

# Barrier-Aware Task Scheduling for Bulk-Synchronous Parallel Architectures

Tim Noack  
TU Darmstadt  
Darmstadt, Germany  
noack@esa.tu-darmstadt.de

Andreas Koch  
TU Darmstadt  
Darmstadt, Germany  
koch@esa.tu-darmstadt.de

## Abstract

Graphcore’s Intelligence Processing Unit (IPU) is a massively parallel processor based on the Bulk-Synchronous Parallel (BSP) model, which requires all processors to synchronize before any inter-processor communication can occur. While this constraint enables extraordinary hardware efficiency, it presents unique challenges for static task scheduling that have received little research attention.

We address this gap by proposing the Barrier-Aware List Scheduler (BALS), which brings list scheduling to the BSP model. BALS splits supersteps to create concurrency, steering placement decisions as synchronization-induced imbalances emerge. Unlike existing BSP schedulers—which each exhibit pathological behavior on certain DAG structures—BALS produces the *best* or *near-best* schedules across *all* graph topologies. For fine-grained workloads, we introduce Boundary Snapping to reduce superstep fragmentation, and Superstep Elimination, a post-processing pass that reduces synchronization overhead through task duplication and that can be applied to any BSP schedule.

Across 426 problems, the best existing BSP scheduler we tested produces schedules on average 269% longer than our non-BSP baseline scheduler. BALS closes this gap to just 11% overhead. These results demonstrate that BSP’s scheduling constraints *can* be effectively managed, making BSP processor designs like the IPU a promising approach for high-performance parallel architectures.

We open-source our BSP scheduling framework, which includes our proposed schedulers, implementations of all existing BSP schedulers evaluated in this work, and our benchmark suite [13].

## CCS Concepts

• Software and its engineering → Scheduling; • Hardware → Emerging architectures; • Theory of computation → Scheduling algorithms.

## Keywords

Task Scheduling, Bulk-Synchronous Parallel (BSP), List Scheduling, Parallel Computing, Heuristic Algorithms

## ACM Reference Format:

Tim Noack and Andreas Koch. 2026. Barrier-Aware Task Scheduling for Bulk-Synchronous Parallel Architectures. In *2026 International Conference on Supercomputing (ICS '26)*, July 06–09, 2026, Belfast, United Kingdom. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3797905.3807837>



This work is licensed under a Creative Commons Attribution 4.0 International License. *ICS '26, Belfast, United Kingdom*

© 2026 Copyright held by the owner/author(s).  
ACM ISBN 979-8-4007-2522-7/2026/07  
<https://doi.org/10.1145/3797905.3807837>

## 1 Introduction

Static task scheduling aims to map task dependency graphs to multiple processors while minimizing overall execution time. Most scheduling research assumes asynchronous execution models where data can be transferred at any time between processors [21].

The Bulk-Synchronous Parallel (BSP) model [24] offers an alternative execution paradigm. BSP structures computation into supersteps separated by global barriers, requiring all processors to synchronize before any inter-processor communication can take place.

Graphcore’s Intelligence Processing Unit (IPU) is the first modern commercial processor to enforce the BSP model in hardware. Rather than supporting asynchronous communication with dynamic routing and arbitration, the IPU requires compile-time scheduling of all data transfers relative to barrier synchronizations [8]. This design enables exceptional hardware efficiency: 1,472 cores per chip connected by a deterministic all-to-all fabric with 10 TB/s bandwidth [5]. However, this hardware-enforced synchronization constraint raises a critical question: **How much performance overhead does BSP impose compared to asynchronous execution, and can effective scheduling algorithms mitigate it?**

Research on static task scheduling for BSP is remarkably sparse [21]. Existing BSP schedulers include Integer Linear Programming formulations [15] that become intractable for realistic problem sizes, the Bulk Communication Scheduling Heuristic (BCSH) [6], HDagg [25], and the greedy heuristics of Papp et al. [14]. However, as we show in our evaluation, each of these schedulers exhibits pathological behavior on certain DAG structures.

This paper makes the following contributions:

- (1) **Translation Strategies.** We propose two approaches that convert existing delay-model schedules into valid BSP schedules.
- (2) **Barrier-Aware List Scheduler (BALS).** We present BALS, the first list-based scheduler that constructs schedules natively in the BSP model by splitting supersteps to create concurrency. For fine-grained workloads, BALS includes a *Boundary Snapping* mode to reduce superstep fragmentation.
- (3) **Superstep Elimination.** We propose a post-processing optimization that eliminates supersteps through task duplication, applicable to any BSP schedule.
- (4) **Comprehensive Evaluation.** We evaluate our schedulers against existing BSP schedulers on 426 problems spanning scientific workflows, fundamental parallel patterns, and fine-grained Sum-Product Networks.

- (5) **Open-Source Framework.** We release our BSP scheduling framework, including all proposed and existing schedulers as well as our benchmark suite [13].

## 2 Background

Static task scheduling involves mapping a task dependency graph onto multiple processors to minimize the overall completion time, known as the makespan. In this section, we formalize the problem, revisit the widely used delay model, introduce the general BSP model, and define the specific BSP execution model enforced by the IPU architecture.

### 2.1 Problem Formulation

We model an application as a Directed Acyclic Graph (DAG)

$$G = (V, E), \quad E \subseteq V \times V, \quad G \text{ acyclic.} \quad (1)$$

Each vertex  $i \in V$  is a task. Each edge  $(i, j) \in E$  denotes that  $j$  depends on the output of  $i$ . We denote the set of predecessors of  $j$  by  $\text{pred}(j)$  and the set of successors of  $i$  by  $\text{succ}(i)$ .

We consider a set of  $m$  processors  $P = \{1, \dots, m\}$ . The execution time of task  $i$  on processor  $p$  is  $c_i(p) > 0$ . For each edge  $(i, j) \in E$ , the time to transfer the data produced by  $i$  (placed on processor  $p$ ) to  $j$  (placed on processor  $q$ ) is  $d_{ij}(p, q) \geq 0$ ; by convention  $d_{ij}(p, p) = 0$ .

A schedule consists of (1.) a placement (mapping)  $\pi : V \rightarrow P$  assigning each task to a processor and (2.) a per-processor total order of the tasks assigned to it that respects all precedences. Let  $s_i$  and  $f_i$  denote the start and finish times of task  $i$  in the schedule. The makespan is

$$T = \max_{i \in V} f_i. \quad (2)$$

The scheduling problem is to choose  $\pi$  and the per-processor orders to minimize  $T$ .

Figure 1a visualizes the task graph of our running example, which we refer to throughout this paper.

### 2.2 Delay Model (Asynchronous Communication)

The vast majority of scheduling research targets asynchronous execution models, in which processors can communicate at any time. The delay model—the most widely used formalization—assumes that processors operate independently, computation overlaps with communication, and arbitrarily many transfers may be in flight simultaneously [21]. Under this model, task  $j$  becomes ready on candidate processor  $p$  at time

$$r_j(p) = \max_{i \in \text{pred}(j)} (f_i + d_{ij}(\pi(i), p)), \quad (3)$$

i.e., the maximum data-arrival time over all predecessors. If  $a_p$  denotes when processor  $p$  next becomes idle, then scheduling  $j$  on  $p$  yields

$$s_j = \max\{a_p, r_j(p)\}, \quad f_j = s_j + c_j(p), \quad a_p \leftarrow f_j. \quad (4)$$

A valid delay-model schedule for our running example (Figure 1a) is visualized in Figure 1b.

### 2.3 General BSP Model

The Bulk-Synchronous Parallel (BSP) model, introduced by Valiant in 1990 [24], structures parallel execution into a sequence of *supersteps* separated by global barrier synchronizations. In each superstep, processors perform local computation and communicate with each other; but all communication from one superstep becomes visible only after the next barrier. The model provides a bridge between hardware and algorithm design, making costs explicit and predictable.

The key constraint for scheduling in the BSP model is simple: **any inter-processor data dependency must cross at least one synchronization barrier.**

### 2.4 BSP Execution Model for the IPU

We now formalize the BSP execution model enforced by the Graphcore IPU architecture, which our schedulers target. This model captures the key constraint of BSP architectures: strict separation of communication and computation phases, with global barrier synchronization between supersteps. Our formalization defines how we compute the makespan for any given BSP schedule in our evaluation.

Throughout this paper, “BSP model” refers to this IPU-specific execution model that we detail in the following section, but we believe that the findings in this paper are generalizable to all BSP-like models.

A valid BSP-model schedule for our running example (Figure 1a) is visualized in Figure 1c.

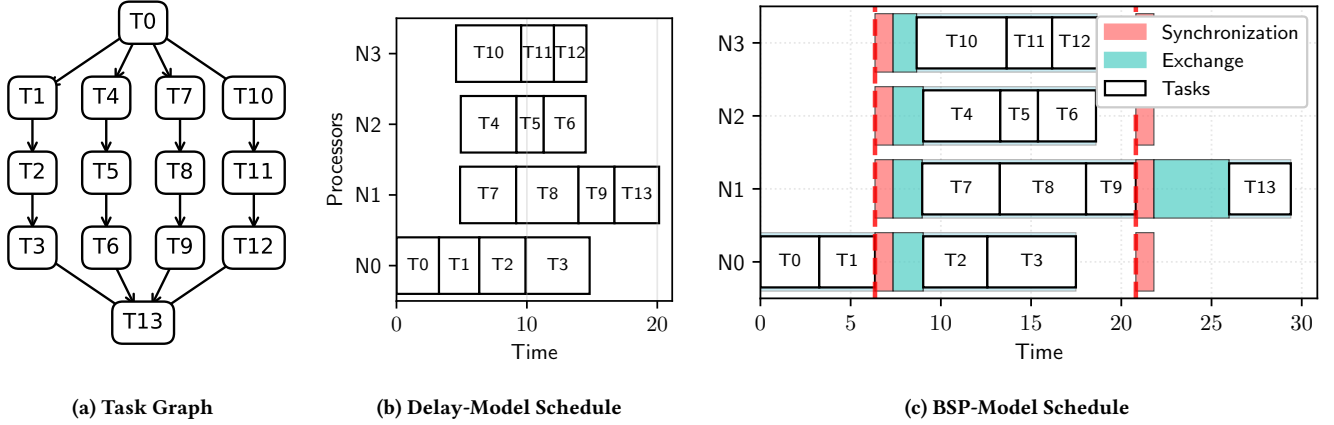
**Background Information for interested readers:** Graphcore IPUs organize computation into tiles (lightweight processor cores; 1,472 per chip), interconnected by an all-to-all fabric that exposes data lines rather than conventional packet-switched networks [5]. There are no on-chip routers with dynamic arbitration or queues. Instead, the compiler computes global, cycle-accurate schedules of all communications relative to synchronization barriers. For each tile, a compiler-generated deterministic communication program specifies exactly in which cycle (measured from the most recent barrier) each datum will be sent and when it arrives at another processor. Because the correctness of this plan depends on a shared time origin, all tiles must synchronize before the next wave of communications—precisely as in the BSP model.

*Formalization.* A BSP schedule for the IPU consists of a placement  $\pi : V \rightarrow P$  assigning tasks to processors and a task-to-superstep assignment  $\varphi : V \rightarrow \{1, 2, \dots, S\}$  determining the execution order. These must satisfy the precedence constraints, as inter-processor dependencies require data to cross at least one barrier:

$$\varphi(j) \geq \begin{cases} \varphi(i), & \text{if } \pi(i) = \pi(j), \\ \varphi(i) + 1, & \text{if } \pi(i) \neq \pi(j), \end{cases} \quad \forall (i, j) \in E. \quad (5)$$

Let  $V_s = \{i \in V \mid \varphi(i) = s\}$  denote tasks in superstep  $s$  and  $V_{s,p} = \{i \in V_s \mid \pi(i) = p\}$  those on processor  $p$ . Each superstep executes three sequential phases:

**1. Synchronization.** All processors wait at a barrier with fixed duration  $\sigma \geq 0$ .



**Figure 1: Task graph of our running example and valid schedules in the delay-model and the BSP-model. Note the synchronization barriers in the BSP schedule that force all processors to wait before communication phases.**

**2. Communication.** Processor  $p$  receives data from tasks in the previous superstep that were placed on other processors. The incoming edge set is

$$E_{s,p}^{\text{in}} = \{(i, j) \in E \mid \varphi(i) < s, \varphi(j) = s, \pi(j) = p, \pi(i) \neq p\}, \quad (6)$$

with communication time

$$G_s(p) = \sum_{(i,j) \in E_{s,p}^{\text{in}}} d_{ij}(\pi(i), p). \quad (7)$$

This formulation models each processor as *receiving* data sequentially, while being able to *send* the same datum to multiple processors in parallel (broadcast).

**3. Computation.** After receiving data, processor  $p$  executes its tasks with total time

$$C_s(p) = \sum_{i \in V_{s,p}} c_i(p). \quad (8)$$

The superstep duration is determined by the slowest processor:

$$T_s = \sigma + \max_{p \in P} (G_s(p) + C_s(p)). \quad (9)$$

The total makespan is

$$T_{\text{BSP}} = \sum_{s=1}^S T_s = \sum_{s=1}^S \left[ \sigma + \max_{p \in P} (G_s(p) + C_s(p)) \right]. \quad (10)$$

Without loss of generality, we assume  $\min(s_i) = 0$ , i.e. the schedule starts at time 0.

*Key Differences from the Delay Model.* This model differs from the delay model in three ways: (1.) communication and computation are strictly sequential—processors cannot compute while receiving data; (2.) inter-processor dependencies must cross at least one barrier; and (3.) superstep duration is governed by the slowest processor, resulting in less busy processors idling at barriers.

### 3 Related Work

Research on static task scheduling is vast under the delay model, with clustering and list-based heuristics widely used in both homogeneous and heterogeneous settings. However, only a few lines of prior work address task scheduling in the BSP model directly: (i) the Bulk Communication Scheduling Heuristic (BCSH) [6], (ii) the Hybrid DAG Aggregation (HDagg) algorithm [25], and (iii) the scheduling algorithms of Papp et al. [14, 15].

#### 3.1 The Bulk Communication Scheduling Heuristic (BCSH) Algorithm

Fujimoto et al. [6] observed a performance gap between delay-model schedules and executions on distributed-memory machines, especially for fine-grained DAGs where send/receive overhead dominates. In 1999, they proposed BCSH to reduce message overhead by packing communication at barriers—the first heuristic designed explicitly for BSP.

BCSH operates in two phases: (1) **Task graph grouping** iteratively pulls predecessors into the current layer starting from exit tasks, permitting task duplication to reduce cross-processor dependencies. Each layer maps to a superstep later. (2) **Group scheduling** places task groups onto processors using Graham’s List-Decreasing heuristic, ordering by weight and assigning greedily to the least-loaded processor.

The original BCSH targets homogeneous platforms. For heterogeneous architectures, we replace the earliest-start-time placement with an **earliest finishing time (EFT)** strategy that accounts for varying processor speeds and communication costs.

#### 3.2 HDagg: Hybrid DAG Aggregation

Zarebavani et al. [25] propose Hybrid DAG Aggregation (HDagg) for scheduling sparse matrix computations with loop-carried dependencies. While originally targeting SpTRSV, SpIC0, and SpILU0 kernels, the algorithm applies to general DAGs.

HDagg operates in two phases. First, it aggregates densely connected tasks: after removing transitive edges, it identifies subtrees

via a modified BFS traversal starting from sink tasks and moving toward predecessors, merging tasks within each subtree to improve locality. Second, it performs *load-balance preserving wavefront coarsening*: starting from the first wavefront (topological level), it greedily merges consecutive wavefronts until further merging would cause load imbalance. The *Potential Gain Proxy* (PGP) metric guides this decision, measuring the deviation from perfect load balance. A bin-packing strategy then assigns connected components within each coarsened wavefront to processors. In our BSP context, the resulting coarsened wavefronts correspond directly to supersteps.

### 3.3 Papp et al. BSP Scheduling Algorithms

Papp et al. [14, 15] present several algorithms for BSP DAG scheduling. Their Integer Linear Programming (ILP) formulation encodes task-to-processor and task-to-superstep assignments, intra-superstep ordering, and inter-superstep communication. While this can yield provably optimal schedules, the number of variables grows with  $(\text{supersteps}) \times (\text{processors}) \times (\text{tasks})$ , making ILPs intractable for the problem sizes we examine in this work.

To handle larger DAGs, they propose two greedy heuristics. **BSPg** assigns tasks to processors as they become idle, closing the current superstep when at least half of the processors cannot be assigned further tasks without requiring inter-processor communication. Tie-breaking favors tasks whose predecessors are already local, minimizing future communication. **Source** iteratively forms supersteps from current source tasks (those with all predecessors already scheduled), assigning them to processors in round-robin order sorted by decreasing work weight for load balance. After assignment, source tasks are removed and the process repeats.

Both heuristics can be improved by **hill-climbing local search**, which iteratively moves tasks between processors or adjacent supersteps whenever this reduces the schedule cost. Additionally, Papp et al. propose a **multilevel scheduler** for problems dominated by high communication costs. It first *coarsens* the DAG by repeatedly contracting edges (merging task pairs) while preserving acyclicity, then schedules the smaller coarsened DAG, and finally *uncoarsens* by iteratively undoing contractions while refining the schedule via hill-climbing at each level.

In our evaluation (Section 7.4), we benchmark against **BSPg**, **Source**, and the **multilevel scheduler** with hill-climbing refinement. We omit ILP-based refinement, as even the hill-climbing variant of the multilevel scheduler timed out on most of our problem instances.

## 4 Scheduling in the Delay-Model and Translating to the BSP-Model

Our first approach is to schedule in an asynchronous communication model using any off-the-shelf scheduler (we use HEFT [22]), and then *translate* that schedule into a BSP schedule by assigning each task to a superstep. Intuitively, the async scheduler already makes good placement and ordering decisions; our translation then inserts barriers and communication phases to enforce BSP semantics. The drawback is that idling caused by barrier synchronization only becomes visible *after* the async schedule has been fixed, so the scheduler cannot react by shifting work to otherwise idle processors.

We present two translation strategies: **Eager** and **Earliest Finishing Next**. Both strategies share a common definition: They consider a task to be *available* if it is ready (i.e., all predecessors are scheduled) and if it is the next unscheduled task in its processor's relative ordering. By only scheduling available tasks, the *per-processor* relative order from the delay-model schedule is conserved.

Figure 1 shows the running example: the async schedule was produced by HEFT; two translations are illustrated in Figure 2a and 2b.

### 4.1 Eager Translation Strategy

Our first strategy places *as many tasks in the current superstep as possible* and is shown in Algorithm 1.

---

#### Algorithm 1: Eager Translation Strategy

---

**Input:** Schedule in the delay-model

**Output:** Schedule in the BSP-model with the same relative task order and processor mapping.

**Strategy:** Start with the entry tasks and place *as many available tasks as legally possible* into the current superstep; then advance to the next superstep and repeat. “Legally possible” means that the tasks do not require inter-processor data produced in the *same* superstep.

---

Because the Eager strategy ignores timing information when forming the supersteps, it can create severe load imbalance. Figure 2a shows such a case: tasks T0–T3 were all mapped to processor N0 by HEFT and (legally) land in the same first superstep because their dependencies are intra-processor only. The result is a very long first compute phase with only processor 0 active, which is clearly suboptimal.

### 4.2 Earliest-Finishing-Next (EFN) Strategy

Our second strategy keeps track of the current (partially built) BSP schedule and uses it to compute, for each available task, how soon it could finish based on its data dependencies alone (Algorithm 2).

---

#### Algorithm 2: Earliest-Finishing-Next Translation Strategy

---

**Input:** Schedule in the delay-model

**Output:** Schedule in the BSP-model with the same relative task order and processor mapping.

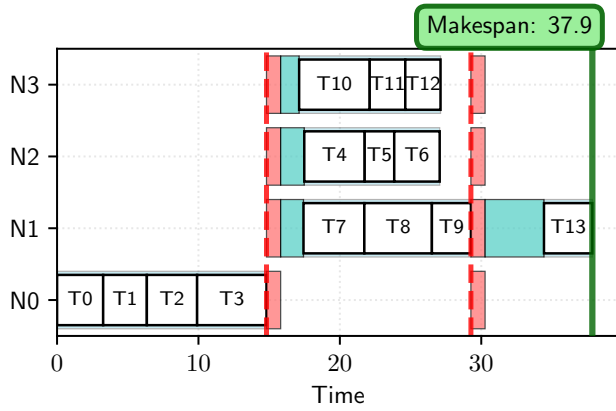
**Strategy:** For each available task  $i$ , compute the earliest time at which it *could finish if it were scheduled directly after all of its predecessors finished*:

$$t_i = \max_{j \in \text{pred}(i)} (f_j) + \sum_{\substack{j \in \text{pred}(i) \\ \pi(j) \neq \pi(i)}} d_{ji} + c_i(\pi(i)),$$

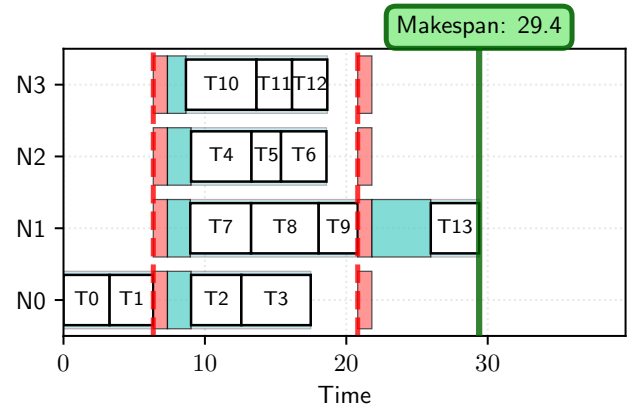
i.e. the latest predecessor finish time in the partially built BSP schedule, plus the time to receive inter-processor data, plus the task's own execution time.

Among the available tasks, place the one with the smallest  $t_i$  into the current superstep if legally possible; otherwise, start a new superstep and place it there.

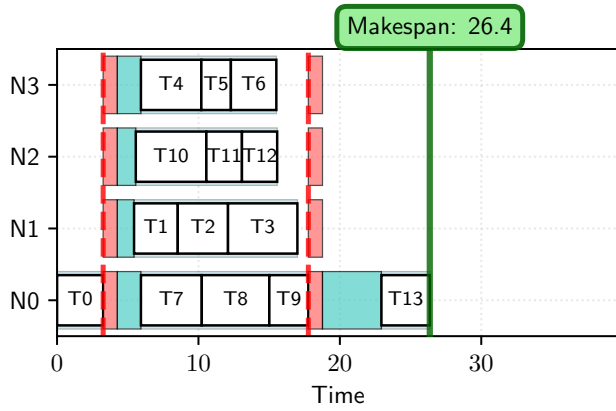
---



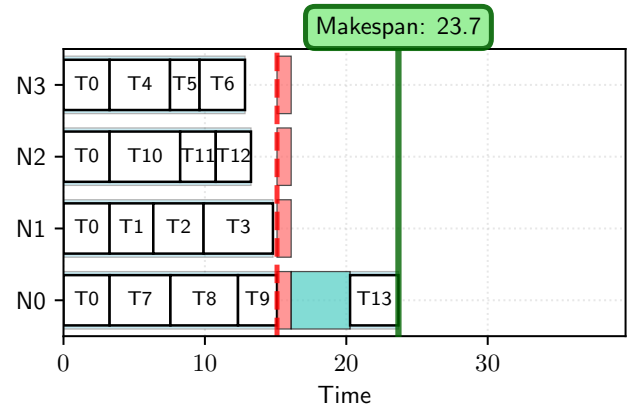
(a) Eager Translation Strategy of a HEFT generated schedule (Section 4.1)



(b) Earliest-Finishing-Next (EFN) Translation Strategy of a HEFT generated schedule (Section 4.2)



(c) Barrier-Aware List Scheduler (BALS) with upward rank for priority (Section 5)



(d) BALS + Superstep Elimination after beneficially eliminating the first superstep (Section 6)

Figure 2: BSP-model schedules of different schedulers proposed in this paper.

Figure 2b shows the result on our running example, for which EFN produces a near-optimal schedule. The only gap is that an optimal schedule would place task T1 into the second superstep. The translation strategy placed T1 in the first superstep because – after placing T0 – the available set becomes {T1, T4, T7, T10}. All of these tasks yield the same (earliest)  $t_i$  when placed on processor N0, since none of them requires inter-processor communication. Whichever is chosen is placed in the same superstep as T0, which later turns out to be suboptimal.

This example highlights a limitation of EFN: it must first *overpack* a superstep to justify introducing a new barrier.

## 5 Barrier-Aware List Scheduler (BALS)

The translation strategies presented in the previous section share a fundamental limitation: they operate on a finished asynchronous schedule, inserting barriers and communication phases *after* placement and ordering decisions have already been made. While they

can identify where synchronization will cause processors to idle, they cannot redistribute work in response.

To overcome this limitation, we propose the Barrier-Aware List Scheduler (BALS), which schedules directly in the BSP model from the start, steering placement decisions as imbalances emerge.

Most list schedulers for asynchronous systems follow a simple two-step pattern: (1.) give every task a priority, and (2.) place the highest-priority ready task onto the processor where it will achieve its earliest finishing time (EFT). This EFT logic involves a simple trade-off: a scheduler will keep placing tasks on the same processor until the benefit of using an idle processor outweighs the cost of communicating data to it.

### 5.1 List Scheduling in the BSP model

A naive attempt to adapt this logic to the BSP model would give the scheduler two choices for placing a task:

- (1) Add the task to an existing superstep on some processor.

- (2) Create a brand-new superstep at the end of the schedule for the task, which allows scheduling it on any processor.

As it turns out, this approach performs poorly because the scheduler will almost always choose the first option, piling tasks onto a single processor within a single superstep.

To understand why, consider the task graph from our running example (Figure 1a) and the BSP schedule from Figure 2a. Imagine the scheduler has just placed tasks T0, T1, T2, and T3 on processor N0. The next task to place is T4, which depends on T0. Placing T4 on any processor other than N0 would require the result of T0 to be communicated first. The scheduler's "thinking" process would be:

- Option 1 (Append): Place T4 on processor N0 right after T3 finishes. The task starts immediately.
- Option 2 (New Superstep): Create a new superstep for T4. However, a new superstep can only begin after the current one finishes and all processors synchronize. This means the new superstep for T4 will always start earliest at the finish time of T3, plus it incurs the cost of an extra barrier and communication phase.

Faced with these choices, the scheduler sees that appending is always locally optimal. The high upfront cost of a new superstep at the end of the schedule means the "pay to communicate" trade-off almost never makes sense.

## 5.2 Splitting Supersteps to create Concurrency

Our key insight was that the scheduler needs a third option: the ability to *split* an existing superstep. This allows the scheduler to create concurrency within the current schedule. When a task becomes ready, the scheduler finds the superstep where all its dependencies are met, splits it at that exact moment, and places the task in the newly created gap on the processor that offers the earliest finish time.

This leads to our Barrier-Aware List Scheduler (BALS), which combines these ideas into a two-tiered placement strategy for each ready task:

- (1) **Fill Idle Gaps First.** The scheduler first scans all existing supersteps for an idle "hole" where the task can run without increasing that superstep's total duration. This is the most efficient move, as it utilizes resources for free.
- (2) **If No Gaps, Make Space by Splitting.** Only if no such free slot is found, split a superstep to create new concurrent execution space.

## 5.3 Boundary Snapping

The splitting strategy described above creates a new superstep whenever a task's dependency-ready time falls inside an existing superstep. This can lead to excessive fragmentation on task graphs where individual task runtimes are comparable to the synchronization overhead  $\sigma$ .

Consider a predecessor task  $i$  with finish time  $f_i$  that completes shortly before its superstep ends. Without boundary snapping, BALS would split at  $f_i$ , creating a new superstep for the successor  $j$ . However, if the gap between  $f_i$  and the superstep boundary is small relative to  $\sigma$ , this split gains little parallelism while incurring a full synchronization penalty.

To address this, Algorithm 3 includes *boundary snapping* (step 2.ii): when a predecessor finishes within  $\kappa \cdot \sigma$  of its superstep boundary, we round the dependency-ready time up to that boundary instead of using the exact finish time. The snapping multiplier  $\kappa \geq 0$  controls how aggressively the scheduler avoids splitting supersteps.

When  $\kappa = 0$ , boundary snapping is disabled. For fine-grained task graphs where  $\sigma$  is on a similar time scale as task runtimes, we find  $\kappa = 10$  to be effective. On such workloads, boundary snapping can dramatically reduce the number of supersteps (e.g., from over 100 to around 10 on certain SPN benchmarks), yielding substantial makespan improvements. On coarse-grained graphs where task runtimes dominate  $\sigma$ , the feature has little to no effect, as tasks rarely finish that close to superstep boundaries.

The complete logic for our BALS scheduler is detailed in Algorithm 3. The schedule generated by BALS for our running example is visualized in Figure 2c.

## 5.4 Task Priorities

At the beginning, BALS assigns each task a priority, which determines the order in which tasks are placed. In this work, we evaluate two prioritization metrics (higher value means higher priority):

**Upward Rank ( $rank_u$ ).** This metric calculates the length of the critical path from a given task to an exit node of the DAG. It is a common heuristic employed by list-scheduling algorithms, such as HEFT [22]. The upward rank of a task  $i$ , denoted  $rank_u(i)$ , is defined recursively as:

$$rank_u(i) = \bar{c}_i + \max_{j \in \text{succ}(i)} (\bar{d}_{ij} + rank_u(j))$$

where  $\bar{c}_i$  is the mean computation time of task  $i$  across all processors, and  $\bar{d}_{ij}$  is the mean communication cost of edge  $(i, j)$  across all processor pairs.

For an exit task with no successors,  $rank_u(i) = \bar{c}_i$ . When scheduling with this priority metric, schedules are often built *left to right*, with tasks scheduled to all available processors.

**Combined Rank ( $rank_c$ ).** This metric prioritizes tasks that lie on the overall critical path of the DAG by summing the upward rank with the downward rank:

$$rank_c(i) = rank_u(i) + rank_d(i)$$

The downward rank,  $rank_d(j)$ , represents the longest path from an entry node to task  $j$  and is defined as:

$$rank_d(j) = \max_{i \in \text{pred}(j)} (rank_d(i) + \bar{d}_{ij}) + \bar{c}_j$$

For an entry task,  $rank_d(j)$  is zero. The combined rank is used by algorithms such as CPoP [22]. When scheduling with this priority metric, tasks on the critical path are scheduled to a single processor first, followed by the tasks on the second-most critical path etc.

## 6 Superstep Elimination

We propose a post-processing optimization called **Superstep Elimination** that can be applied to any BSP schedule, regardless of which scheduler produced it. The optimization **iteratively removes supersteps by inlining (duplicating) their tasks to adjacent supersteps if beneficial**. This technique can be particularly effective for two reasons:

**Algorithm 3:** Barrier-Aware List Scheduler (BALS)

**Input:** Task graph  $G$ , BSP hardware with synchronization cost  $\sigma$ , snapping multiplier  $\kappa \geq 0$

- 1) **Prioritization.** Prioritize tasks with any standard policy (e.g., upward rank, combined rank).
- 2) **Placement.** Place the highest-priority *ready* task  $j$ :
  - (1) **Fill holes first.** Try to place  $j$  into an *existing* superstep and processor such that the step's compute duration does not increase. Among all legal holes, choose the one where  $j$ 's finishing time is earliest. A placement is legal if  $j$  does not need inter-processor data produced in the *same* superstep.
  - (2) **If no hole exists, place at the dependency-ready time.**
    - (i) **Compute finish times:** For each predecessor  $i \in \text{pred}(j)$ , let  $f_i$  be its finish time and  $t_{\text{end}}^{(s_i)}$  be the end time of its superstep  $s_i$ .
    - (ii) **Apply boundary snapping:** Compute the snapped finish time
 
$$\tilde{f}_i = \begin{cases} t_{\text{end}}^{(s_i)}, & \text{if } t_{\text{end}}^{(s_i)} - f_i < \kappa \cdot \sigma, \\ f_i, & \text{otherwise.} \end{cases}$$
    - (iii) **Determine dependency-ready time:**  $t_{\text{ready}} = \max_{i \in \text{pred}(j)} \tilde{f}_i$
    - (iv) **Place based on where  $t_{\text{ready}}$  falls:**
      - (i) **Inside a superstep:** Split that superstep at  $t_{\text{ready}}$ . Move all tasks starting after the split to a new superstep immediately following. Schedule  $j$  in the new superstep on the processor where it finishes earliest.
      - (ii) **Exactly on a superstep boundary:** Schedule  $j$  at the start of the later superstep on the processor where it finishes earliest.
      - (iii) **Beyond the current schedule:** Create a new superstep at the end and schedule  $j$  there on the processor where it finishes earliest.
    - (v) Merge the superstep that the task was placed in with its predecessor if legal to avoid unnecessary supersteps.

When estimating  $j$ 's finishing time on a candidate processor  $p$ , account for *both* the tasks compute time  $c_j(p)$  and additional *receive* time for its inter-processor predecessors ( $d_{ij}(\pi(i), p)$ , where  $\pi(i)$  is the processor of predecessor  $i$ ).

- (1) Most BSP schedulers don't take the synchronization delay into account during scheduling—the only exception being BALS with Boundary Snapping (Section 5.3). This can result in suboptimal schedules when the delay is large compared to task execution times. Paying the cost of duplicating computation can be worthwhile if it eliminates a synchronization barrier.
- (2) Second, task duplication alone—even if the synchronization delay is negligible—can result in better schedules, as we show in our evaluation. This is because it can be worth recomputing a task instead of communicating its result.

Our heuristic operates in a loop, continuing as long as it can find beneficial eliminations. In each pass, it identifies all "eliminatable" supersteps. A key constraint is that any superstep containing a sink

**Algorithm 4:** Superstep Elimination

**Input:** Schedule  $S$

**Output:** Optimized schedule  $S$

**Iterate until convergence:**

- (1) **Identify candidates.** Find all supersteps that do not contain any sink tasks. These are the eliminatable supersteps. If none exist, terminate.
- (2) **Sort candidates.** Sort the eliminatable supersteps by their total duration in increasing order.
- (3) **Attempt elimination.** For each candidate superstep  $s_{\text{elim}}$  in sorted order:
  - (a) Create a temporary copy of the schedule,  $S_{\text{trial}} = S$
  - (b) **Eliminate and Repair.** Remove  $s_{\text{elim}}$  from  $S_{\text{trial}}$ . For every subsequent superstep  $s_{\text{succ}}$  in  $S_{\text{trial}}$ , repair dependencies by finding tasks whose predecessors were in  $s_{\text{elim}}$  and recursively duplicating those predecessors into  $s_{\text{succ}}$ .
  - (c) **Evaluate.** Calculate the makespan of  $S_{\text{trial}}$ . If  $T_{\text{BSP}}(S_{\text{trial}}) < T_{\text{BSP}}(S)$ , then the elimination is beneficial.
  - (d) **Commit or Discard.** If beneficial, replace  $S$  with  $S_{\text{trial}}$

task (a task with no successors) cannot be eliminated, as its results must be preserved for the final output.

The algorithm then sorts the eliminatable supersteps by their total duration (smallest first) and attempts to remove them one by one. For each candidate superstep, it simulates the elimination, repairs all broken dependencies in subsequent supersteps via task duplication, and calculates the new makespan. If the new makespan is shorter, the change is committed, and the process restarts from the beginning. If not, the change is discarded, and the next candidate is tried.

The Superstep Elimination approach is detailed in Algorithm 4. The schedule of Figure 2c optimized by Superstep Elimination is visualized in Figure 2d.

## 7 Evaluation

We evaluate our schedulers against existing BSP schedulers using a benchmark suite built from combinations of task graphs and network graphs, resulting in 426 unique scheduling problems.

### 7.1 Network Graph

For the network graphs, we use a randomized heterogeneous model of the IPU. Examining the current generation IPU (Mk2) more closely, it is organized into islands and columns [8]. Each IPU has 16 columns, and each column contains 23 islands. Every island includes 4 tiles. Although all tiles are interconnected (fully connected graph), the communication latency varies between tiles in different islands, columns, and across IPU. To account for the slower latency between tiles in different groups, our IPU model has a link speed of 64 bits per cycle for tiles within the same island, 32 bits for tiles within the same column (but different islands), and 16 bits for tiles on the same IPU (but different columns). We schedule each generated task graph with IPU network graphs with 2, 4, 16, 32,

92, and 184 tiles (half island, one island, 4 islands, 8 islands, one column, and 2 columns).

## 7.2 Task Graphs

We use a diverse set of task graphs from three sources to ensure our results are robust and generalizable. An overview is provided in Table 1.

**Table 1: Task graphs used in our benchmarks. Every task graph variation is tested in combination with six hardware variations (2 tiles to 184 tiles).**

Name	Domain	Num. Variations	Num. Tasks
<i>Primitives</i>			
in-trees	Synthetic Graph	10	40 - 3906
out-trees	Synthetic Graph	10	40 - 3906
parallel chains	Synthetic Graph	10	100 - 1765
<i>Scientific Applications</i>			
blast	Bioinformatics [7]	5	43 - 303
bwa	Bioinformatics [7]	5	104 - 1004
cycles	Agroecosystem Modeling [20]	5	131 - 2181
epigenomics	Epigenomics [9]	5	261 - 793
genome	Bioinformatics [3]	5	260 - 902
montage	Astronomy [18]	5	100 - 1312
seismology	Seismology [4]	5	101 - 901
soykb	Agricultural Genomics [10]	5	96 - 544
srsearch	Bioinformatics [19]	5	22 - 104
<i>Machine Learning using Sum-Product Networks</i>			
NIPS-5	Density estimation [16]	1	13
nlts	Density estimation [11]	1	3641
plants	Species classification [11]	1	1297
speaker	Speaker identification [12]	1	1378

1. **WfCommons** provides open-source datasets of scientific workflow instances and tools for generating synthetic workflow instances [2]. The framework generates recipes based on execution traces of real applications. The scientific applications were run multiple times with different parameters, and execution traces (including execution times) were captured. WfCommons then builds task dependency graphs from those traces and detects common patterns. Based on those recipes, it can generate synthetic task graphs for an arbitrary number of tasks with the same characteristics as the real programs.

The weights in the WfCommons synthetic task graph and the IPU-based network graph are in no physical relation. Thus, we scale the weights of the network graph to match a given Communication-to-Compute Ratio:  $CCR = \bar{d}/\bar{c} = 0.1$ . Synchronization delay is chosen to be 1% of the average computation time:  $\sigma = 0.01 \cdot \bar{c}$ .

2. **Primitives**: To test performance on fundamental parallel patterns, we generate randomized in-trees, out-trees, and graphs

of parallel chains. For the in- and out-trees, the number of levels as well as the branching factor (number of successors/predecessors per node) is chosen randomly between 3 and 6. For the parallel chains, the number of parallel chains as well as the length of each chain is chosen randomly between 5 and 50. CCR and synchronization delay are set as with the WfCommons dataset.

3. **Sum-Product Networks (SPNs)** are probabilistic machine learning models that translate to DAGs of mostly sum and product operations [17]. As nodes represent operations on scalar values, they take only a few processor cycles to execute, making its execution very fine-grained. Research interest in SPNs has decreased in recent years, likely because widely used GPGPUs are very inefficient in calculating unstructured sparseness. On the IPU, all cores and threads are fully independent and can execute different instructions. Thus, it is exceptionally well suited for unstructured sparse computations. Additionally, its pre-scheduled inter-processor communication allows transmitting small messages efficiently. With this background, we evaluate our schedulers' performance on a common set of benchmark SPNs.

In our SPN benchmark, weights of task graphs (SPNs) and network graphs have physical meaning and are in relation to one another. The weights of nodes in the task graph represent the approximate number of instructions needed to calculate them; edge weight is the number of bits of result. In the IPU network, node weight is instructions per second, and edge weight is communication bandwidth in bit/s. Thus, the communication delay can be chosen in seconds too and is based on a realistic value (100 clock cycles at 1.8 GHz).

This makes the characteristics of the SPN benchmark very different from the other two: A single synchronization takes much longer (100 cycles) than the execution time of a task (a few cycles); and communication and computation times are of similar magnitude.

## 7.3 Reference Scheduler and Metrics

As the scheduling problem is NP-hard [23], obtaining an optimal "ground truth" schedule is intractable for non-trivial graphs. To navigate this, we evaluate performance using a normalized makespan ratio. For each test case, we run all schedulers and find the minimum makespan achieved by any of them. Each scheduler's result is then normalized by this best-observed makespan. A ratio of 1.0 signifies that the scheduler produced the best result among all competitors for that run, while a ratio of 5.0 means its schedule was five times worse.

To contextualize the performance of our BSP schedulers, we include results from a modified HEFT scheduler. Because its scheduling model requires no synchronization barriers—and thus also no synchronization delays—the resulting schedules usually have the smallest makespan, **setting the baseline** for the schedulers in the BSP model.

To make the results of the HEFT scheduler comparable to those in the BSP model, we applied two modifications to the standard HEFT scheduler:

- (1) **Blocking Communication**: First, the communication time needed to receive data dependencies from other processors becomes part of a task's duration. This causes that processors cannot compute other tasks while receiving data.

- (2) **Serial Data Transfers:** Second, a task’s total communication time is the **sum** of the transfer times of inter-processor data dependencies. In contrast, the delay model assumes that all data dependencies can be received in parallel (max of all transfer times).

This creates an asynchronous baseline that shares the more realistic non-overlapping communication constraints of our BSP model but without its global synchronization barriers.

## 7.4 Results

A heatmap visualizing makespan ratios of every testcase individually is shown in Figure 3. Per-scheduler mean results over all testcases are presented in Table 2 and Figure 4.

**Table 2: Aggregated benchmark results. Runtimes for our Python implementations in seconds; timeouts at 5 min. limit.**

Algorithm	Makespan Ratio			Sched. Runtime (s)			Time-outs
	Mean	Median	Dev.	Mean	Median	Dev.	
HEFT, delay-model	1.03	1.00	0.07	0.36	0.08	0.83	0
HEFT + EFN	1.76	1.40	1.27	0.81	0.13	2.11	0
HEFT + Eager	1.50	1.21	0.63	0.37	0.09	0.89	0
BALS Combined	1.19	1.05	0.41	1.55	0.21	4.69	0
BALS Comb. + Elim.	1.16	1.03	0.36	2.27	0.24	9.08	0
BALS Upward	1.16	1.06	0.31	1.50	0.22	4.29	0
BALS Upw. + Elim.	1.13	1.05	0.27	2.29	0.25	9.37	0
BALS Upw. Snap	1.13	1.07	0.22	1.48	0.21	4.44	0
<b>BALS Upw. Snap + Elim.</b>	<b>1.11</b>	<b>1.05</b>	<b>0.18</b>	1.86	0.24	5.03	0
HDagg ( $\epsilon=0.01$ )	6.25	1.72	12.7	0.93	0.05	5.06	0
HDagg ( $\epsilon=0.1$ )	6.25	1.72	12.7	0.94	0.05	5.13	0
BCSH (LDSH)	5.05	1.99	8.73	1.24	0.06	5.26	0
BCSH (EFT)	3.69	1.77	6.79	2.94	0.17	9.96	0
BSPg	7.25	1.28	15.1	1.09	0.09	3.33	10
Source	8.18	2.46	14.8	0.04	0.01	0.09	0
Multilevel (15%)	9.31	3.74	15.5	15.5	0.96	36.3	59
Multilevel (30%)	8.96	3.23	15.8	14.2	0.58	33.6	66

*Existing BSP Schedulers.* The existing BSP schedulers often struggle on our diverse benchmark suite. No single scheduler performs consistently well: BCSH (EFT) achieves the lowest mean ratio (3.69 $\times$ ), while BSPg achieves the lowest median (1.28 $\times$ ) but suffers from pathological outliers that inflate its mean to 7.25 $\times$ .

BCSH with the original LDSH placement variant (mean ratio 5.05 $\times$ ) consistently produces weak results, as its placement logic is not heterogeneity-aware. Our EFT placement variant offers a tangible improvement (mean ratio 3.69 $\times$ , median 1.77 $\times$ ) and notably performs reasonably well on SPNs, but remains uncompetitive on WfCommons and Primitives.

HDagg (mean ratio 6.25 $\times$ , median 1.72 $\times$ ) struggles on several datasets, particularly on in-trees (over 20 $\times$ ) and certain SPNs like speaker (over 20 $\times$ ). HDagg was designed for sparse matrix workloads with regular wavefront structure, and its coarsening strategy does not adapt well to irregular DAGs. We did not observe a significant impact from the load-balance threshold parameter  $\epsilon$ .

**Papp et al.’s schedulers** show varied performance. BSPg achieves the lowest median ratio among existing schedulers (1.28 $\times$ ), performing well on in-trees (ratio 1.09) and SPNs. Source (mean ratio 8.18 $\times$ ) achieves 1.06 on nltcs; SPNs naturally form in-tree-like structures, which suits Source’s topological wavefront approach. Importantly, Source is also extremely fast, making it an attractive choice for SPN workloads when scheduler runtime matters. However, both schedulers exhibit pathological behavior on out-trees: their greedy assignment strategies cause all tasks to cascade onto a single processor, resulting in ratios exceeding 24 $\times$ —explaining BSPg’s high mean despite its low median. The Multilevel scheduler times out on most of our problem instances, as its hill-climbing refinement becomes intractable at our problem sizes.

*Translation Strategies.* Our translation-based strategies significantly outperform the existing BSP schedulers. HEFT+Eager (mean 1.50 $\times$ , median 1.21 $\times$ ) outperforms the more complex HEFT+EFN (mean 1.76 $\times$ , median 1.40 $\times$ ). However, both are limited by their inability to respond to barrier-enforced idle times: once an async schedule is fixed, there is limited room for a translation algorithm to recover from placement decisions that are suboptimal for the BSP model.

*BALS Performance.* Our native BALS scheduler consistently produces the best BSP schedules, with the best variant achieving a mean ratio of 1.11 $\times$  and median of 1.05 $\times$ . The choice of task priority (Upward vs. Combined Rank) has only a modest impact, with no clearly superior option.

**Superstep Elimination** (Section 6) further reduces makespan by eliminating supersteps and duplicating their tasks into successor supersteps. This is particularly effective on datasets with large relative synchronization delays, such as SPNs—on the plants dataset, BALS with Upward Rank improves from 2.6 $\times$  to 2.1 $\times$  after Superstep Elimination. Task duplication can occasionally allow BALS to outperform even the asynchronous HEFT baseline (e.g., on blast), when recomputing a task is faster than communicating its result.

**Boundary Snapping** (Section 5.3) addresses excessive superstep fragmentation on fine-grained workloads where task runtimes are comparable to synchronization overhead. Using a snapping multiplier of  $\kappa = 10$ , the impact on SPN benchmarks is substantial: on plants, enabling Boundary Snapping reduces the makespan ratio from 2.98 $\times$  to 1.67 $\times$ ; on speaker, from 1.63 $\times$  to 1.13 $\times$ . Combined with Superstep Elimination, our best variant *BALS Upw. Snap + Elim.* achieves the lowest overall mean (1.11 $\times$ ) and median (1.05 $\times$ ). On coarse-grained workloads (WfCommons, Primitives), Boundary Snapping has minimal effect.

*Scheduler Runtimes.* We measured the execution time of our Python-based scheduler implementations for every schedule generated, with results reported in Table 2. The benchmarks were run on a system equipped with an AMD Ryzen 9 7950X CPU and 128 GB of RAM.

BALS achieves practical runtimes with a mean of 1.50–2.27 seconds across variants, comparable to BCSH (mean 1.24 s and 2.94 s) and a bit slower than HDagg (mean 0.93 s). The Source scheduler stands out as exceptionally fast (mean 0.04 s), making it attractive for large SPNs.

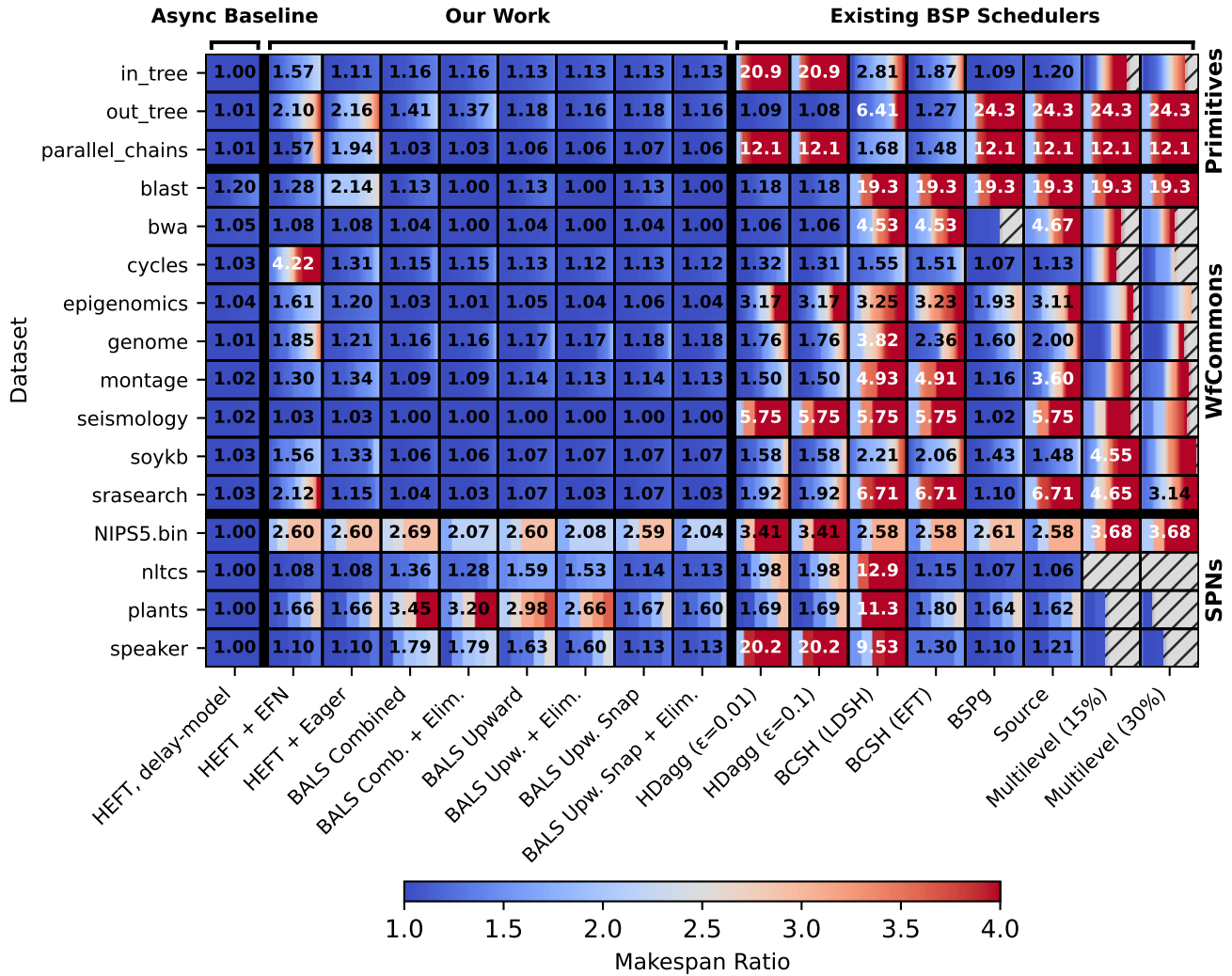


Figure 3: Makespan ratio comparison across all datasets. Cell gradients show the distribution across dataset variations (lower/blue is better); hatched portions indicate the fraction of timed-out runs (5 min. limit). The baseline (HEFT, delay-model) permits asynchronous communication, serving as a reference; all other schedulers produce BSP schedules. Visualization inspired by [1].

## 7.5 Discussion

We were surprised to see that our BALS scheduler can almost entirely mitigate the makespan impact of the BSP model, with makespans on average only 11% longer than the asynchronous baseline—or in some cases even outperform the HEFT scheduler despite the need for synchronization barriers.

We now examine additional aspects of our results:

**Existing BSP Scheduler Limitations:** The existing BSP schedulers each show pathological behavior on certain DAG structures. BCSH can only merge entire layers of tasks at once, preventing fine-grained placement decisions; its duplication strategy can also produce degenerate schedules that map all tasks to a single processor. HDagg’s wavefront-based coarsening assumes regular, layered

DAG structures typical of sparse matrix computations, but fails on irregular workflows. BSPg and Source both employ greedy strategies that work well on in-tree-like structures but exhibit pathological behavior on out-trees, where their locality optimizations cause all tasks to cascade onto a single processor.

**BALS Robustness:** In contrast to the existing schedulers, BALS exhibits no pathological edge cases. Across all 16 datasets in our benchmark—spanning in-trees, out-trees, parallel chains, scientific workflows, and fine-grained SPNs—BALS either produces the best result or remains competitive with the best-performing scheduler for that structure. This robustness stems from its adaptive placement strategy: by dynamically splitting supersteps and filling idle gaps, BALS naturally handles diverse DAG topologies without relying on assumptions about graph structure.

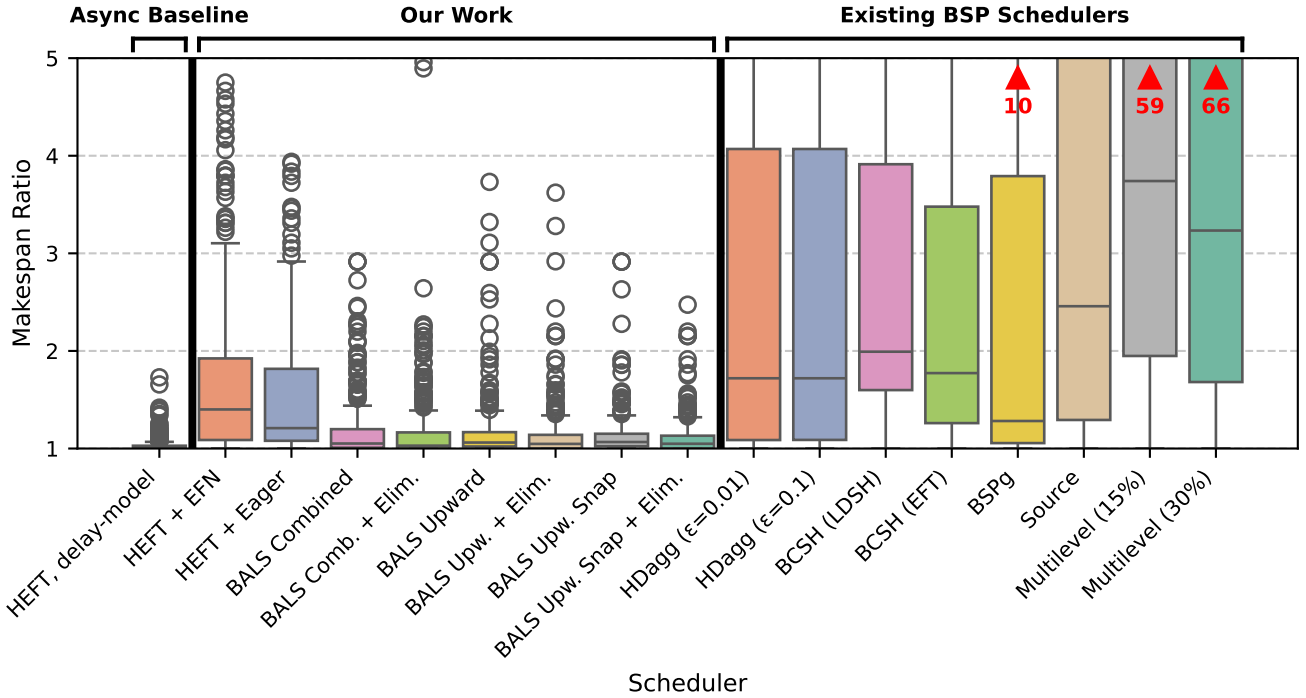


Figure 4: Makespan ratio distribution across all benchmarks. Red triangles indicate timeout counts (5 min. limit).

**Interpreting SPN Results:** We believe that the larger makespan ratios for Sum-Product Networks relative to the delay-model baseline do not indicate poor BSP scheduling. Instead, they reflect limitations of the baseline itself. SPNs transfer 32-bit scalars, where message-passing overhead (headers, protocols, buffering) dominates actual data transmission time. The delay model ignores these costs, producing unrealistically optimistic makespans. Our BSP model explicitly accounts for these overheads through synchronization delays and blocking communication. We expect BSP makespan values to closely match real IPU performance, while the delay model significantly underestimates execution time for fine-grained workloads.

## 7.6 Scheduler Recommendations

Based on our findings, we offer the following practical recommendations for scheduling on BSP-style architectures:

*For coarse-grained workloads (e.g., scientific workflows).* Our BALS scheduler with Upward Rank, Boundary Snapping, and Superstep Elimination consistently produces the best schedules across diverse DAG structures. Unlike existing BSP schedulers, BALS exhibits no pathological edge cases—it either produces the best result or remains competitive across all graph topologies.

*For fine-grained workloads (e.g., SPNs, ML models).* The Source scheduler produces competitive results on SPN-like structures while being 10–100× faster than BALS, making it the recommended choice for large fine-grained models. If the time budget allows, combining

Source with our Superstep Elimination post-processing may further improve schedule quality.

## 8 Conclusion

In this paper, we addressed static task scheduling for the Bulk-Synchronous Parallel (BSP) model, where all processors must synchronize before any inter-processor communication can take place. While Graphcore’s Intelligence Processing Unit (IPU) demonstrates that this model enables processor designs with extraordinary hardware efficiency, it fundamentally changes the scheduling problem.

Our evaluation on 426 unique problems shows that our proposed Barrier-Aware List Scheduler (BALS) reduces **the makespan gap between BSP and asynchronous scheduling to only 11% on average**. BALS achieves this by scheduling *natively* in the BSP model, steering placement decisions as synchronization-induced imbalances emerge. This confirms that the hardware efficiency offered by BSP-style architectures does *not* need to come at the cost of poor application parallelization.

A key strength of BALS is its robustness: unlike existing BSP schedulers, which each exhibit pathological behavior on certain DAG structures (BCSH on irregular graphs, HDagg on in-trees, BSPg and Source on out-trees), BALS *consistently* produces the *best* or *near-best* schedules across *all* graph topologies.

For fine-grained workloads like SPNs, BALS’s *Boundary Snapping* option reduces excessive superstep fragmentation when task

runtimes are comparable to synchronization overhead. Our *Superstep Elimination* post-processing removes barriers by duplicating tasks, proving valuable when recomputation is faster than communication—and can be applied to any BSP schedule.

Ultimately, this work provides strong evidence that the BSP model, when paired with appropriate scheduling algorithms, represents a highly viable paradigm for parallel processor architectures. By demonstrating that its scheduling constraints can be effectively managed, we open the door for broader adoption of architectures that trade the hardware overhead of asynchronous communication for raw, deterministic communication bandwidth.

## 9 Acknowledgements

We gratefully acknowledge support by the Federal Ministry of Education and Research (BMBF) under Grant No. 01IS22091.

## References

- [1] Jared Coleman and Bhaskar Krishnamachari. 2025. Pisa: an adversarial approach to comparing task graph scheduling algorithms. (2025). <https://arxiv.org/abs/2403.07120> arXiv: 2403.07120 [cs.DC].
- [2] Tainã Coleman, Henri Casanova, Loïc Pottier, Manav Kaushik, Ewa Deelman, and Rafael Ferreira da Silva. 2022. WfCommons: A Framework for Enabling Scientific Workflow Research and Development. *Future Generation Computer Systems*, 128, 16–27. doi:10.1016/j.future.2021.09.043.
- [3] Rafael Ferreira da Silva, Rosa Filgueira, Ewa Deelman, Erola Pairo-Castineira, Ian M. Overton, and Malcolm P. Atkinson. 2019. Using simple PID-inspired controllers for online resilient resource management of distributed scientific workflows. *Future Generation Computer Systems*, 95, 615–628. doi:<https://doi.org/10.1016/j.future.2019.01.015>.
- [4] Rosa Filgueira, Rafael Ferreira da Silva, Amrey Krause, Ewa Deelman, and Malcolm Atkinson. 2016. Asterism: pegasus and dispel4py hybrid workflows for data-intensive science. In *Proceedings of the 7th International Workshop on Data-Intensive Computing in the Cloud (DataCloud '16)*. IEEE Press, Salt Lake City, Utah, 1–8. ISBN: 9781509061587.
- [5] Graphcore. [n. d.] Hardware overview - IPU Programmer's Guide. (). [https://docs.graphcore.ai/projects/ipu-programmers-guide/en/latest/about\\_ipu.html](https://docs.graphcore.ai/projects/ipu-programmers-guide/en/latest/about_ipu.html).
- [6] Takashi Hashimoto, Kenichi Hagihara, Tomoki Baba, and Noriyuki Fujimoto. 1999. A Task Scheduling Algorithm to Package Messages on Distributed Memory Parallel Machines. In *Parallel Architectures, Algorithms, and Networks, International Symposium on*. IEEE Computer Society, Los Alamitos, CA, USA, (June 1999), 236. doi:10.1109/ISPAN.1999.778945.
- [7] Nick Hazekamp and Douglas Thain. 2017. Makeflow examples repository. Repository of sample workflows for use with the Makeflow workflow system. (2017). <http://github.com/cooperative-computing-lab/makeflow-examples>.
- [8] Zhe Jia, Blake Tillman, Marco Maggioni, and Daniele Paolo Scarpazza. 2019. Dissecting the graphcore ipu architecture via microbenchmarking. (2019). <https://arxiv.org/abs/1912.03413> arXiv: 1912.03413 [cs.DC].
- [9] Gideon Juve, Ann Chervenak, Ewa Deelman, Shishir Bharathi, Gaurang Mehta, and Karan Vahi. 2013. Characterizing and profiling scientific workflows. *Future Generation Computer Systems*, 29, 3, 682–692. Special Section: Recent Developments in High Performance Computing and Security. doi:<https://doi.org/10.1016/j.future.2012.08.015>.
- [10] Yang Liu et al. 2016. PGen: large-scale genomic variations analysis workflow and browser in SoyKB. en. *BMC Bioinformatics*, 17, Suppl 13, (Oct. 2016), 337.
- [11] Daniel Lowd and Jesse Davis. 2010. Learning markov network structure with decision trees. In *2010 IEEE International Conference on Data Mining*, 334–343. doi:10.1109/ICDM.2010.128.
- [12] Aaron Nicolson and Kuldip Paliwal. 2020. Sum-product networks for robust automatic speaker identification. In (Oct. 2020), 1516–1520. doi:10.21437/Interspeech.2020-1501.
- [13] Tim Noack. 2026. *bsp\_scheduling* GitHub Repository. [https://github.com/esa-tu-darmstadt/bsp\\_scheduling](https://github.com/esa-tu-darmstadt/bsp_scheduling). (2026).
- [14] Pál András Papp, Georg Anegg, Aikaterini Karanasiou, and Albert-Jan N. Yzelman. 2024. Efficient multi-processor scheduling in increasingly realistic models. In *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '24)*. Association for Computing Machinery, Nantes, France, 463–474. ISBN: 9798400704161. doi:10.1145/3626183.3659972.
- [15] Pál András Papp, Georg Anegg, and Albert-Jan N. Yzelman. 2025. DAG Scheduling in the BSP Model. In *SOFSEM 2025: Theory and Practice of Computer Science: 50th International Conference on Current Trends in Theory and Practice of Computer Science, SOFSEM 2025, Bratislava, Slovak Republic, January 20–23, 2025, Proceedings, Part II*. Springer-Verlag, Bratislava, Slovakia, 238–253. ISBN: 978-3-031-82696-2. doi:10.1007/978-3-031-82697-9\_18.
- [16] Valerio Perrone. 2016. NIPS Conference Papers 1987-2015. UCI Machine Learning Repository. DOI: <https://doi.org/10.24432/C5KC80>. (2016).
- [17] Hoifung Poon and Pedro Domingos. 2011. Sum-product networks: a new deep architecture. In *2011 IEEE International Conference on Computer Vision Workshops (ICCV Workshops)*, 689–690. doi:10.1109/ICCVW.2011.6130310.
- [18] M. Rynge, G. Juve, J. Kinney, J. Good, B. Berriman, A. Merrihew, and E. Deelman. 2014. Producing an Infrared Multiwavelength Galactic Plane Atlas Using Montage, Pegasus, and Amazon Web Services. In *Astronomical Data Analysis Software and Systems XXIII* (Astronomical Society of the Pacific Conference Series). N. Manset and P. Forshay, (Eds.) Vol. 485. (May 2014), 211.
- [19] Mats Rynge. 2017. SRA search pegasus workflow. Repository of sample workflows for use with the Makeflow workflow system. (2017). <https://github.com/pegasus-isi/sra-search-pegasus-workflow>.
- [20] Rafael Ferreira da Silva, Rajiv Mayani, Yuning Shi, Armen R. Kemanian, Mats Rynge, and Ewa Deelman. 2019. Empowering agroecosystem modeling with htc scientific workflows: the cycles model use case. In *2019 IEEE International Conference on Big Data (Big Data)*, 4545–4552. doi:10.1109/BigData47090.2019.9006107.
- [21] Oliver Sinnen. 2007. *Task Scheduling for Parallel Systems (Wiley Series on Parallel and Distributed Computing)*. Wiley-Interscience, USA. ISBN: 0471735760.
- [22] H. Topcuoglu, S. 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, 260–274. doi:10.1109/71.993206.
- [23] J.D. Ullman. 1975. Np-complete scheduling problems. *Journal of Computer and System Sciences*, 10, 3, 384–393. doi:[https://doi.org/10.1016/S0022-0000\(75\)80008-0](https://doi.org/10.1016/S0022-0000(75)80008-0).
- [24] Leslie G. Valiant. 1990. A bridging model for parallel computation. *Commun. ACM*, 33, 8, (Aug. 1990), 103–111. doi:10.1145/79173.79181.
- [25] 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)*, 1217–1227. doi:10.1109/IPDPS53621.2022.00121.