum. Besides this, our results also highlight the fact that many real-world aspect, such as communication volume and NUMA costs, are critical to include in our model, otherwise the algorithms may return solutions that are significantly suboptimal.

There are many natural directions for future work, i.e. towards finding schedules of even lower cost in BSP, or in even more detailed models. In particular, we note that most of the ingredients in our algorithmic framework are relatively simple applications of a specific technique, used as prototypes to demonstrate the feasibility of our entire approach. As such, each of these algorithmic building blocks can be further improved into, or replaced by more sophisticated methods: we could apply e.g. more complex local search techniques that also attempt to escape local minima, or more efficient ILP formulations for our problems, or more advanced methods for the coarsening and refining phase of the multilevel method. We also point out that our experimental results were achieved with an open-source ILP solver; simply employing one of today's more powerful commercial ILP solvers could already significantly improve these results without any changes to the framework itself.

ACKNOWLEDGMENTS

We would like to thank Benjamin Lozes for his valuable help with applying the HDagg algorithm as a baseline in our work.

REFERENCES

- [1] 2024. Supplementary material. The scheduling algorithm implementations used in our experiments, our computational DAG database, and the data from our experiments are available at: https://github.com/Algebraic-Programming/Artifacts/tree/master/SPAA_2024_Efficient_Multi-Processor_Scheduling.
- [2] Thomas L Adam, K. Mani Chandu, and JR Dickson. 1974. A comparison of list schedules for parallel processing systems. *Commun. ACM* 17, 12 (1974), 685–690.
- [3] Ishfaq Ahmad and Yu-Kwong Kwok. 1998. On exploiting task duplication in parallel program scheduling. *IEEE Transactions on parallel and distributed systems* 9, 9 (1998), 872–892.
- [4] Rob H Bisseling. 2020. *Parallel Scientific Computation: A Structured Approach Using BSP*. Oxford University Press, USA.
- [5] Robert D Blumofe and Charles E Leiserson. 1999. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)* 46, 5 (1999), 720–748.
- [6] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten Von Eicken. 1993. LogP: Towards a realistic model of parallel computation. In *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, 1–12.
- [7] John Forrest and Robin Lougee-Heimer. 2005. CBC user guide. In *Emerging theory, methods, and applications*. INFORMS, 257–277.
- [8] Alfredo Goldman, Gregory Mounié, and Denis Trystram. 1998. Near optimal algorithms for scheduling independent chains in BSP. In *Proceedings. Fifth International Conference on High Performance Computing*. IEEE, 310–317.
- [9] Mohammadtaghi Hajiaghayi, Theodore Johnson, Mohammad Reza Khani, and Barna Saha. 2014. Hierarchical graph partitioning. In *Proceedings of the 26th ACM symposium on Parallelism in algorithms and architectures (SPAA)*, 51–60.
- [10] Julien Herrmann, Jonathan Kho, Bora Uçar, Kamer Kaya, and Ümit V Çatalyürek. 2017. Acyclic partitioning of large directed acyclic graphs. In *2017 17th IEEE/ACM international symposium on cluster, cloud and grid computing (CCGRID)*. IEEE, 371–380.
- [11] Jonathan MD Hill, Bill McColl, Dan C Stefanescu, Mark W Goudreau, Kevin Lang, Satish B Rao, Torsten Suel, Thanasis Tsantilas, and Rob H Bisseling. 1998. BSPlib: The BSP programming library. *Parallel Comput.* 24, 14 (1998), 1947–1980.
- [12] JA Hoogeveen, Jan Karel Lenstra, and Bart Veltman. 1994. Three, four, five, six, or the complexity of scheduling with communication delays. *Operations Research Letters* 16, 3 (1994), 129–137.
- [13] Jing-Jang Hwang, Yuan-Chieh Chow, Frank D Anger, and Chung-Yee Lee. 1989. Scheduling precedence graphs in systems with interprocessor communication times. *siam journal on computing* 18, 2 (1989), 244–257.
- [14] Hidehiro Kanemitsu, Masaki Hanada, and Hidenori Nakazato. 2016. Clustering-based task scheduling in a large number of heterogeneous processors. *IEEE Transactions on Parallel and Distributed Systems* 27, 11 (2016), 3144–3157.
- [15] George Karypis, Rajat Aggarwal, Vipin Kumar, and Shashi Shekhar. 1997. Multi-level hypergraph partitioning: Application in VLSI domain. In *Proceedings of the 34th annual Design Automation Conference*. 526–529.
- [16] Janardhan Kulkarni, Shi Li, Jakub Tarnawski, and Minwei Ye. 2020. Hierarchy-based algorithms for minimizing makespan under precedence and communication constraints. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM, 2770–2789.
- [17] Yu-Kwong Kwok and Ishfaq Ahmad. 1996. Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors. *IEEE transactions on parallel and distributed systems* 7, 5 (1996), 506–521.
- [18] Jan Karel Lenstra and AHG Rinnooy Kan. 1978. Complexity of scheduling under precedence constraints. *Operations Research* 26, 1 (1978), 22–35.
- [19] Joseph Y-T Leung and Gilbert H Young. 1990. Minimizing total tardiness on a single machine with precedence constraints. *ORSA Journal on Computing* 2, 4 (1990), 346–352.
- [20] Elaine Levey and Thomas Rothvoss. 2016. A (1+ epsilon)-approximation for makespan scheduling with precedence constraints using LP hierarchies. In *Proceedings of the forty-eighth annual ACM symposium on Theory of Computing*. 168–177.
- [21] Shi Li. 2021. Towards PTAS for precedence constrained scheduling via combinatorial algorithms. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*. SIAM, 2991–3010.
- [22] Quanquan C Liu, Manish Purohit, Zoya Svitkina, Erik Vee, and Joshua R Wang. 2022. Scheduling with Communication Delay in Near-Linear Time. In *39th International Symposium on Theoretical Aspects of Computer Science (STACS)*.
- [23] Bill McColl. 2021. Mathematics, Models and Architectures. *Mathematics for Future Computing and Communications* (2021), 6.
- [24] William F McColl. 1995. Scalable computing. *Computer Science Today* (1995), 46–61.
- [25] William F McColl and Alexandre Tiskin. 1999. Memory-efficient matrix multiplication in the BSP model. *Algorithmica* 24 (1999), 287–297.
- [26] Shang Mingsheng, Sun Shixin, and Wang Qingxian. 2003. An efficient parallel scheduling algorithm of dependent task graphs. In *4th International Conference on Parallel and Distributed Computing, Applications and Technologies*. IEEE, 595–598.
- [27] M Yusuf Özkaya, Anne Benoit, Bora Uçar, Julien Herrmann, and Ümit V Çatalyürek. 2019. A scalable clustering-based task scheduler for homogeneous processors using DAG partitioning. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 155–165.
- [28] Pál András Papp, Georg Anegg, and AN Yzerman. 2023. DAG Scheduling in the BSP Model. *arXiv preprint arXiv:2303.05989* (2023).
- [29] Pál András Papp, Georg Anegg, and Albert-Jan N Yzerman. 2023. Partitioning hypergraphs is hard: Models, inapproximability, and applications. In *35th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. 415–425.
- [30] Christophe Picouleau. 1995. New complexity results on scheduling with small communication delays. *Discrete Applied Mathematics* 60, 1–3 (1995), 331–342.
- [31] Merten Popp, Sebastian Schlag, Christian Schulz, and Daniel Seemaier. 2021. Multilevel Acyclic Hypergraph Partitioning. In *2021 Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 1–15.
- [32] Andrei Radulescu and Arjan JC Van Gemund. 2002. Low-cost task scheduling for distributed-memory machines. *IEEE transactions on parallel and distributed systems* 13, 6 (2002), 648–658.
- [33] Sebastian Schlag, Vitali Henne, Tobias Heuer, Henning Meyerhenke, Peter Sanders, and Christian Schulz. 2016. K-way hypergraph partitioning via n-level recursive bisection. In *2016 Proceedings of the Eighteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 53–67.
- [34] David B Skillicorn, Jonathan Hill, and William F McColl. 1997. Questions and answers about BSP. *Scientific Programming* 6, 3 (1997), 249–274.
- [35] Ola Svensson. 2010. Conditional hardness of precedence constrained scheduling on identical machines. In *Proceedings of the forty-second ACM symposium on Theory of computing (STOC)*. 745–754.

- [36] Haluk Topcuoglu, Salim Hariri, and Min-You Wu. 2002. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE transactions on parallel and distributed systems* 13, 3 (2002), 260–274.
- [37] Aleksandar Trifunović and William J Knottenbelt. 2008. Parallel multilevel algorithms for hypergraph partitioning. *J. Parallel and Distrib. Comput.* 68, 5 (2008), 563–581.
- [38] Jeffrey D. Ullman. 1975. NP-complete scheduling problems. *Journal of Computer and System sciences* 10, 3 (1975), 384–393.
- [39] Leslie G Valiant. 1990. A bridging model for parallel computation. *Commun. ACM* 33, 8 (1990), 103–111.
- [40] Leslie G Valiant. 2011. A bridging model for multi-core computing. *J. Comput. System Sci.* 77, 1 (2011), 154–166.
- [41] Huijun Wang and Oliver Sinnen. 2018. List-scheduling versus cluster-scheduling. *IEEE Transactions on Parallel and Distributed Systems* 29, 8 (2018), 1736–1749.
- [42] Tao Yang and Apostolos Gerasoulis. 1994. DSC: Scheduling parallel tasks on an unbounded number of processors. *IEEE Transactions on parallel and distributed systems* 5, 9 (1994), 951–967.
- [43] AN Yzelman and Rob H Bisseling. 2012. An object-oriented bulk synchronous parallel library for multicore programming. *Concurrency and Computation: Practice and Experience* 24, 5 (2012), 533–553.
- [44] AN Yzelman, Rob H Bisseling, Dirk Roose, and Karl Meerbergen. 2014. Multi-coreBSP for C: a high-performance library for shared-memory parallel programming. *International Journal of Parallel Programming* 42 (2014), 619–642.
- [45] AN Yzelman, D Di Nardo, JM Nash, and WJ Suijlen. 2020. A C++ GraphBLAS: specification, implementation, parallelisation, and evaluation. *arXiv preprint arXiv:1906.03196* (2020).
- [46] Behrooz Zarebavani, Kazem Cheshmi, Bangtian Liu, Michelle Mills Strout, and Maryam Mehri Dehnavi. 2022. HDagg: hybrid aggregation of loop-carried dependence iterations in sparse matrix computations. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 1217–1227.

A DETAILS ON OUR ALGORITHMS

This section provides a more detailed discussion of our different algorithms. Recall that the implementations of these algorithms that we used in the experiments (together with our database and the data from the experiments) are available at [1], while a more up-to-date version of our scheduling framework can be accessed at <https://github.com/Algebraic-Programming/OneStopParallel>.

Our algorithms can be run via a set of Python scripts, which allow us to configure the files or directories containing the input problems to solve, and several parameters of the solving process. Besides this user interface, the Python scripts also contain the implementations of our ILP-based methods, and are responsible for invoking the CBC ILP solver.

The remaining C++ code (the Cilk, ETF and BL-EST baseline algorithms and their conversion to BSP, the BSPg algorithm, the HC and HCCs local search methods, and the ingredients of the multilevel framework) are available in the file `simple_schedulers.cpp`. The only exception is the Source heuristic, which is implemented in a separate file named `second_heuristic_weights_cluster.cpp`. These two C++ files are automatically compiled and invoked by the Python scripts.

Finally, we provide the Python scripts required for running and evaluating HDagg separately, since these are independent from our own implementations, and they also require the HDagg library as an external dependency [46].

As a general remark on our algorithms, we also note that some of our simpler methods only return an assignment of nodes to processors and supersteps, but not a concrete communication schedule. These algorithms implicitly assume that the corresponding Γ is the *lazy communication schedule* defined by the assignment, where every value is sent directly and in the last possible superstep; that is, if a node v has a direct predecessor u , and $\pi(v) \neq \pi(u)$, then we have $(u, \pi(u), \pi(v), \tau(v) - 1) \in \Gamma$. In particular, the BSP-converted variants of the baseline algorithms, the BSPg and Source heuristics, and even HC assumes such a lazy schedule, whereas the ILP-based methods and HCCs return a specific, optimized communication schedule. We note that ILPinit also returns a specific Γ , but this is replaced by the corresponding lazy schedule in our pipeline, since the algorithm is followed by HC, which is only designed to work with lazy communication schedules.

A.1 Baseline methods

The main idea of the Cilk work-stealing scheduler was already outlined before: it is a greedy algorithm where every ready task (i.e. node with all of its predecessors already computed) is added to the “ready stack” of one of the processors. Whenever one of the processors become idle (it finishes computing a node), it takes the topmost ready task from its own stack, and begins the execution of this node. If the processor’s stack is empty, it selects another processor with a non-empty stack (uniformly at random among these processors), and starts executing the node at the bottom of this stack, also removing the node from the stack. This ensures that no processor is idle whenever there is a ready node.

We note that originally, Cilk was not defined on DAGs, but instead on processes that spawn further subtasks during execution. In this original setting, each spawned subtask is added to the top

of the stack of the processor executing the parent process. In contrast to this original setting, our DAGs do not have parent-child relationships between the given nodes. To adapt the algorithm to DAGs, we use a very similar rule: whenever the execution of the last direct predecessor of a node v is finished, if this happens on processor p , then v is added to the top of processor p ’s stack.

As for BL-EST and ETF, we refer the reader to the work of [27] for more details. We note that our baseline implementation for both of these algorithms corresponds to the versions described in [27], which is already extended with the concept of communication volume (but not with latency or NUMA effects, since these are new in our model). In case have NUMA effects, we compute the Earliest Start Time (EST) in these baseline algorithms by considering the average coefficient of the communication cost over all pairs of processors, and multiplying $c(v)$ with this number when considering the required communication steps. Note that an extension of the EST computation with NUMA factors would also be possible, to obtain a version of these baselines that is more specialized towards our more realistic model; however, we leave the investigation of this approach to future work.

We also note that Cilk, BL-EST and ETF assign nodes to concrete points in time. This can be naturally converted into a BSP schedule by sorting it into supersteps iteratively: if some of the nodes are already assigned to supersteps $1, \dots, (s-1)$, we can find the earliest point t in time when our classical schedule begins executing a node v such that (i) v is not yet assigned to a superstep, (ii) v has a direct predecessor v_0 also not yet assigned to a superstep, and (iii) our schedule has $\pi(v) \neq \pi(v_0)$. This implies that the next computation phase can last at most until time t ; hence we assign all nodes that are executed before step t to superstep s , and we continue with this procedure for superstep $(s+1)$.

Finally, HDagg is a more sophisticated algorithmic baseline, which directly develops a BSP-like schedule, sorting the nodes of the DAG into supersteps (called ‘wavefronts’), and then distributing nodes among processors in each wavefront. The algorithm strives to ensure both a balanced workload among processors in a superstep, and a reduced amount of inter-processor communication between the supersteps. For more details of the algorithm, we refer the reader to [46]. Note that this algorithm was originally developed (and analyzed) for the specific purpose of speeding up SpTRSV computations, i.e. solving sparse linear systems defined by a triangular matrix. However, the method is in fact a general DAG scheduling algorithm that can be applied to any computational DAG; it is even analyzed in [46] using DAG terminology. That is, after a topological ordering, the adjacency matrix of any DAG becomes lower triangular, and by artificially adding an edge from each node to itself (i.e. setting the main diagonal of the matrix to 1, thus ensuring it is non-singular), we obtain an SpTRSV problem where the dependency DAG derived from the matrix is identical to our original computational DAG. As such, the schedule returned by HDagg for this matrix directly corresponds to a schedule for our original DAG in the BSP model.

Since the original implementation of HDagg from [46] is openly available, we directly apply this in our experiments: we convert our computational DAG into the required triangular matrix format, we use the example script from the HDagg library to output the

Algorithm 1 Summary of the BSPg heuristic

```

superstep  $\leftarrow$  0
endStep  $\leftarrow$  False
finishTimes  $\leftarrow$  {0}
free[p]  $\leftarrow$  True for all p
while there are still unassigned nodes do
  if endStep = True and finishTimes =  $\emptyset$  then
    readyp  $\leftarrow$   $\emptyset$  for all p
    readyall  $\leftarrow$  ready
    superstep  $\leftarrow$  superstep + 1
    endStep  $\leftarrow$  False
    finishTimes  $\leftarrow$  {0}
  end if
  t  $\leftarrow$  earliest time from finishTimes
  for all nodes v that finish at t do
    free[ $\pi(v)$ ]  $\leftarrow$  True
    for all out-neighbors u of v do
      if u's in-neighbors are all finished then
        ready  $\leftarrow$  ready  $\cup$  {u}
        if  $\forall (u_0, u) \in E$  we have either  $\pi(u_0) = \pi(v)$  or  $\tau(u_0) <$ 
          superstep then
          ready $\pi(v)$   $\leftarrow$  ready $\pi(v)$   $\cup$  {u}
        end if
      end if
    end for
  end for
  if endStep = False then
    while  $\exists p$  with free[p] = True and readyp  $\neq$   $\emptyset$  or  $\exists p$ 
    with free[p] = True and readyall  $\neq$   $\emptyset$  do
      v  $\leftarrow$  ChooseNode(p)
      delete v from ready, readyp and/or readyall
       $\pi(v) \leftarrow p$ ,  $\tau(v) \leftarrow$  superstep
      finishTimes  $\leftarrow$  finishTimes  $\cup$  {t + w(v)}
      free[p]  $\leftarrow$  False
    end while
  end if
  if readyall =  $\emptyset$  and  $\exists \frac{P}{2}$  idle processors then
    endStep  $\leftarrow$  True
  end if
end while

```

returned schedule, and then we evaluate this according to the cost model of our paper.

A.2 Initialization heuristics

The BSPg heuristics is a greedy method that is designed specifically for developing BSP schedules. Even though the output assigns the nodes to supersteps, the algorithm still considers the concrete time step of starting and finishing each task, like in classical schedulers such as Cilk or ETF; this helps us develop a balanced work cost between the processors in each superstep. When a task is finished, we consider the new tasks $v \in V$ that become ready (i.e. have all of their direct predecessors already finished). If v already has a predecessor on multiple processors in the current superstep, then v is added to a general *ready*_{*all*} set of nodes that only become available for all processors in the next superstep. Otherwise, if v only

has predecessors on a single processor p in the current superstep, then v is added to a set *ready*_{*p*} of that processor, signaling that processor p is already allowed to execute v in the current superstep. Whenever a task is finished and some processor p becomes free, they are assigned a new node to execute (based on rules discussed below), if this is still possible without closing the current computation phase, i.e. there are still nodes that have all their direct predecessors on p or in earlier supersteps. When half of the processors become idle (they cannot be assigned new nodes), the computation phase is closed, and a following superstep is started, where all the nodes in *ready*_{*all*} are now available to each processor.

When selecting a new node to compute for a processor p , we employ the following method (named ChooseNode in our pseudocode). We first attempt to choose from the nodes in *ready*_{*p*} that are only available for p , and if this is empty, we choose from the set *ready*_{*all*} available to all processors. From both sets, we use the same tie-breaking rule that attempts to reduce communication costs, assigning a score metric for each node v in the set, and selecting the node with the highest score. For this score, we consider all direct predecessors u of v , and if either u or one of its direct successors is already assigned to p , then we increase the score of v by $\frac{c(u)}{\text{outdeg}(u)}$, where $\text{outdeg}(u)$ is the outdegree of u . Intuitively, this can be understood as an estimator for the importance and probability of the fact that we can assign u and all its direct successors to p , thus not having to communicate u at all. If $c(u)$ is high, then sending u results in a higher cost, and thus more important to avoid; if $\text{outdeg}(u)$ is small, then there is a higher chance that we can indeed achieve this, hence avoiding a communication. As such, this simple score metric for v aims to estimate our opportunities to save communication costs in the future if we assign v to p .

We provide a pseudocode for BSPg in Algorithm 1. Note that p always denotes a processor from $\{1, \dots, P\}$. We also note that the pseudocode uses in-neighbor and out-neighbor as the short form of direct predecessor and direct successor, respectively.

The Source heuristic is similar in the sense that it begins processing the DAG from the source nodes. In each iteration, it essentially assigns all the current source nodes of the DAG to processors in some way to form the next superstep. It then disregards (“removes”) these source nodes (and their outgoing edges) from the DAG to create the next set of source nodes for the next superstep. In each superstep, the nodes are assigned to processors following a round-robin approach. The first superstep is handled in a special way: the initial source nodes are first organized into clusters, joining pairs of nodes when they have a common direct successor, and then these clusters are assigned to processors in a round-robin way. In all other supersteps, no clustering is done; instead, the source nodes are sorted in decreasing order according to their work weight, and then assigned to processors in this order in a round-robin fashion, to avoid accumulating a too large work cost on any of the processors.

After the round-robin assignment in each superstep, the heuristic also considers the direct successors of all the source nodes, and if for any of them it holds that all its in-neighbors are already assigned to the same processor p , then it also assigns this node to processor p in the current superstep. This allows us to avoid creating further supersteps unnecessarily, by slightly relaxing the principle that only the source nodes are assigned in each superstep.

We show a pseudocode for Source in Algorithm 2.

Algorithm 2 Summary of the Source heuristic

```

superstep  $\leftarrow 0$ 
while there are still unassigned nodes do
  sources  $\leftarrow$  all unassigned  $v \in V$  with  $\text{indeg}(v) = 0$ 
   $p \leftarrow 1$ 
  if superstep = 1 then
    for  $v \in \text{sources}$  do
      if  $v$  shares an out-neighbor with another  $u \in \text{sources}$ 
      then
        if  $u$  is already in a cluster then
          add  $v$  to  $u$ 's cluster
        else
          combine  $v$  and  $u$  into a new cluster
        end if
      end if
    end for
    for all clusters  $c$  do
      for all nodes  $v \in c$  do
         $\pi(v) \leftarrow p$ ,  $\tau(v) \leftarrow \text{superstep}$ 
        delete  $v$  from  $G$ 
      end for
       $p \leftarrow (p + 1)$  modulo  $P$ 
    end for
  else
    sort sources in decreasing order by  $w(v)$ 
    for  $v \in \text{sources}$  in order do
       $\pi(v) \leftarrow p$ ,  $\tau(v) \leftarrow \text{superstep}$ 
      delete  $v$  from  $G$ 
       $p \leftarrow (p + 1)$  modulo  $P$ 
    end for
  end if
  for all edges  $(v, u) \in E$  with  $v \in \text{sources}$  do
    if all in-neighbors of  $u$  are already assigned to  $\pi(v)$  then
       $\pi(u) \leftarrow \pi(v)$ ,  $\tau(u) \leftarrow \text{superstep}$ 
    end if
  end for
  superstep  $\leftarrow \text{superstep} + 1$ 
end while

```

Note that both BSPg and Source only define the computation phases of our schedule by assigning nodes to processors and supersteps; the corresponding communication steps for each communication phase are derived in the end according to a lazy communication schedule.

The ILPinit initializer is discussed later, together with the remaining ILP-based methods.

A.3 Local search

Our hill climbing methods take an initial solution (BSP schedule) provided by one of the initialization methods, and execute small modifications to this schedule that decrease its total cost, until a local minimum is reached where none of the modifications lead to an improvement (or a predefined time limit is exceeded).

As mentioned before, from each current solution, we consider the modification steps which only change the assignment of single

node v . In particular, if we currently have $\pi(v) = p$ and $\tau(v) = s$, we consider all the solutions for where we have $\pi(v) \in \{1, \dots, P\}$ (i.e. v on any processor) and $\tau(v) \in \{(s - 1), s, (s + 1)\}$ (i.e. v in the previous, same, or next superstep), with the assignments for every other node unchanged.

Our HC method also employs several data structures to ensure that the cost of a potential modification can be computed efficiently, and the same data structures can also be updated efficiently after executing a modification. For instance, for each superstep s , we keep the work cost and the communication cost of each processor in a sorted set, and for each processor p we keep an external pointer to the entry describing this processor in the set; this allows for a lookup of the maximum in $O(1)$ time, and when updating the cost, a deletion in amortized $O(1)$ and an insertion in logarithmic time.

Furthermore, for each node v and processor p , we explicitly store the first superstep s when node v is required on processor p (due to one of v 's direct successors); due to the lazy communication schedule, this implies that v will be sent from $\pi(v)$ to p in superstep $(s - 1)$ (unless $p = \pi(v)$). This ensures that whenever we consider moving v to a different processor, we can immediately identify the communication steps that are affected by this change, and hence we can efficiently compute the resulting change in the total communication cost.

The list of valid moves in the search space (i.e. modification options that still provide a valid BSP schedule) is also computed in a preprocessing phase, and maintained throughout the algorithm; when a node v is moved to another processor and/or superstep, we only need to update the possible move options for v and its direct predecessors/successors.

Together, these data structures ensure that in each step of HC, we simply iterate through the list of valid move options, efficiently check if any of them leads to an improvement, and apply the selected move option, also updating our data structures efficiently.

The communication schedule hill climbing (HCCs) follows a very similar approach. Note that this settings already considers the assignments π and τ to be fixed. For each necessary communication step (when v needs to be sent from $\pi(v)$ to p), this assignment naturally defines an earliest and latest communication phase when this transfer can happen: the earliest is $\tau(v)$, and the latest is $(s_0 - 1)$ for the first superstep s_0 that assigns a direct successor of v to processor p . Our HCCs method considers alternative communication schedules with $(v, \pi(v), p, s) \in \Gamma$ for different possible $s \in [\tau(v), s_0 - 1]$. As before, in each step, we consider a modification of selecting a different such s for only one of the required communication steps, with the remaining communication steps in Γ unchanged. Note that for simplicity, this setting implicitly assumes that the value of v is directly sent to p from the processor $\pi(v)$ where it was computed. Similarly to HC, this hill climbing also uses sorted sets and external pointers to efficiently maintain the send cost and receive cost of each processor in every superstep.

For both hill climbing methods, there are two possible variants: we either (i) always greedily apply the first modification step that we find which decreases the cost, or (ii) we always consider all the possible modifications from the current solution, and select the one which decreases the cost by the largest amount. Our preliminary experiments have shown that neither of these two approaches is

clearly superior to the other in terms of the final schedule found; however, the second method is much more time-consuming, since there are usually numerous possible modification opportunities to analyze from a given solution. Due to this, we have applied the former, greedy variant of the approach in our experiments.

In our experiments, we allow the HC+HCcs methods to run for a time limit of 5 minutes on our main datasets, and for 30 minutes on the huge dataset. Given such a time limit, we always allocate 90% of the allowed time to HC, and the remaining 10% to HCcs.

A.4 ILP-based methods

Our most sophisticated approach is the ILP-based solving of the scheduling task or some of its subproblems.

The general approach to formulate the scheduling problem as an ILP has already been outlined in [28]: our model here corresponds to the submodel called FS by the authors of [28]. In this full ILP representation, there is a binary variable $\text{COMP}_{v,p,s}$ to indicate whether node v is computed on processor p in superstep s , and a value $\text{PRES}_{v,p,s}$ to indicate if the value of v is already available on p by the end of (the computational phase of) superstep s . Besides these, the communication steps are represented by binary variables $\text{COMM}_{v,p_1,p_2,s}$, indicating whether $(v, p_1, p_2, s) \in \Gamma$. These variables allow a natural way to express both the validity conditions of BSP model (precedence constraints and valid communication steps) and the elements of the cost functions with auxiliary variables; see [28] for details. For our ILPfull method, we implement the representation above, with only minimal changes: we aggregate some of the (originally separate) linear constraints into a single constraint, and we replace the role of the explicit variables $\text{PRES}_{v,p,s}$ by the unused variables $\text{COMM}_{v,p,p,s}$.

In our ILPcs method to optimize the communication schedule, the main idea is similar to that of HCcs: we consider π and τ already fixed from the initial solution, and only try to optimize the necessary communication steps in Γ . Once again, the assignment defines an earliest superstep $\tau(v)$ and a latest superstep $(s_0 - 1)$ when we can send a value v from $\pi(v)$ to p if this is required. Hence for each $s \in [\tau(v), s_0 - 1]$, we define a binary variable $\text{COMM}_{v,p,s}$ to indicate whether the value is communicated in superstep s . Note that similarly to HCcs, this implicitly assumes that the value of v is always sent from $\pi(v)$. Given these variables, the main techniques in the ILPcs formulation are identical to ILPfull: we can use the same approach to ensure with linear constraints that the communication costs are measured properly, and that each communications step is indeed scheduled to some superstep.

The ILPpart and ILPinit formulations are similar to each other in the sense that they both only consider a particular subproblem (a concrete subset of nodes V_0 and a concrete interval of supersteps S_0), and express this as an ILP. In ILPpart, we begin from a given subset of supersteps S_0 : we split the BSP supersteps into disjoint intervals (from back to front), and consider the nodes that are currently assigned to the supersteps in S_0 in order to obtain our V_0 . We form these intervals iteratively, always growing the current interval until $|V_0| \cdot |S_0| \cdot P^2$ exceeds 4000; this is because the number of variables in the ILP representation is in the magnitude of $|V_0| \cdot |S_0| \cdot P^2$, so we use this metric to estimate (and limit) the size of the ILP problem obtained. In contrast to this, in case of ILPinit, we begin with a topological ordering of the DAG, and always select the next

batch of nodes in this ordering as V_0 , and set S_0 to cover the next 3 supersteps; once again, the number of nodes in each batch is increased until $|V_0| \cdot 3 \cdot P^2$ exceeds a given threshold (2000 in this case).

Note that given V_0 and S_0 , the ILP formulation for ILPpart and ILPinit are similar, but differ in the fact that in ILPpart, all the nodes outside of V_0 are already assigned to a processor and superstep, whereas in ILPinit, the successors of the nodes in V_0 are not assigned yet (hence we disregard them for the optimization).

For the ILP formulation of ILPpart and ILPinit, it is not surprising that the variables $\text{COMP}_{v,p,s}$ and $\text{COMM}_{v,p_1,p_2,s}$ are restricted to $v \in V_0$ and $s \in S_0$. This already ensures that the ILP problem is drastically smaller than ILPfull, since the number of variables only scales with $|V_0|$ and $|S_0|$, instead of n and the total number of supersteps. However, in these ILPs, there are several further design decisions that allow us to reduce the number of required variables, at the cost of some simple restrictions to the solution space:

- Some of the nodes $v \in V_0$ might have direct successors u that are scheduled into supersteps after the interval S_0 . Let $\pi(u) = p_1$. Then e.g. if we originally had $\pi(v) = p_1$, but the ILP solver sets $\pi(v) = p_2$, then this results in a newly required communication step; on the other hand, if we originally had $\pi(v) = p_2$, but the ILP solver sets $\pi(v) = p_1$, then a previous communication step becomes unnecessary. Note that these communication steps might either be within the superstep interval S_0 , or after S_0 . In order to ensure that our superstep formulation only scales with $|S_0|$ (instead of also optimizing communications in supersteps after S_0), we choose to ignore the case when v was originally communicated after S_0 : in this case, reassigning v might make this communication step unnecessary, and hence reduce the cost of an h -relation after S_0 , but we ignore this potential further gain in the ILP objective. On the other hand, whenever v is sent within S_0 , this is easily captured by our formulation via the variables $\text{COMM}_{v,p,p',s}$. In particular, in the last communication phase of S_0 , we only include the variables $\text{COMM}_{v,p,p',s}$ for processors p' that require the value of v after S_0 , but for these processors, we also add the constraint that v indeed needs to be present on processor p' by the end of S_0 . This ensures that our ILP indeed captures the fact that the reassignment comes with newly required communication steps; however, it limits our flexibility by requiring that these new communication steps need to happen until the end of S_0 .
- Some of the nodes $v \in V_0$ might also have direct predecessors u which are scheduled into supersteps before S_0 . Similarly to the case of successors, if v is rescheduled to a different processor, then this might result in newly required communication steps or previous communication steps becoming unnecessary. As before, we ignore the potential gains from removable communication steps that were scheduled before S_0 , in order to ensure that our ILP size only scales with $|S_0|$. Within S_0 , these communication steps can once again be captured with variables $\text{COMM}_{u,p_1,p_2,s}$ as before; however, we also need to add such variables for

the communication phase of the last superstep before S_0 . Furthermore, note that the number of such predecessors u can potentially be even larger than $|V_0|$, hence if we add these variables without consideration, they could notably increase the size of the ILP. To avoid this problem in practice, we apply two simplifications. On the one hand, note that Γ might ensure that the value of predecessor u is already communicated to several processors p_0 before S_0 ; since u is always present on these processors throughout S_0 , we do not add the variables $\text{COMM}_{u,p_1,p_0,s}$ in this case, and we also disregard the precedence constraint (u, v) for $\text{COMP}_{v,p_0,s}$ (since they are satisfied anyway). On the other hand, as before, we assume that the value of u is directly sent from $\pi(u)$ (which is fixed, since u is computed before S_0) to any processor; this removes a factor P from the number of communication variables for u . Finally, note that there may also be predecessors u that have both an out-neighbor $v \in V_0$ and an out-neighbor u' scheduled after S_0 ; in this case, if u is originally sent to $\pi(u')$ during S_0 , then we also need to add the constraint that the value of u must still be present on $\pi(u')$ by the end of S_0 in our ILP solution.

- In our communication schedule Γ , we may also have communication steps (v, p_1, p_2, s) where v was computed before S_0 , it is only required on p_2 after S_0 , but our Γ still just happens to schedule it during the interval S_0 , even though v has no direct successor in V_0 . These communication steps have no direct connection to the assignment of the nodes V_0 , but they still contribute to the communication costs in one of the supersteps of S_0 . The number of these communication steps can theoretically be as high as $\Theta(n)$; hence if we include them as optimizable options in our ILP formulation, then we may lose the advantage of only scaling with $|V_0|$. As such, we consider these communication steps fixed: the communication costs they incur appear as a pre-computed constant in the send and receive costs of the appropriate processors in the given supersteps, unaffected by our choice of the assignment for V_0 .

As mentioned before, we use the CBC open-source solver [7] for solving the ILP problems described above. Since `ILPfull` tries to solve the entire problem, we allow a larger time limit of 1 hour when running this method. `ILPcs` also looks for a global solution, but in a much more restricted problem, so we set its time limit to 5 minutes, similarly to `HC+HCcs`. We select an even shorter time limit for `ILPpart` and `ILPinit`: 3 minutes for the former, 2 minutes for the latter (since this is supposed to be an even faster heuristic just for initialization). Even this way, `ILPpart` and `ILPinit` still dominate the running time of the algorithms in our experiments, since both of these are iterative methods that are repeatedly applied on many different parts of the DAG or the schedule.

A.5 Multilevel approach

Our most novel algorithmic contribution is perhaps the application of the multilevel coarsen-solve-refine technique, which is a state-of-the-art approach in hypergraph partitioning tools [15, 31, 33, 37], to the domain of scheduling problems. We develop and analyze a

simple variant of this multilevel scheduling approach for the case when a scheduling problem is dominated by very high communication costs, since our other methods often fail to find high-quality solutions in this case.

In the first phase of the multilevel algorithm, our goal is to gradually coarsen the DAG into a significantly smaller representation that captures most of the structure of the original DAG. For this, we repeatedly contract a directed edge of the DAG into a single node, combining all the incoming and outgoing edges of the two nodes. Each such operation reduces the number of nodes by exactly one, so a repeated execution of this procedure allows us to obtain a coarsified DAG representation of any desired size.

In order to have a valid schedule that can be interpreted for each intermediate step of the coarsening procedure, it is critical to ensure that our graph indeed remains a DAG after each contraction step. We ensure this by only selecting edges to contract in each step that satisfy this property. In particular, an edge $(u, v) \in E$ can be contracted into a single node (without creating a directed cycle) if and only if there is no other directed path in the DAG from u to v apart from the edge (u, v) . Note that there are always contractable edges in a DAG: e.g. each non-sink node u can be contracted with its out-neighbor v that appears earliest in a topological ordering, since there can be no other directed path from u to v .

In our algorithm, we evaluate the contractable edges (u, v) based on two properties: (i) the total work weight $w(u) + w(v)$ that we would obtain after contraction, which should be small to ensure that no large cluster of nodes is forced onto the same processor in the same superstep, and (ii) the communication weight $c(u)$ of the source node, which should preferably be large, since the contraction step implies that the output of u will not require a communication step to be sent to $\pi(v)$, at least not due to this edge. In our implementation, we consider the following simple technique: we sort the list of contractable edges in an increasing order according to $w(u) + w(v)$, and we always select from the first $\frac{1}{3}$ of this list to ensure that we do not merge nodes with large work weight. From this first part of the list, we select the edge with the largest $c(u)$ value.

Note that after the contraction step, we sum up both the work weights and the communication weights of u and v to obtain the weights for the contracted node. This is entirely appropriate for work weights, since these are summed up on the same processor and superstep anyway. For communication weights, this only gives us an estimate (upper bound) on the actual communication requirements: if there is an edge (u, v) in our coarsened DAG, then from (possibly many) original nodes contracted into u , maybe only a few had an actual edge to v . This means that when a scheduling of the coarse DAG sends the value of u to v , and computes the cost of this based on summed weights $c(u)$, then this is only an upper bound on the amount of data that actually has to be communicated.

In our implementation of this coarsening phase, we find the list of all contractable edges in the DAG in the beginning, and we update this list after each contraction step. Note that after contracting (u, v) , this does not only involve updating the edges incident to u and v : there might be edges arbitrarily far in the DAG that become uncontractable after this step. In particular, if there is another edge (u', v') such that there is a long directed path from u to v' , and

another one from u' to v , then both edges are contractable originally; however, after contracting (u, v) , there will be a directed path from u' to v' via the contracted node, so this distant edge (u', v') loses its contractability.

In general, we note that in our implementation, each of the contraction steps are rather time-consuming, due to e.g. the need to update the edges described above, and because we always iterate through the list contractable edges to select the one with highest $c(u)$; both of these operations can require $O(|E|)$ time for each contraction step. This is not a problem in our work, since our more complex scheduling methods require significantly more running time anyway. However, it is an interesting question for future work whether we can develop efficient coarsening techniques with significantly smaller time complexity.

Another crucial question regarding the coarsening phase is the optimal amount of coarsening that provides the best schedule in the end. A too coarse DAG might not capture the overall structure well enough, whereas an insufficient amount of coarsening might not provide the advantages of the multilevel approach; in particular, in our setting, it might not remedy the problem of excessively high communication costs (i.e. that it is only beneficial to reassign larger clusters of nodes simultaneously). This task of finding the most favorable coarsening ratio is a very challenging and complex problem on its own; investigating this in detail is far beyond the scope of our paper. For our preliminary experiments to validate the multilevel approach, we simply select two specific rates: we coarsen the DAG to 30% and 15% of its original size, run the multilevel approach for both of these cases, and out of the two schedules obtained this way, we select the one with lower cost as the final output of our multilevel scheduler.

We also note that our main question in this coarsening phase was to obtain a significantly coarsened version of the DAG that maintains the acyclic property. This general question has been studied by multiple works in recent years [10, 31], but was mostly evaluated as a partitioning problem in itself, rather than analyzed as a tool for scheduling. The work of [27], as mentioned before, is an exception to this, which has applied one of these acyclic partitioners to further improve state-of-the-art list schedulers. However, in contrast to our iterative coarsening procedure, this approach applies a final acyclic partitioning; as such, it offers no straightforward way to uncoarsen the DAG gradually, and hence there is no opportunity to execute further refinement steps in a multilevel fashion in this case.

The solving phase is the simplest part of the multilevel approach: here we apply the algorithmic pipeline of Figure 3 on the coarsened DAG. In particular, recall that `ILPfull` is only applicable on very small DAGs, and `ILPpart` is also significantly more useful on smaller DAGs; as such, the multilevel approach allows us to also apply these methods more successfully on DAGs that are originally larger.

In the uncoarsening and refinement phase, we gradually undo the contraction steps in a reverse order, thus moving closer and closer to our original DAG. Every time after we execute a given number of uncontraction steps, we *refine* the current schedule. That is, we first project our current schedule (obtained from the solving phase or the previous refinement step) into our current, slightly

more uncoarsened DAG: we assign each node to the same processor and same superstep as its contracted counterpart. Note that since the contracted graph was still a DAG, this still produces a valid BSP schedule. However, the newly executed uncontraction steps reveal slightly more of the structure of the original DAG, so we can now fine-tune our schedule towards this. For this, we execute several improvement steps with our local search method (HC) to obtain a slightly more refined variant of our current schedule on this slightly more coarsified version of the DAG.

In our experiments, we choose to refine the schedule after every 5 uncontraction steps, and we run HC for at most 100 steps (or until a local minimum is reached). This makes the number of refinement phases proportional to the number of contraction steps; alternatively, one could also opt to only have a fixed number of refinement phases altogether.

We also note that this uncoarsening phase (and hence the whole multilevel approach) can be adapted much more naturally to BSP than to classical models: if nodes are assigned to concrete starting times instead of supersteps, then it is not immediately clear how a schedule on a coarser DAG should be projected to a slightly uncoarsened variant of the same DAG.

Recall that during the refinement steps, we only apply HC, but not HCCs, since the (partially) coarsened DAG often overestimates the required amount of communications (as it merges the weights $c(v)$ in each cluster). Instead, HCCs and ILPCs are applied separately on the original DAG after the uncoarsening has been finished.

Regarding the entire multilevel approach, we note that our implementation is only a preliminary exploration of this idea, and each element of this approach can be further improved to obtain a more advanced version of this algorithm. In particular, analyzing more complex DAG contraction methods, or refinement with more advanced algorithms, or clever methods to estimate the optimal coarsification factor are all promising directions for further improvement. We leave it to future work to investigate these more sophisticated variants of the approach.

Finally, as a side note, we point at that the work of [27] also applies the concept of Communication-to-Computation Ratio (CCR) to capture the concept that a scheduling problem is dominated by communication costs, i.e. when our multilevel algorithm seems superior to other methods. In the work of [27], CCR was simply defined as the ratio of $\sum_{v \in V} c(v)$ and $\sum_{v \in V} w(v)$. This metric is not straightforward to generalize to our model with significantly more parameters: multiplying the numerator with g and also the average NUMA coefficient $\frac{\sum \lambda_{p_1, p_2}}{p^2}$ is a rather natural extension, but it is not trivial to include e.g. the effect of the parameter ℓ in this formula.

B DETAILS ON THE DAG DATABASE

This section discusses the details on our computational DAG database. We note that the database itself (coarse grained instances, fine-grained generator, some examples files and tools) are available at https://github.com/Algebraic-Programming/HyperDAG_DB. The test set of DAGs used in our experiments are available with the remaining supplementary material in [1].

As noted before, the DAGs in our database are stored in a hyperDAG format, to be in line with the recent works of [29, 31].

However, this difference is only relevant from a theoretical modelling perspective, to emphasize the fact that even if a node v has multiple out-neighbors on another processor p , its output only needs to be sent to p once; due to this, for partitioning problems, it is more adequate to represent the output data of v as a hyperedge, containing v and all its out-neighbors. For our work, this is simply an alternative representation of the precedence constraints in the DAG, and hence it has no effect. In fact, all of our algorithms begin with the simple step of transforming these hyperDAGs back into a regular DAG representation.

B.1 Coarse-grained DAGs

In order to obtain the coarse-grained representation of a variety of computations, we considered the GraphBLAS implementation of [45], and extended this with a so-called HyperDAG backend. The goal of this backend is to run simultaneously to a regular GraphBLAS algorithm and gather meta-data during this run, which is then used to generate the representation of the executed computation. For this, the backend considers the various operation primitives that are used as building blocks for the computations in GraphBLAS, and ensures that each of these primitives identifies the inputs and outputs of the given operation, allowing us to reconstruct the structure of the computational DAG.

This backend automatically allows us to extract the DAG representation of a wide variety of algebraic computations implemented in GraphBLAS. This involves common iterative methods for linear solvers (e.g. Conjugate Gradient for positive definite systems, or BiCGStab for general systems), graph algorithms that are naturally expressible in an algebraic form (such as k -hop reachability, connected components, or the PageRank algorithm), classical or more advanced methods from machine learning (such as k -means, label propagation or sparse neural network inference), and more. For more details on the concrete GraphBLAS computations, we refer the reader to [45]. We note that several of the algorithms above are iterative methods; for these, we extract the corresponding computational DAGs both for a predefined small number of iterations (we set this to 3), and for the case when the algorithm is running until the iterative method converges.

We note that while larger containers (matrices, vectors) are easy to track within a GraphBLAS algorithm, some simpler data structures, such as scalars, are not trivial to track without introducing extensive changes to the algorithm implementations. Due to this, our extraction from GraphBLAS sometimes provides an incomplete DAG representation, leading to some isolated nodes or smaller isolated components in the resulting DAG. To obtain the test DAGs for our experiments, we simply consider the largest connected component in each of the extracted DAGs; while this does not always cover the entire GraphBLAS algorithm in question, it still represents a subDAG that corresponds to a valid (sub)computation, and anyway captures most of the structure of the whole computation in the majority of cases.

Note that assigning work weights $w(v)$ and communication weights $c(v)$ to the nodes of the extracted DAG is a non-trivial task: while the DAG structure of the computation is fixed for an algorithm, the sizes of the matrices and vectors involved can be arbitrary, based on the inputs in the concrete run. As such, for simplicity, we assign $w(v) = \text{indeg}(v) - 1$ to each node v of the

DAG, where $\text{indeg}(v)$ is the indegree of v : since the operation v represents combining $\text{indeg}(v)$ distinct values, $\text{indeg}(v) - 1$ is a reasonable estimation for the workload this requires (consider e.g. a summation or a multiplication). The only exception to this is the source nodes of the DAG, where instead of setting $w(v) = 0$, we still assign a work cost of $w(v) = 1$: while they correspond to inputs of the computation, loading or initializing these values might still require some computational resources. As for communication weights, we uniformly assign a weight of $c(v) = 1$ to all nodes in the coarse-grained DAG. We leave it to potential future work to develop a way to assign more accurate weights to the nodes in these extracted DAGs.

B.2 Fine-grained DAGs

In order to obtain fine-grained DAG representation of algebraic computations, we have implemented a simple generator tool that synthetically creates and outputs the computational DAG corresponding to a few specific algebraic computations. Each of the four computations depend on a square matrix A . In our experiments, A is always defined by a size N (number of rows/columns), and a probability parameter q , such that each entry in the matrix is nonzero independently with probability q . To construct computations from real-world matrices, the generator also has the option to load input matrices (i.e. nonzero patterns) from a file, but this is not used in our experiments.

Given this input matrix, our tool artificially creates the fine-grained computational DAG corresponding to this matrix, with each node describing a simple operation with a few nonzero values (e.g. addition or multiplication of scalars). In particular, the generator outputs DAGs corresponding to the following algorithms:

- **spmv**: multiplication of a sparse matrix A with a dense vector u . An example for this operation is also shown in Figure 2.
- **exp**: an iterative version of **spmv**, i.e. given a sparse matrix A with a dense vector u , the naive computation of the vector $A^k \cdot u$, by executing k distinct **spmv** operations. This computation is an important building block of many applications.
- **CG**: the well-known conjugate gradient method for finding a numerical solution to a system of linear equations, executed for k iterations.
- **kNN**: in GraphBLAS, this method refers to finding the nodes in a graph that are at most k hops away from a specific node (in contrast to machine learning, where k -NN usually covers a different concept). In terms of algebraic computations, this can be represented as to the multiplication of sparse matrix A and a vector u with a single non-zero entry, for k iterations.

Besides N and q , another defining parameter for the last three algorithms is the number of iterations k .

As before, the work weight is set to 1 for source nodes, and to the indegree of the node minus 1 for all other nodes; this is indeed realistic for our fine-grained DAGs, since e.g. the addition of 4 scalars indeed requires 3 addition operations. The communication weights $c(v)$ are again set to 1 for all nodes.

Table 4: Number of times each initialization method is the best, on spmv computations in the training set, separated for different values of P .

| | $P = 4$ | $P = 8$ | $P = 16$ |
|------------|------------|------------|------------|
| Source: 5 | Source: 7 | Source: 7 | Source: 7 |
| BSPg: 3 | BSPg: 2 | BSPg: 2 | BSPg: 2 |
| ILPinit: 1 | ILPinit: 0 | ILPinit: 0 | ILPinit: 0 |

Table 5: Number of times each initialization method is the best, on exp, cg and kNN computations in the training set, separated for different values of P and different DAG sizes.

| | $P = 4$ | $P = 8$ | $P = 16$ |
|----------------------|------------------------------------|------------------------------------|------------------------------------|
| $n \in [15, 120]$ | ILPinit: 6 Source: 0 BSPg: 0 | ILPinit: 6 Source: 0 BSPg: 0 | BSPg: 4 ILPinit: 1 Source: 1 |
| $n \in [200, 350]$ | ILPinit: 6 Source: 0 BSPg: 0 | BSPg: 6 ILPinit: 0 Source: 0 | BSPg: 6 ILPinit: 0 Source: 0 |
| $n \in [1000, 2000]$ | ILPinit: 6 BSPg: 3 Source: 0 | BSPg: 9 ILPinit: 0 Source: 0 | BSPg: 9 ILPinit: 0 Source: 0 |

B.3 Datasets for our experiments

To form our datasets, we generate fine-grained DAGs for different n values, and we also strive to produce DAGs of different shape: e.g. a higher number of iterations k results in a deeper DAG (where the longest path is longer), in contrast to a smaller k , or in contrast to spmv DAGs where no iterations happen at all, and the longest path always has size only 3.

For the dataset used for the initial training, we create 10 fine-grained instances, with n ranging from 15 to 1950.

For the actual test datasets, we always specified an interval for n first ([40, 80] for tiny, [250, 500] for small, [1000, 2000] for medium and [5000, 10000] for large), and we specifically created a fine-grained DAG with each of the 4 methods in our generator that is approximately in the beginning, middle, and end of this interval. This results in 12 fine-grained DAGs for the tiny set. For the remaining sets, there is a higher number of possible parameter combinations to create DAGs in the interval; as such, with all the iterative methods that allow more freedom to influence the depth of the DAG (i.e. exp, cg and kNN), we create two different DAGs, a deeper and a wider one, for the beginning, middle, and end of each node interval. This results in 6 fine-grained DAGs with exp, cg and kNN, and 3 fine-grained DAGs with spmv. Hence altogether, we have 21 fine-grained DAGs in the small, medium and large datasets.

Besides this, we also add to each dataset the coarse-grained instances in our database where the number of nodes fits into the defined interval. This adds 4 coarse-grained DAGs to the tiny dataset, and 3 coarse-grained DAGs to the small dataset.

In order to create the huge dataset, we generate 7 fine-grained instances with $n \in [50000, 100000]$: one with spmv, and two with

each of the remaining algorithms. We then extend this with 3 coarse-grained instances where n is in a similar magnitude (one of the instances only has $n = 47023$, the other two are within the interval).

C DETAILS ON THE EXPERIMENTS

C.1 Comparison of initializers

We begin by discussing our preliminary runs on the training set to evaluate the performance of the different initialization methods. With the 10 training set DAGs and 9 parameter combinations from $P \in \{4, 8, 16\}$ and $g \in \{1, 3, 5\}$, this amounts to 90 runs altogether.

During these runs, we found that each of our initialization methods can outperform the others. In particular, the best schedule was returned by BSPg in 44 cases, by Source in 20 cases and by ILPinit in 26 cases. Furthermore, we can observe that the relative performance of the heuristics show a strong dependence on the properties of the scheduling problem itself; in particular, on the size of the DAG, the number of processors P , and besides these, also on the “shape” of the DAG: the shallow DAGs produced by spmv computations behave rather differently from the rest of the fine-grained DAGs.

In particular, Tables 4 and 5 show the number of times each heuristic turned out to be the most successful, Table 4 for spmv computations, and Table 5 for all other fine-grained instances combined. Table 4 is separated according to P , while Table 5 is separated according to P and n . There are several straightforward observations from these tables. On the one hand, it is clear that Source is rather effective for the shallow spmv DAGs, but not very useful otherwise. ILPinit performs well either for very small DAGs, or for very small P . Finally, BSPg consistently delivers good results for most of the parameter combinations.

Since ILPinit is a very time-consuming initialization methods, based on these observations, we decide to only run ILPinit for problems with $P = 4$ processors in the experiments; this significantly reduces the running time required for the experiments. Although ILPinit is also often superior when n is small, we do not apply it in this case, since ILPfull and ILPiter can essentially fulfill the same role for these problems. Since both BSPg and Source are very fast heuristics with negligible running time compared to the other elements in our framework, we apply both of them on every input problem, regardless of the parameters.

C.2 Experiments without NUMA

The results of our experiments without NUMA effects have already been outlined in Section 7. For completeness, here we provide a table with the respective improvements for each combination of P , g and dataset, shown in Table 6. Note that each number in the tiny, small, medium and large rows is still the average of 16, 24, 21 and 21 runs on different DAGs, respectively. This more detailed table reveals some further details compared to Table 1; for instance, while the cost reduction generally increases with larger P , this in fact only holds for the larger datasets, and the effect is in fact the opposite for the tiny DAGs. It also shows that for the most extreme subcase, our scheduler achieves almost a factor $2\times$ improvement with respect to HDagg, and well over a factor $2\times$ improvement with respect to Cilk, even in this non-NUMA setting.

Table 6: Improvement achieved by our scheduler (without NUMA) for each combination of g , P and dataset, with respect to Cilk (first number in cell) and HDagg (second number in cell).

| | $g = 1$ | | | $g = 3$ | | | $g = 5$ | | |
|--------|---------|---------|----------|---------|---------|----------|---------|---------|----------|
| | $P = 4$ | $P = 8$ | $P = 16$ | $P = 4$ | $P = 8$ | $P = 16$ | $P = 4$ | $P = 8$ | $P = 16$ |
| tiny | 41%/34% | 33%/28% | 20%/16% | 49%/43% | 40%/36% | 28%/26% | 54%/49% | 30%/36% | 33%/32% |
| small | 33%/23% | 41%/25% | 39%/20% | 40%/28% | 46%/31% | 46%/30% | 43%/30% | 46%/32% | 49%/35% |
| medium | 31%/14% | 43%/17% | 53%/20% | 38%/16% | 47%/20% | 56%/27% | 42%/18% | 47%/20% | 58%/31% |
| large | 27%/9% | 41%/13% | 53%/16% | 34%/8% | 46%/12% | 56%/21% | 38%/7% | 46%/12% | 58%/13% |

Table 7: Ratio of costs achieved by our algorithms (similarly to Figure 5), for $g = 5$, on the different datasets.

| | BL-EST | ETF | Cilk | HDagg | Init | HCcs | ILPpart | ILPcs |
|--------|--------|-------|------|-------|-------|-------|---------|-------|
| tiny | 1.126 | 0.883 | 1 | 0.943 | 0.728 | 0.619 | 0.57 | 0.569 |
| small | 1.54 | 1.073 | 1 | 0.791 | 0.66 | 0.579 | 0.556 | 0.539 |
| medium | 1.896 | 1.254 | 1 | 0.658 | 0.592 | 0.542 | 0.529 | 0.506 |
| large | 2.142 | 1.517 | 1 | 0.609 | 0.591 | 0.547 | 0.542 | 0.521 |

Table 8: Cost reduction achieved by our scheduler compared to ETF on the tiny dataset, for each combination of g and P , without NUMA.

| | $g = 1$ | $g = 3$ | $g = 5$ |
|----------|---------|---------|---------|
| $P = 4$ | 38% | 43% | 46% |
| $P = 8$ | 33% | 31% | 32% |
| $P = 16$ | 22% | 27% | 28% |

Table 9: Cost reduction achieved by our scheduler for different values of ℓ , on the medium dataset for $g = 1$ and $P = 8$ (compared to Cilk/HDagg).

| $\ell = 2$ | $\ell = 5$ | $\ell = 10$ | $\ell = 20$ |
|------------|------------|-------------|-------------|
| 38% / 16% | 43% / 17% | 50% / 19% | 58% / 21% |

In order to better understand the different ILP methods, we show the performance of each algorithm on the datasets for $g = 5$ (and all of $P \in \{4, 8, 16\}$) in Table 7, with the ratios normalized to Cilk as in our figures before. The ILPpart column shows the relative cost of the schedule after running ILPfull and ILPpart, whereas ILPcs shows the final schedule after also running ILPcs. The table shows that in the tiny dataset, ILPfull/ILPpart has a significant effect, decreasing the mean ratio from 0.619 to 0.569, which amounts to a 8% cost decrease from 0.619. As n grows larger, the improvement

achieved by these methods becomes less significant. In contrast to this, ILPcs only achieves a minimal improvement on the smaller dataset, but it is more impressive when n is larger: it decreases the ratio from 0.529 to 0.506 on medium and from 0.542 to 0.521 on large (4% improvement in both cases). This suggests that ILPpart and ILPcs excel in different situations, and hence they are both valuable ingredients in our scheduler.

The table also contains our other academic baselines, ETF and BL-EST. The data shows that as n grows, even Cilk becomes more superior to both ETF and BL-EST. Also, both ETF and BL-EST are significantly outperformed by HDagg, except for a single case: ETF performs better than HDagg on the tiny dataset. Since ETF is the strongest baseline altogether for this specific dataset, for completeness, we also show the cost improvement of our scheduler on the tiny dataset compared to ETF in Table 8. The table shows that our scheduler is also consistently superior to ETF by a significant margin.

C.3 The role of latency

So far, we always had a fixed choice of $\ell = 5$ in our scheduling problems. As such, we also run a small experiment to investigate the effect of the parameter ℓ on our schedules. For this, we consider the medium dataset, with a choice of $g = 1$ (to ensure that communication costs are dominated by ℓ) and $P = 8$. We investigate different values for the latency in this setting: $\ell \in \{2, 5, 10, 20\}$.

We show the improvement achieved by our scheduler for each of these cases in Table 9. The table shows that similarly to g , the improvement increases for larger values of ℓ . However, in contrast to g , the latency needs to be set to much larger numerical values in order for this tendency to become clearly noticeable.

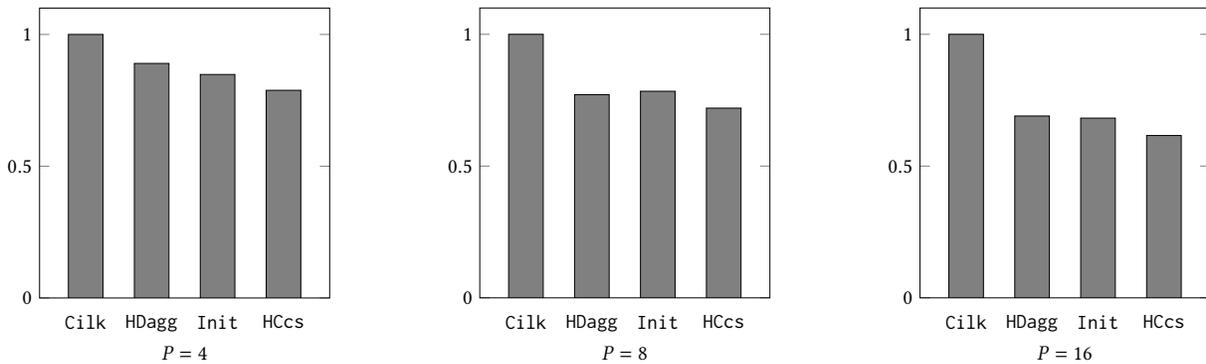


Figure 7: Improvement achieved by the initialization methods and the local search algorithm on the huge dataset, separated according to the value of P .

C.4 Experiments with NUMA

In our experiments with NUMA effects, we only considered $P = 8$ and $P = 16$ for the number of processors, since even as a binary tree, $P = 4$ gives a very shallow hierarchy with only two levels. Similarly, we only considered $g = 1$, since with $P = 16$ and $\Delta = 4$, this already gives a coefficient as high as $\Delta^3 = 64$ between pairs of processors that are only connected over the highest level of the hierarchy. As before, we set $\ell = 5$. For the NUMA multiplier, we consider the values $\Delta \in \{2, 3, 4\}$. Generally, as n grows larger, the improvement with respect to HDagg typically get smaller, whereas the improvement with respect to Cilk does not show such a clear pattern in terms of n .

The improvements achieved by our scheduler, separated according to P , Δ and the dataset, are shown in Table 10. The table confirms that the improvement achieved by our scheduler grows with a larger P and Δ , and this general rule holds consistently for all cases in all of the datasets, i.e. regardless of n . Compared to Cilk, the difference between the parameter combinations is most significant in the tiny dataset: the smallest and largest improvement is both achieved in this dataset.

Recall that the same results are aggregated over the different dataset (for each P and Δ) in Table 2. Finally, combined over all datasets and all values of P and Δ , the mean improvement is 60% and 43% compared to Cilk and HDagg, respectively. We note that ETF and BL-EST are also clearly inferior in this setting with NUMA effects: on all datasets combined, they returns a schedule that is on average 4% and 41% worse, respectively, than even Cilk.

C.5 The huge dataset

Our results on the huge dataset without NUMA effects are shown in Table 11, for separate values of P and g . As mentioned before in Section 7, the improvement compared to Cilk ranges from 15% to 41%, and the improvement to HDagg ranges from 6% to 13%.

To also see the amount of improvement we can attribute to the initializers and to HC+HCcs, we illustrate the improvement in Figure 7. This shows that the initializers achieve a 15%, 22% and 32% cost reduction compared to Cilk (for $P = 4, 8, 16$, respectively). The local search methods then further improve this by 7%, 8% and 10% (for $P = 4, 8, 16$, respectively) compared to Init. Hence even for these much larger DAGs, without applying our ILP-based methods,

our schedulers achieve a rather significant improvement to Cilk, especially for higher P values. When compared to HDagg, we see that Init is approximately on par with this baseline, and altogether, Init+HC+HCcs provides an improvement of 11%, 7% and 11% (for $P = 4, 8, 16$, respectively). We leave it to future work to improve the scaling of our algorithms, to ensure that they are also superior to HDagg by a more significant margin for DAGs of this size.

The results on the huge dataset with NUMA effects are mostly similar; these are shown in Table 12, separated according to P and Δ . The improvement here ranges from 30% to 48% compared to Cilk, and 7% to 21% compared to HDagg. As such, similarly to the smaller dataset, the improvements become larger when we have NUMA effects; however, the difference between our scheduler and HDagg still remains relatively small in several of the cases.

C.6 Multilevel scheduling

We primarily experiment with our multilevel scheduler in the settings where the communication costs are high; in our case, this mostly means the setting with NUMA effects. In our experiments, we only consider the multilevel scheduler on the small, medium and large datasets, since coarsening the DAGs in tiny would lead to an absurdly small DAG representation with as few as 6 nodes in the most extreme case.

Note that if the communication costs are very high in general (such as with NUMA), then finding a good schedule becomes a very challenging task: intuitively, it is only beneficial to assign two parts of the DAG to separate processors if the relative size of these parts is much larger than the number of edges going between them, otherwise the corresponding communication steps result in more extra cost than what we gain from the parallel execution of the subtasks. In particular, when we have high NUMA costs in our model, it can happen in our experiments that the best solution found by both the baseline methods and our scheduler is in fact more costly than the trivial solution of assigning the entire DAG to a single processor and single superstep. This clearly shows that the schedulers (both the baselines and ours) fail to find any reasonable schedule in this case. In particular, with very high communication cost parameters, it is not even clear at first whether a solution of lower cost even exists (just not found by our schedulers), or

Table 10: Improvement achieved by our scheduler (with NUMA) for each combination of P , Δ and dataset, with respect to Cilk (first number) and HDagg (second number), for a fixed choice of $g = 1$ (and $\ell = 5$).

| | $P = 8$ | | | $P = 16$ | | |
|--------|--------------|--------------|--------------|--------------|--------------|--------------|
| | $\Delta = 2$ | $\Delta = 3$ | $\Delta = 4$ | $\Delta = 2$ | $\Delta = 3$ | $\Delta = 4$ |
| tiny | 43% / 39% | 57% / 54% | 66% / 64% | 45% / 45% | 68% / 68% | 77% / 78% |
| small | 48% / 31% | 55% / 40% | 60% / 47% | 55% / 38% | 66% / 52% | 71% / 59% |
| medium | 50% / 23% | 55% / 30% | 58% / 35% | 61% / 34% | 67% / 44% | 69% / 49% |
| large | 49% / 14% | 54% / 18% | 57% / 20% | 61% / 28% | 67% / 38% | 69% / 42% |

Table 11: Cost reduction achieved by Init+HC+HCcs on the huge dataset without NUMA, compared to Cilk/HDagg.

| | $g = 1$ | $g = 3$ | $g = 5$ |
|----------|----------|-----------|-----------|
| $P = 4$ | 15% / 9% | 22% / 12% | 26% / 13% |
| $P = 8$ | 24% / 7% | 30% / 6% | 30% / 7% |
| $P = 16$ | 35% / 9% | 39% / 11% | 41% / 13% |

Table 12: Cost reduction achieved by Init+HC+HCcs on the huge dataset with NUMA, compared to Cilk/HDagg.

| | $\Delta = 2$ | $\Delta = 3$ | $\Delta = 4$ |
|----------|--------------|--------------|--------------|
| $P = 8$ | 30% / 7% | 34% / 7% | 37% / 7% |
| $P = 16$ | 41% / 12% | 45% / 16% | 48% / 21% |

whether the trivial solution mentioned above is in fact the optimal schedule in these problems.

Our multilevel algorithm answers this question, demonstrating that an approach designed especially for this communication-heavy case can find a solution with lower than the trivial cost even in this very challenging setting. In particular, over all experiments with NUMA costs ($P \in \{8, 16\}$, $\Delta \in \{2, 3, 4\}$), there were as many as 114 out of 396 cases where our base scheduler (and the baselines) could not find a solution with lower than trivial cost; however, with the application of the multilevel algorithm, the number of such cases was only 8 out of 396. These numbers are understood over all datasets except `tiny` (where multilevel scheduling was not applied, and hence it cannot be included in this comparison). This shows that even our relatively simple implementation of this multilevel idea can indeed very nicely fill the specialist role of scheduling these kind of problems.

The concrete improvement factors achieved by our multilevel algorithm (separately for $P \in \{8, 16\}$, $\Delta \in \{2, 3, 4\}$) are shown in Table 13 with respect to the baselines, and in Table 14 with respect to our (non-multilevel) scheduling framework. The tables show

that compared to the baselines, the multilevel approach can achieve a cost reduction of up to 87% and 79% in the most extreme case (with respect to Cilk/HDagg), which is more than a factor 7.7 \times improvement from Cilk and more than a factor 4.7 \times from HDagg. When compared to our base scheduling framework, the multilevel approach clearly remains inferior when $\Delta = 2$, it is approximately equally strong when $\Delta = 2$ and $P = 8$, and is clearly superior when we have $\Delta = 3$ and $P = 16$, or $\Delta = 4$. For the most extreme case of $\Delta = 4$ and $P = 16$, it provides another more than 2 \times improvement compared to the base scheduling framework.

Note that the tables are split into multiple rows based on whether we use the approach with a coarsification factor of 0.15 or 0.3, or whether we run both and select the better out of the two schedules. The table shows that applying and comparing both coarsifications indeed leads to some further improvement over the single-ratio approach. In case if we prefer running the multilevel approach with only a single ratio, 0.3 seems to be the clearly superior choice from the two.

In contrast to these cases, when we apply the multilevel scheduler to settings with lower communication costs, it typically provides weaker schedules than our base scheduling approach. This was already indicated before by the case of $\Delta = 2$ in Table 14. In our setting without NUMA, the difference is even larger: for instance, the mean ratio of the multilevel algorithm to our base scheduler is 1.246 for a choice of $P = 8$ and $g = 3$, and 1.515 for a choice of $P = 16$ and $g = 1$ (still understood over all datasets except `tiny`). While these ratios imply that the multilevel approach is still somewhat better than Cilk (and even marginally better than HDagg in the $P = 8$, $g = 3$ case), altogether, it is clearly inferior to our base scheduler. This suggests that our current implementation of the multilevel method can indeed be understood as a specialized tool for the case when communication costs are really dominant in the scheduling problem.

However, we believe that with further polishing, the multilevel approach can be improved to find a favorable coarsification ratio by itself (intuitively, to decide if coarsification is even necessary, or in a more advanced case, on which parts of the DAG it is necessary), and hence it can become a technique that provides high-quality schedules on all problems, regardless of input parameters. We believe that this is one of the most promising directions for future work in this topic.

Table 13: Cost reduction achieved by the multilevel scheduler in case of NUMA with respect to Cilk/HDagg, for each combination of P and Δ , on the combination of all datasets except tiny, for a fixed choice of $g = 1$ (and $\ell = 5$). The rows C_{15} and C_{30} show the result obtained when running the multilevel algorithm with a coarsification factor of 15% and 30%, respectively, while the C_{opt} row denotes the variant when we run both of these algorithms, and select the schedule of lower cost from the two outputs.

| | $P = 8$ | | | $P = 16$ | | |
|-----------|--------------|--------------|--------------|--------------|--------------|--------------|
| | $\Delta = 2$ | $\Delta = 3$ | $\Delta = 4$ | $\Delta = 2$ | $\Delta = 3$ | $\Delta = 4$ |
| C_{15} | 31% / -3% | 48% / 21% | 63% / 41% | 47% / 14% | 73% / 56% | 85% / 75% |
| C_{30} | 39% / 9% | 54% / 29% | 64% / 44% | 53% / 24% | 74% / 58% | 85% / 75% |
| C_{opt} | 40% / 10% | 56% / 32% | 67% / 48% | 54% / 26% | 76% / 61% | 87% / 79% |

Table 14: Improvement factor achieved by the multilevel scheduler with respect to our base scheduler (the framework of Figure 3), in the same setting as described in Table 13.

| | $P = 8$ | | | $P = 16$ | | |
|-----------|--------------|--------------|--------------|--------------|--------------|--------------|
| | $\Delta = 2$ | $\Delta = 3$ | $\Delta = 4$ | $\Delta = 2$ | $\Delta = 3$ | $\Delta = 4$ |
| C_{15} | 1.353 | 1.136 | 0.912 | 1.291 | 0.813 | 0.506 |
| C_{30} | 1.195 | 1.014 | 0.871 | 1.141 | 0.774 | 0.502 |
| C_{opt} | 1.179 | 0.979 | 0.812 | 1.122 | 0.711 | 0.429 |