A Novel Heuristic for Directed Acyclic Graph Task Scheduling using Longest Betweenness Centrality

N.P. DAMINK, University of Twente, The Netherlands

Task scheduling is a well-known NP-hard problem that involves efficiently allocating computational tasks across available resources. Existing heuristic approaches, such as HEFT and MinMin, typically rely on basic task properties and may fail to capture deeper structural dependencies in the task graph. In this work, we propose a novel list-scheduling heuristic based on Longest Betweenness Centrality (LBC), a metric designed to quantify a task's influence by evaluating its presence on long dependency paths. We introduce six LBC-based ranking methods, including variants that are source-based and successor-weighted. The most promising variant, LBC-SRL, estimates task criticality by analyzing each task's current critical dependencies. Experimental evaluations across five synthetic DAG generation models and multiple processor configurations demonstrated that LBC-SRL consistently outperforms classical heuristics such as MinMin, HCPT and PEFT. On DAGs with a high density, LBC-SRL achieves performance comparable to HEFT. Although HEFT remains the top performer overall, pairwise comparisons reveal that LBC-SRL frequently matches its performance and occasionally even surpasses it. These findings illustrate the potential of incorporating global graph-theoretical metrics into task scheduling.

Additional Key Words and Phrases: Task Scheduling, Directed Acyclic Graphs (DAGs), Longest Betweenness Centrality (LBC), Parallel Computing, Makespan, List Scheduling Algorithms, Scheduling Heuristics, Static Scheduling, Graph Centrality

1 INTRODUCTION

Task scheduling is a fundamental problem in computer science with widespread applications in domains such as cloud computing [4], data processing [14], and real-time systems [20]. In the modern era, single processor systems are often insufficient for processing the huge amounts of data generated. This has led to the emergence of parallel and distributed computing environments, where multiple processors or computing nodes execute tasks concurrently [7]. Although these environments significantly increase computational power, they also introduce considerable complexity into the scheduling process. Task scheduling involves assigning computational tasks to available resources, such as multiple CPUs, distributed nodes, or virtual machines. The primary goal is to optimize performance metrics, such as minimizing makespan or balancing load distribution across processors. Suboptimal scheduling can result in resource under-utilization and increased execution times. In large-scale environments, even small inefficiencies can lead to significant performance bottlenecks, which in practice may result in increased operational costs. Therefore, effective task scheduling is of great importance in modern computing environments. However, many existing task scheduling heuristics rely on simple task properties and fail to account for the global structure of the task graph. To address this limitation, we propose a novel scheduling approach that leverages a new centrality metric to capture deeper structural dependencies.

TScIT 43, July 4, 2025, Enschede, The Netherlands

1.1 Contributions

This paper makes the following key contributions:

- It introduces the novel graph-theoretic metric LBC, designed to quantify task importance based on their presence along long dependency paths in DAGs.
- It proposes several LBC-based task scheduling heuristics and formally analyzes their computational complexity.
- It presents a comprehensive empirical evaluation comparing the proposed LBC-based heuristics to established scheduling heuristics across a diverse set of synthetic DAG topologies and processor counts.
- It demonstrates the value of incorporating global structural information into task scheduling decisions.

1.2 Problem Statement

This study considers the task scheduling problem $P \mid prec \mid C_{\max}$ in Graham's notation [15]. This notation represents a scheduling problem on parallel machines with identical processors (P), indicating that all processors execute any task with the same speed. Each task is associated with a processing cost and is subject to precedence constraints ($prec$). Finally, the objective of this problem is to minimize the makespan ($C_{\max}$). This study focuses specifically on static scheduling, in which all tasks and their dependencies are known ahead of scheduling. Unlike dynamic scheduling, which makes decisions at runtime, static scheduling allows optimization based on the complete structure of the task DAG.

Since this problem is NP-complete [27], it is believed that no algorithm can find the optimal solution in polynomial time. Consequently, heuristic algorithms are essential for finding efficient and scalable solutions.

This paper investigates how the Longest Betweenness Centrality (LBC) metric, introduced in section 4.1, can be effectively integrated into heuristic task scheduling algorithms for parallel and distributed computing systems. To address this main research objective, we will investigate the following key questions:

- How do different LBC-based ranking strategies affect the makespan of task scheduling across diverse DAG structures?
- How does the computational complexity of the proposed LBC-based scheduling algorithms compare to that of state-of-the-art methods such as HEFT, PEFT, MinMin, and HCPT?
- How do the proposed LBC-based heuristics compare to state-of-the-art scheduling algorithms in terms of makespan performance across various task graph types and processor configurations?

The remainder of this paper is structured as follows. Section 2 present the foundational concepts upon which this paper is built. Section 3 reviews the state-of-the-art literature on heuristic scheduling algorithms and identifies key limitations in existing methods. Section 4 introduces the Longest Betweenness Centrality (LBC) metric and several variants of the novel LBC-based heuristic. Theoretical formulations and complexity analyses are provided to establish their computational feasibility. Section 5 outlines the

experimental methodology, including details on DAG generation and evaluation parameters. Section 6 presents the empirical results, comparing the proposed heuristics against established algorithms across a range of scenarios. Finally, Section 7 discusses the findings, limitations, and directions for future research.

2 BACKGROUND

In task scheduling, an application's components and their dependencies can be represented by a Directed Acyclic Graph (DAG). Formally, a task DAG is a graph $G = (V, E)$, where $V = \{t_1, t_2, \ldots, t_n\}$ is a finite set of tasks and E is a set of directed edges representing dependencies between these tasks. We denote the number of tasks as $n = |V|$ and the number of dependencies as $e = |E|$. Each task $t_i \in V$ is associated with a processing cost c_i. This represents the time or computational effort required to execute the task. Each directed edge from task t_i to task t_j, denoted as $(t_i, t_j) \in E$, indicates that t_i must be completed before task t_j begins. By definition, a DAG does not contain cycles, which means that there is no sequence of tasks $t_{i_1}, t_{i_2}, \ldots, t_{i_k}, t_{i_1}$ such that each $(t_{i_l}, t_{i_{l+1}}) \in E$. Thus, the structure of a DAG directly represents the order in which tasks must be executed. DAG-based models are especially useful in parallel and distributed computing environments, where tasks can run concurrently if dependencies are respected.

One of the main objectives in task scheduling is minimizing the makespan. Makespan (C_{max}) is commonly used as a performance metric to evaluate scheduling algorithms. It indicates the completion time of the last task [22]. Effectively, it measures the total time required to complete a set of tasks. Note that this is different from the cumulative processing time of all tasks, as our parallel computing environment allows multiple tasks to execute concurrently. Formally, the makespan can be defined as

$$C_{max} = \max_{i \in tasks} C_i \tag{1}$$

where C_i is the finishing time of task i and $tasks$ is the set of all tasks to be scheduled.

We now introduce key definitions that will be used throughout the remainder of this paper.

Definition 1: Let $Pred(t_i)$ denote the set of direct predecessors of the task t_i. Formally, in the context of DAGs, this set is defined as:

$$Pred(t_i) = \{t_j \mid (t_j, t_i) \in E\} \tag{2}$$

Definition 2: Let $Succ(t_i)$ denote the set of direct successors of the task t_i. Formally, in the context of DAGs, this set is defined as:

$$Succ(t_i) = \{t_j \mid (t_i, t_j) \in E\} \tag{3}$$

We further define $Succ^*(t_i)$ as the set of all successors of t_i. That is, the set of all tasks $t_j \in V$ such that there exists a directed path from t_i to t_j in the DAG.

Definition 3: Let the *critical path* denote the longest path from an entry task to an exit task within the DAG. Formally, let the sets of entry and exit tasks be defined as:

$$T_{entry} = \{t_i \mid Pred(t_i) = \emptyset\} \tag{4}$$

$$T_{exit} = \{t_i \mid Succ(t_i) = \emptyset\} \tag{5}$$

Then, the *critical path* is the path of maximum total weight among all directed paths from any task in T_{entry} to any task in T_{exit}. This path determines the minimum completion time of the entire task DAG and forms the bottleneck for scheduling.

Definition 4: Let $EST(t_i, p_j)$ represent the earliest start time of task t_i on processor p_j:

$$EST(t_i, p_j) = \max \left\{ T_{available}(p_j), \max_{t_p \in Pred(t_i)} \left(AFT(t_p) + c_{p,i} \right) \right\} \tag{6}$$

where $T_{available}(p_j)$ is the earliest time at which processor p_j is ready for task execution, $AFT(t_p)$ is the actual finish time of predecessor t_p, and $c_{p,i}$ is the communication cost between tasks t_p and t_i. Note that $c_{p,i} = 0$, as communication costs are not considered in our study.

Definition 5: Let $EFT(t_i, p_j)$ represent the earliest finish time of task t_i on processor p_j:

$$EFT(t_i, p_j) = EST(t_i, p_j) + w_{i,j} \tag{7}$$

where $w_{i,j}$ is the processing cost of task t_i on processor p_j. However, we assume a homogeneous processor environment, meaning that each processor executes tasks with the same efficiency. Consequently, the cost of executing task t_i on any processor p_j is uniform and given by $w_{i,j} = c_i$.

3 STATE-OF-THE-ART

Numerous task scheduling heuristics have been developed in prior work. These heuristics are commonly categorized into clustering-based, duplication-based, and list-based scheduling algorithms. Clustering-based algorithms aim to minimize communication costs by grouping related tasks onto the same processor. Duplication-based methods aim to reduce data transfer latency by replicating the predecessors of tasks across processors. Finally, list-based scheduling algorithms assign priorities to tasks and schedule them according to these priorities [2].

Beyond these classical categories, recent research has explored metaheuristic techniques, such as genetic algorithms [21] and particle swarm optimization [5], as well as machine learning-based methods, like reinforcement learning [25]. These methods are particularly effective in heterogeneous or cloud environments, where dynamic scheduling plays a significant role.

Among these categories, list-based scheduling algorithm have consistently demonstrated superior performance in static and homogeneous environments [2], which aligns with the scope of this study. Consequently, list-based approaches are the primary focus of this work.

Well-known list-based scheduling algorithms include MinMin [16], HEFT [26], HCPT [17], and PEFT [3]. These algorithms mainly rely on basic task graph properties, such as task execution time, immediate dependencies, or placement on the critical path. Although such properties can sometimes be effective for efficiently computing a near-optimal solution, they fail to exploit deeper structural characteristics of DAGs. As a result, these methods may overlook more optimal scheduling opportunities that can arise from global graph patterns. Therefore, a gap exists in leveraging advanced graph-theoretic concepts to make scheduling decisions. This work aims to address this gap by proposing a novel heuristic that incorporates both global and local structural properties of DAGs to improve scheduling performance. The following section will provide a more detailed description of existing algorithms, which will serve as benchmarks in the comparative evaluation of our proposed approach.

3.1 MinMin

MinMin [16] is a greedy task scheduling heuristic that aims to minimize makespan by greedily assigning tasks that have the minimum EFT among all ready tasks. Specifically, for each ready task, it computes the EFT on all available processors and selects the task-processor pair that results in the minimum EFT. This process repeats until all tasks have been assigned.

3.2 HCPT

HCPT (Heterogeneous Critical Path Task scheduling) [17] consists of two phases, a ranking phase and a processor selection phase. In the ranking phase, tasks are prioritized based on their depth and position on the critical path. It schedules tasks in decreasing order of depth, giving precedence to those that are deep ancestors of nodes on the critical path. In the processor selection phase, HCPT applies an EFT scheduling strategy. For each task, it selects the processor time slot that yields the earliest finish time while respecting precedence and availability constraints.

3.3 HEFT

HEFT (Heterogeneous Earliest Finish Time) [26] also follows these two main phases. In the ranking phase, tasks are prioritized based on their upward rank, which estimates the longest path from the task to the exit node. This metric can be interpreted as a reverse depth, highlighting tasks that lie on or near the critical path. During the processor selection phase, HEFT applies an EFT scheduling strategy similar to HCPT. However, HEFT also supports task insertion. This allows tasks to be scheduled between existing time slots, provided that precedence and availability constraints are maintained.

3.4 PEFT

PEFT (Predict Earliest Finish Time) [3] also consists of two primary phases. In the ranking phase, tasks are prioritized using an Optimistic Cost Table (OCT). Each entry optimistically estimates the EFT by considering both computation and communication costs along critical paths. Tasks are then scheduled in descending order of their rank, where the rank is defined as the average value of the task's entries in the OCT. In the processor selection phase, PEFT assigns tasks to processors by estimating the EFT based on predicted values. It uses a similar EFT insertion technique as HEFT. However, it improves task scheduling by using more accurate finish time predictions. These predictions better capture communication delays, leading to improved load balancing and increased efficiency.

4 PROPOSED METHOD: LBC-BASED SCHEDULING HEURISTICS

Algorithm 1 presents the structure of the proposed LBC-based scheduling heuristic, which is divided into two main phases. In the ranking phase, a supernode s^* is added using Add-Super-Node(G), after which task scores are computed using one of the two proposed *LBC* variants. These scores are then refined using one of six scoring strategies detailed in Section 4.2.1, applied through Apply-Scoring-Strategy($scores, G$). The introduced supernode is then removed from the final ranking. In the processor selection phase, tasks are scheduled onto processors using an insertion-based EFT policy, implemented as Schedule($G, ranking, P, C$). The following sections provide a detailed explanation of the underlying theory behind the LBC prioritization metric, followed by a discussion of each phase in more detail.

Algorithm 1 Main pseudocode for the LBC-based heuristics.

```
function LBC-Heuristic(Graph G, Processors P, Costs C)
    G ← Add-Super-Node(G)
    scores ← LBC_(s)(G, C)
    scores ← Apply-Scoring-Strategy(scores, G)
    ranking ← Score-Guided-Ranking(G, scores)
    remove s* from ranking
    schedule ← Schedule(G, ranking, P, C)
    return schedule
end function
```

4.1 Longest Betweenness Centrality

The scheduling heuristic is built around a novel metric called Longest Betweenness Centrality (LBC). Traditional betweenness centrality measures the importance of a node within a network by counting how often it lies on the shortest paths between pairs of nodes [6]. In contrast, LBC redefines this concept by focusing on the longest paths. We define the longest path between two nodes t_i and t_j as the path with the maximum total weight among all directed paths from t_i to t_j. The existence of these paths is guaranteed by the acyclic nature of DAGs. The LBC for a node v can be formally defined as follows:

$$LBC(v) = \sum_{\substack{s,t \in V \\ s \neq v \neq t}} \frac{L_{st}(v)}{L_{st}} \tag{8}$$

where L_{st} is the number of longest paths from node s to node t and $L_{st}(v)$ is the number of those paths that pass through v.

Makespan is primarily affected by the longest execution paths in the DAG. Therefore, effective scheduling requires identifying tasks that frequently occur along those paths. The LBC metric captures this by prioritizing tasks that contribute the most to the DAG's critical structure. Unlike most state-of-the-art methods, LBC also considers the influence of tasks on long, non-critical paths, which can significantly affect the makespan. An illustrative example of how LBC is computed on a small weighted DAG is provided in Appendix C.

Brandes' algorithm [8] is widely recognized as one of the most efficient methods for computing standard betweenness centrality. We adapt a variation of this algorithm to compute LBC. Specifically, the new variant tracks maximum path lengths and counts the number of such longest paths that pass through each node. Moreover, the BFS or Dijkstra algorithm is replaced with a topological traversal to ensure that nodes are processed in the correct order for DAGs. This adaptation maintains the polynomial-time efficiency of the original algorithm, making it suitable for large task DAGs. The full pseudocode for this adaptation can be found in Appendix B.1.

In addition to the general LBC metric, we introduce a variant called Source-based Longest Betweenness Centrality (LBC_s). This version focuses on the influence of a node on the longest paths originating from entry tasks in the DAG. This approach still captures tasks that are critical to long execution chains, but avoids the computational overhead of evaluating all node pairs.

The LBC_s score for a node v is defined as:

$$LBC_s(v) = \sum_{\substack{t \in V,\ s \in T_{entry} \\ t \neq v \neq s}} \frac{L_{st}(v)}{L_{st}} \tag{9}$$

The full pseudocode for this adaptation can be found in Appendix B.2.

4.2 Ranking Phase

4.2.1 Scoring Strategies for Task Prioritization. To determine the most effective use of LBC, this paper explores six task scoring approaches based on the metric. For each approach, a supernode s^*, with edges to all entry nodes, is added to the DAG. Formally, the new augmented DAG is defined as:

$$G' = (V \cup \{s^*\},\ E \cup \{(s^*, v) \mid v \in V,\ Pred(v) = \emptyset\}) \tag{10}$$

In this augmented DAG, the new vertex s^* has outgoing edges to all nodes $v \in V$ for which $Pred(v) = \emptyset$. The main advantage of introducing this supernode is that the source nodes are included in the longest path computations, allowing them to be properly quantified.

LBC: The first scoring approach directly uses the LBC score as the priority score for each node. The idea is that tasks with high LBC scores lie on many long dependency chains, contributing the most to the total execution time. Using this approach, the score of a node can be defined as:

$$Score(i) = LBC(i) \tag{11}$$

LBC Source-based (LBC-S): This second scoring approach uses the source-based variant of the LBC algorithm. Priority scores are computed analogously to the previous method, but using the source-based LBC metric:

$$Score(i) = LBC_s(i) \tag{12}$$

This approach only considers the longest paths that originate from the starting node, which is sufficient for our goal of minimizing these paths.

LBC Source-based with Direct Successors (LBC-SDS): This scoring approach builds upon the source-based variant of the LBC algorithm. However, the score also incorporates the scores of its dependent successor tasks. Specifically, the score is defined as a combination of the LBC_s score of the task and its immediate successors:

$$Score(i) = LBC_s(i) + \sum_{s \in Succ(i)} LBC_s(s) \tag{13}$$

The underlying rationale is to prioritize tasks that lead to successors with higher scores. This promotes the execution of tasks that contribute more significantly to the overall execution time.

LBC Source-based with All Successors (LBC-SAS): This scoring approach extends the previous algorithm by considering not only the immediate successors of a task, but all its transitive successors. Specifically, the score is computed as the sum of the LBC_s scores of the task itself and of all of its downstream tasks:

$$Score(i) = LBC_s(i) + \sum_{s \in Succ^*(i)} LBC_s(s) \tag{14}$$

This approach places even more emphasis on tasks that contribute to a larger number of high-scoring successors.

LBC Source-based with Weighted Successors (LBC-SWS): This approach further extends the previous algorithm by introducing a weighted scoring mechanism for the successor tasks. It draws inspiration from the work of Lin et al. [19], who proposed a weighted out-degree (WOD) metric to determine task priorities. This approach prioritizes tasks with a high out-degree and gives additional weight to those whose successor nodes have a low in-degree. Building on this concept, our approach adopts a similar, but reversed rationale. We will be prioritizing tasks with a high in-degree, as their high number of dependencies can delay execution and significantly impact the makespan if scheduled late. The score is defined as follows:

$$Score(i) = LBC_s(i) + \sum_{s \in Succ^*(i)} LBC_s(s) \cdot |Pred(s)| \tag{15}$$

This approach emphasizes tasks that unlock access to highly connected successors, which are likely to become bottlenecks in later execution stages.

LBC Source-based Repeated Loop (LBC-SRL): The final scoring approach introduces an iterative variant of the source-based LBC method. Instead of computing all scores in a single pass, LBC-SRL recomputes the LBC_s scores at each step of the ranking process. At each iteration, the algorithm identifies all source nodes and computes their source-based LBC scores using the current state of the DAG. The highest-scoring node is selected, added to the ranking, and removed from the graph. This process is repeated until all nodes are scheduled. As this procedure involves iterative graph updates and score recalculations, it is better expressed procedurally rather than with a simple formula. The complete pseudocode is provided in Appendix B.3. Although this design increases the computational overhead, it enables more informed scheduling decisions. At each step, the algorithm selects the task upon which the most remaining tasks depend, thereby prioritizing the tasks that are most critical for subsequent execution stages.

In summary, these LBC-based strategies differ primarily in how they incorporate downstream influence: LBC and LBC-S focus solely on the centrality of the task, while the LBC-SDS, LBC-SAS, and LBC-SWS variants progressively expand the scope of influence from direct to weighted transitive successors. LBC-SRL stands out by recalculating the influence after each scheduling step, allowing the task prioritization to adapt to the evolving DAG structure.

4.2.2 Score-Guided Topological Scheduling. After computing the scores using one of the described scoring approaches, the final ranking is determined using a score-guided topological sort. During the traversal, tasks are ordered in a ranking that respects all precedence constraints. At each step, the algorithm selects the highest-scoring ready task using a priority queue implemented as a heap. Note that this ranking procedure is applied to all approaches except LBC-SRL, which already constructs the ranking iteratively during the score computation. The detailed pseudocode for this ranking procedure can be found in Appendix B.4.

4.3 Processor Selection Phase

The processor selection phase follows the task order of the generated ranking. This phase applies a similar insertion-based scheduling strategy used in HEFT [26]. Each task is scheduled on the processor that provides the lowest EFT. Task insertion between existing tasks is allowed, provided that dependency and availability constraints are satisfied. The detailed pseudocode for this scheduling procedure is provided in Appendix B.5.

4.4 Complexity Analysis

Table 1 presents the asymptotic time complexity of the scheduling algorithms evaluated in this study. In this table, we denote the number of tasks as $n = |V|$, the number of edges as $e = |E|$, and the number of processors as $p = |P|$. These complexity bounds are essential for evaluating the practical scalability of each approach, especially when applied to large task graphs.

The LBC and LBC-SRL heuristics have a worst-case time complexity of $O(n^2 \cdot p + n \cdot e)$. The term $O(n \cdot e)$ originates from the calculation of the LBC scores, with each LBC_s computation requiring $O(n+e)$ time. When repeated across n tasks, this results in a total of $O(n^2 + n \cdot e)$ time. However, the term $O(n^2)$ is asymptotically dominated by the processor selection phase, which has a time complexity of $O(n^2 \cdot p)$. In this phase, each task is compared against all already scheduled tasks on every processor to identify the optimal insertion point. Consequently, the total complexity for these two algorithms is $O(n^2 \cdot p + n \cdot e)$. In dense graphs, where $e \approx n^2$, the traversal term dominates, leading to a complexity of $O(n^3)$. In contrast, in sparser graphs or systems with many processors, the insertion phase dominates the complexity.

The remaining LBC variants have an overall complexity of $O(n^2 \cdot p)$. These approaches simplify the scheduling phase by using a lighter LBC_s with complexity $O(n + e)$ and a sorting step with $O(n \log n)$. Neither outweighs the cost of the insertion phase, which remains the asymptotically dominant component.

Similarly, the complexities of HEFT and PEFT are dominated by the insertion phase, resulting in a total complexity of $O(n^2 \cdot p)$.

For each scheduled task, MinMin evaluates all ready tasks across all processors to find the minimum EFT, leading to a time complexity of $O(n^2 \cdot p)$.

Finally, HCPT uses a simpler insertion-free selection mechanism, reducing its complexity to $O(n \cdot p)$. The dominant step for HCPT is the task ranking, which takes $O(n \log n)$ time.

Algorithms	Total Complexity
LBC, LBC-SRL	$O(n^2 \cdot p + n \cdot e)$
LBC-S, LBC-SDS, LBC-SAS, LBC-SWS, HEFT, MinMin, PEFT	$O(n^2 \cdot p)$
HCPT	$O(n \log n)$

Table 1. Asymptotic time complexities of the evaluated scheduling heuristics, expressed in number of tasks n, number of processors p, and number of dependencies e.

5 EXPERIMENTAL SETUP

5.1 System Configuration

All experiments were conducted in a Windows 11 environment using Python 3.10.12 for implementation and R 4.1.2 for statistical analysis and visualization. The baseline algorithms HEFT, HCPT and MinMin, used for the comparative evaluation, were adopted directly from the publicly available implementations provided by Canon et al. [9].

Each experimental configuration was executed 1000 times to ensure that the results are statistically robust and not influenced by randomness in the DAG generation process. Five different random DAG generators were used to create a diverse set of synthetic DAGs. To evaluate the scheduling heuristics under different levels of parallelism, each DAG configurations was tested using 2, 3, 5, 7, and 10 processors.

Further details on the evaluation dataset, experimental parameters, and DAG generation techniques are provided in the following section. All source code and additional resources are publicly available through the accompanying GitHub repository [11]. This repository includes all scripts needed to replicate the experiment, ensuring full reproducibility.

5.2 Evaluation Data

To evaluate the proposed algorithms, we use synthetically generated DAGs with $n = 100$ vertices. The processing costs are sampled uniformly over a range of 1 to 20. Although actual processing costs vary significantly by domain, this range provides a balanced set of task execution times, suitable for a general evaluation. The generation methods produce DAGs with varying structural properties, allowing us to assess the strengths and limitations of each heuristic across various DAG topologies. These generation techniques were implemented and analyzed in detail by Canon et al. [9]. The following methods will be used for evaluation:

Erdős–Rényi-Based Random Generation: This generation method is based on the Erdős–Rényi algorithm [12], which constructs DAGs by adding edges between pairs or vertices with independent probability p. To maintain acyclicity, edges are only added in the upper triangle of the adjacency matrix. The value of p controls the graph's sparsity or density. In our study, we use $p = 0.1$ and $p = 0.5$. The former produces sparse graphs with few edges between the nodes, while the latter produces dense graphs with many edges between the nodes. These p-values allow us to evaluate whether our algorithm performs better on sparse, dense, or both types of graphs.

Uniform Random DAG Generation: This generation method uniformly samples from the set of all labeled DAGs with n vertices using a recursive counting approach [23]. It guarantees an unbiased distribution within this well-defined class, ensuring every labeled DAG has the same probability of being selected. This enables an evaluation of the scheduling algorithms without any bias towards particular DAG shapes or structural patterns. However, the total number of dense DAGs significantly exceeds that of sparse ones in the sample space. This is due to the increasing number of possible edge combinations as graphs become denser. Consequently, dense graphs naturally appear more frequently in the generated dataset.

Random Orders Method: This method derives a DAG from randomly generated orders [28]. These orders are generated by intersecting k random total orders. The parameter k controls the depth and structure of the resulting DAGs. Higher values of k produce sparser and shallower DAGs, as edges are only added when multiple random total orders consistently agree on the direction. For our study, we select $k = 3$ to generate DAGs that reflect a balanced combination of task dependencies and parallelism. This choice produces graphs with moderate depth and sparsity, offering a realistic workload for scheduling algorithms.

Layer-by-Layer Method: The layer-by-layer method, first proposed by Adam et al. [1], is adapted in this study using a variant introduced by Canon et al. [9]. This method constructs DAGs by assigning vertices to layers and adding edges between layers with probability p. In this study, we use $\sqrt{n}$ layers and set $p = 0.5$. This layered structure naturally models workflows with sequential phases, which enables a realistic assessment of scheduling algorithms under layered dependency constraints.

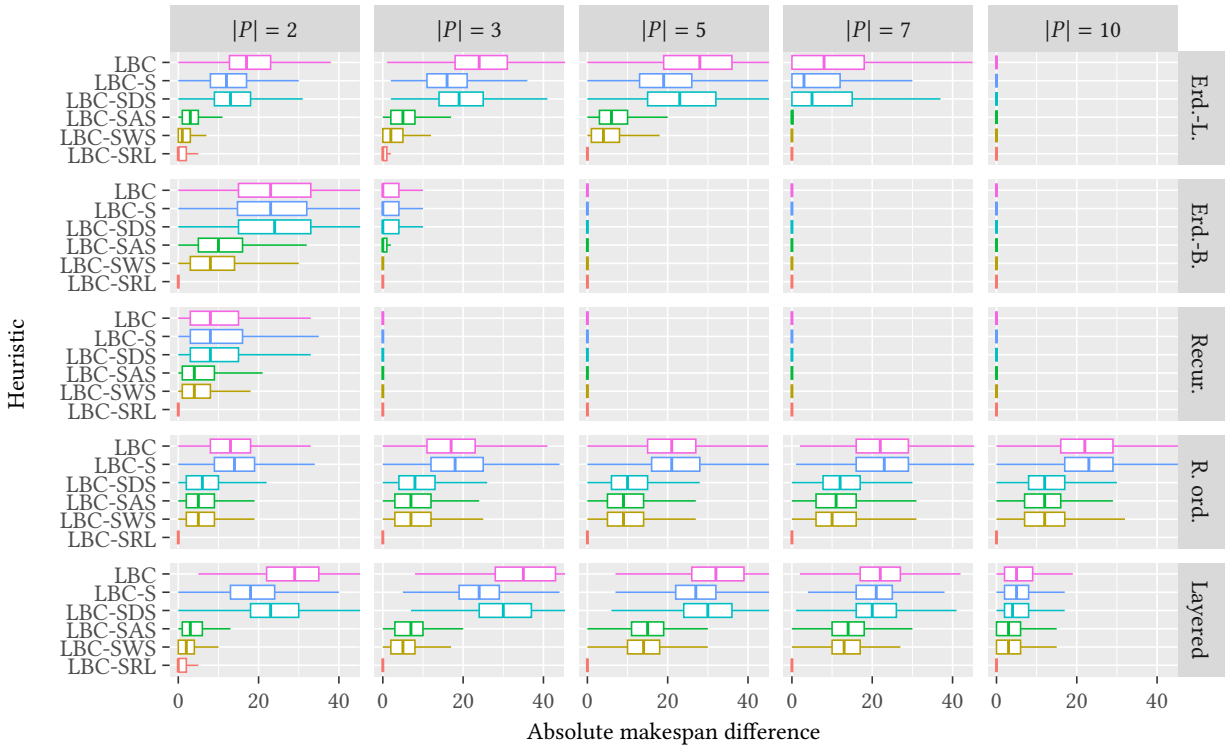

Fig. 1. Comparative performance of LBC-based task ranking strategies across various DAG topologies and processor configurations. The x-axis shows the absolute makespan difference. Lower values reflect better scheduling performance. 'Erd-L' and 'Erd-B' refer to Erdős–Rényi DAGs with edge probabilities $p = 0.1$ and $p = 0.5$, respectively. Other methods include recursive, random order, and layered DAG generators.

6 RESULTS

6.1 Comparative Evaluation of LBC-Based Ranking Strategies

To evaluate the effectiveness of the proposed LBC-based ranking strategies, the absolute makespan differences achieved by each approach across a diverse set of DAG topologies and processor configurations were measured. For each scheduling run, we compute the makespan C_{method} achieved by a given heuristic and subtract the minimal makespan C_{min} obtained across all heuristics for that specific configuration. Formally, the value plotted on the y-axis is:

$$\Delta C = C_{method} - C_{min} \tag{16}$$

Figure 1 summarizes these results. In this figure, algorithms positioned further to the left correspond to lower makespan values, indicating better scheduling performance. Among all ranking strategies, LBC-SRL consistently showed the best performance, demonstrating the effectiveness of recalculating task criticalities.

Furthermore, the figure shows that the graph density has a significant influence on the results. Sparser DAGs tend to result in a greater variability in makespan across the heuristics. These graphs have less restrictive precedence constraints, which offer greater flexibility in task ordering. In such cases, more informed heuristics can leverage the graph structure more effectively, resulting in lower makespan values. In contrast, as the processor count increases, denser DAGs fail to differentiate ranking strategies. Denser DAGs restrict the range of available scheduling permutations more, causing the results to converge more to the length of the *critical path*.

Among the remaining ranking strategies, LBC-SWS achieves the second-best results, closely followed by LBC-SAS. These findings support the hypothesis that successor tasks play a significant role in determining the criticality of a task. This highlights the value of incorporating successor information in the task prioritization process. In contrast, both LBC and LBC-S show consistently poor results across all DAG structures. This suggests that relying solely on a node's LBC score is insufficient for effective task prioritization.

Overall, LBC-SRL proved to be the most effective ranking strategy among those evaluated. Therefore, it is selected as the primary heuristic for the comparison against the established scheduling algorithms. However, because of its relatively lower computational complexity, LBC-SWS is also included in further analyses.

6.2 Evaluation Against Established Heuristics

Figure 2 presents the absolute difference in makespan between the established scheduling algorithms and the two proposed LBC-based heuristics. The figure clearly shows that HEFT consistently achieves the lowest makespan. This supports its reputation as one of the most effective list scheduling algorithms.

LBC-SRL achieves the second best results. It outperforms several well-known heuristics. For instance, it achieves consistently better results than both HCPT and MinMin. This outcome is expected for MinMin due to its simplistic design, which does not consider the broader graph structure or future dependencies. HCPT also has a major limitation. Despite considering the graph structure, it does not utilize insertion-based scheduling. This reduces HCPT's flexibility in utilizing available processor

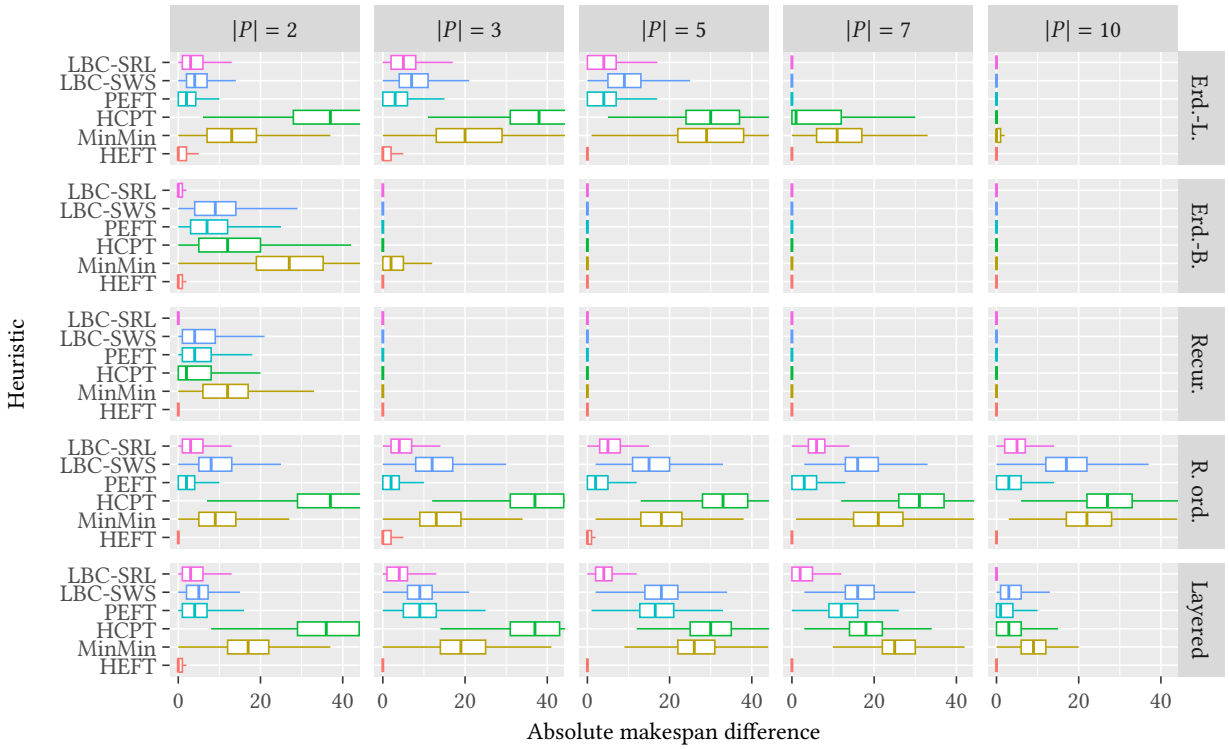

Fig. 2. Absolute makespan differences for established scheduling heuristics and the proposed LBC-based algorithms (LBC-SRL, LBC-SWS) across various DAG generation models and processor counts. The x-axis shows the absolute makespan difference. Lower values reflect better scheduling performance.

time slots, resulting in less efficient schedules. This drawback is especially noticeable when the number of processors is low. Efficient task placement becomes more critical when the number of available processors is limited. Consequently, as shown in the figure, HCPT's performance improves with higher processor counts, where its lack of insertion flexibility has less impact.

Furthermore, PEFT slightly underperforms compared to LBC-SRL. However, it should be noted that PEFT typically excels in environments with significant communication costs. Such costs are explicitly omitted in this study. This design decision likely reduces the advantage of PEFT's optimistic cost table, which is optimized for both computation and communication costs.

LBC-SRL especially excels in denser graphs (such as those generated using Erdős–Rényi-based method with $p = 0.5$ as well as DAGs generated using the recursive and layered methods). This is likely because LBC-SRL better captures global structural dependencies in complex DAGs, allowing it to prioritize tasks that are critical across long dependency chains. In these dense graphs, LBC-SRL achieves similar results to HEFT. HEFT's reliance on upward ranks may cause suboptimal task placements in graphs with multiple equally critical paths, a situation more common in dense graphs.

LBC-SWS outperformed both HCPT and MinMin. Like LBC-SRL, this is likely due to its ability to account for global task dependencies and better task placement. It even performs similarly to PEFT. Although LBC-SWS does not quite match the performance of LBC-SRL, its lower computational complexity makes it a strong option in scenarios where efficiency is a priority.

Figure 3 provides a pairwise comparison of the heuristics, indicating how frequently each algorithm outperforms another. For each pair of heuristics h_i and h_j, we calculate the percentage of test cases where h_i produces a lower makespan than h_j. Formally, the win percentage is defined as:

$$W(h_i, h_j) = \frac{\text{Number of cases where } C_{h_i} < C_{h_j}}{\text{Total number of comparisons}} \times 100 \quad (17)$$

where C_{h_i} and C_{h_j} are the makespan produced by heuristics h_i and h_j. Ties, representing cases where both heuristics yield the same makespan, can be inferred as the remaining percentage. Specifically, they can be computed with the following formula.

$$T(h_i, h_j) = 100\% - W(h_i, h_j) - W(h_j, h_i) \quad (18)$$

The pairwise results reveal that LBC-SRL beats PEFT in 33.2% of the cases, while being outperformed in only 21.7%. This clearly indicates a significant improvement. However, it only outperforms HEFT in 3.1% of cases and is outperformed in 42.0%, the remaining 54.9% being ties. This shows that, although LBC-SRL does not surpass HEFT, it remains competitive by tying more than half of the cases. Moreover, LBC-SRL is only outperformed in less than 2% of the cases by HCPT and MinMin. This demonstrates a significant improvement over these algorithms. Similarly, LBC-SWS performs well, outperforming both MinMin and HCPT more than 52% of the time.

7 CONCLUSION

This paper introduced novel heuristics for task scheduling based on the Longest Betweenness Centrality metric. We developed and evaluated six LBC-based ranking methods for task scheduling. Among these, LBC-SRL emerged as the most effective variant. Experimental results demonstrated that our novel approach

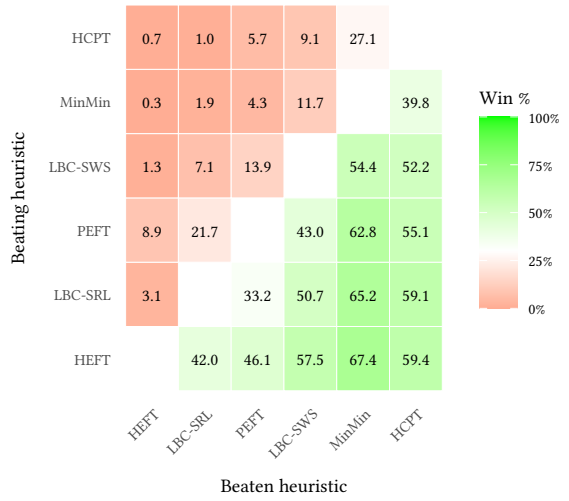

Fig. 3. Pairwise win percentage heatmap for all evaluated heuristics based on makespan performance. Each cell represents the proportion of runs in which the row algorithm outperforms the column algorithm.

yields competitive makespan performance compared to established heuristics, such as HEFT, MinMin, PEFT, and HCPT. It particularly excelled in DAGs with high density.

To address the first research question, we investigated the impact of various LBC-based strategies on scheduling performance. Experimental results showed that LBC-SRL consistently achieved the lowest makespan among the proposed heuristics. This indicates that recalculating LBC iteratively after each scheduling step is the most effective approach for estimating a task importance, as it enables us to iteratively identify tasks that lie on the most longest dependency paths.

Regarding computational efficiency, we observed that LBC-based heuristics have a time complexity of $O(n^2 \cdot p)$ or $O(n^2 \cdot p + n \cdot e)$. While LBC-SRL offered the highest performance, LBC-SWS traded some of this for reduced time complexity in dense graphs. This makes it a more suitable choice in environments where execution speed is of high priority. These complexities are comparable to those of established algorithms, such as HEFT and PEFT. As a result, LBC-based heuristics remains practical for large task graphs and represent competitive alternatives to existing heuristics without introducing significant computational overhead.

Finally, in comparison to established heuristics, LBC-SRL and LBC-SWS demonstrated competitive performance. LBC-SRL outperformed MinMin, HCPT, and PEFT across nearly all configurations. Results indicated that LBC-SRL especially excels on DAGs with high density, achieving results similar to HEFT. Although HEFT remained the top performer overall, pairwise comparison revealed that LBC-SRL matches HEFT's performance in 54.9% of cases, while even outperforming it 3.1% of the time. These findings highlight the effectiveness of LBC as a scheduling metric.

Overall, incorporating global task influence in task scheduling heuristics can significantly enhance performance. The strong results of LBC-SRL in dense graphs underscore the potential of centrality-based heuristics as viable alternatives to traditional methods, particularly in domains with many inter-task dependencies. Beyond immediate performance improvements, these findings indicate that incorporating advanced network analysis techniques into task scheduling represents a promising direction for the development of more topology-aware heuristics.

7.1 Limitations

Although the proposed LBC-based scheduling heuristics show promising results, several limitations should be acknowledged.

Assumption of Homogeneous Processors: All experiments in the evaluation were conducted assuming homogeneous processors with identical capabilities. Although this simplifies the scheduling model, it may not reflect the heterogeneous nature of some real-world domains. However, given that the scheduling phase of the proposed heuristics is based on the same insertion-based EFT scheduling principle as HEFT, adapting it for heterogeneous settings is feasible and would likely maintain competitive performance.

Assumption of Static Scheduling Environment: This study assumes complete knowledge of the task graph and uses static scheduling. However, in many real-world systems, scheduling must be performed dynamically due to partial information, runtime variability, or unexpected delays. The current approach does not address these dynamic constraints.

Exclusion of Inter-Task Communication Costs: Communication delays between tasks assigned to different processors are excluded in our model. Communication overhead can significantly impact scheduling performance in distributed environments.

Use of Synthetic Benchmark DAGs: The evaluation was performed exclusively on synthetically generated DAGs. These allow for controlled experimentation and structural variety, but may not fully capture the irregularities of real-world workflows.

7.2 Future Work

Considering the findings and limitations of this study, there are several meaningful directions to explore for future research.

Integration of Communication Costs: Incorporating inter-task communication delays would provide a more realistic evaluation of scheduling performance. Future adaptations of LBC-based heuristics may include adaptations that account for communication costs during the processor selection phase.

Extension to Heterogeneous Environments: Adapting the LBC framework to account for heterogeneous processors would increase its applicability to modern distributed computing platforms. This may involve modifying the processor selection strategy to account for processor-specific execution times per task. Moreover, the generation methods should be adapted to produce processing times that accurately reflect the characteristics of each processor.

Benchmarking on Real-World DAGs: Evaluating the heuristics on real-world DAGs would provide stronger validation of their practical effectiveness. As part of future work, real-world DAGs from the WfCommons WfInstances collection [10] can be leveraged, including Epigenomics [18], Genome [13], and Montage [24]. These datasets cover diverse domains, such as biomedical sequencing, genomic analysis, and astronomical imaging. Consequently, they provide a wide range of structural characteristics for evaluation.

Broader Algorithmic Benchmarking: While comparisons were made against several established heuristics, future work could expand this set to include more recent or domain-specific scheduling algorithms. This would provide a more comprehensive evaluation of where LBC excels and where it needs further improvement.

REFERENCES

[1] Thomas L. Adam, K. M. Chandy, and J. R. Dickson. 1974. A comparison of list schedules for parallel processing systems. *Commun. ACM* 17, 12 (1974), 685–690. https://doi.org/10.1145/361604.361619

[2] Wakar Ahmad, Gaurav Gautam, Bashir Alam, and Bhoopesh Singh Bhati. 2024. An Analytical Review and Performance Measures of State-of-Art Scheduling Algorithms in Heterogeneous Computing Environment. *Archives of Computational Methods in Engineering* 31, 5 (2024), 3091–3113. https://doi.org/10.1007/s11831-024-10069-8

[3] Hamid Arabnejad and Jorge G. Barbosa. 2014. List Scheduling Algorithm for Heterogeneous Systems by an Optimistic Cost Table. *IEEE Transactions on Parallel and Distributed Systems* 25, 3 (2014), 682–694. https://doi.org/10.1109/TPDS.2013.57

[4] AR. Arunarani, D. Manjula, and Vijayan Sugumaran. 2019. Task scheduling techniques in cloud computing: A literature survey. *Future Generation Computer Systems* 91 (2019), 407–415. https://doi.org/10.1016/j.future.2018.09.014

[5] A.I. Awad, N.A. El-Hefnawy, and H.M. Abdel-kader. 2015. Enhanced Particle Swarm Optimization for Task Scheduling in Cloud Computing Environments. *Procedia Computer Science* 65 (2015), 920–929. https://doi.org/10.1016/j.procs.2015.09.064

[6] David A. Bader, Shiva Kintali, Kamesh Madduri, and Milena Mihail. 2007. Approximating Betweenness Centrality. In *Algorithms and Models for the Web-Graph*, Anthony Bonato and Fan R. K. Chung (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 124–137.

[7] D. Bertsekas and J. Tsitsiklis. 2015. *Parallel and Distributed Computation: Numerical Methods*. Athena Scientific, Athena. https://books.google.nl/books?id=n_Q5EAAAQBAJ

[8] Ulrik Brandes. 2001. A faster algorithm for betweenness centrality*. *The Journal of Mathematical Sociology* 25, 2 (2001), 163–177. https://doi.org/10.1080/0022250X.2001.9990249

[9] Louis-Claude Canon, Mohamad El Sayah, and Pierre-Cyrille Héam. 2019. A Comparison of Random Task Graph Generation Methods for Scheduling Problems. In *Euro-Par 2019: Parallel Processing*, Ramin Yahyapour (Ed.). Springer International Publishing, Cham, 61–73.

[10] 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 (2022), 16–27. https://doi.org/10.1016/j.future.2021.09.043

[11] Niek Damink. 2025. LBC Task Scheduling Heuristic. https://github.com/Niek-Damink/LBC_TaskSched

[12] P. Erdős and A. Rényi. 1959. On random graphs I. *Publ. math. debrecen* 6, 290-297 (1959), 18.

[13] 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 (2019), 615–628. https://doi.org/10.1016/j.future.2019.01.015

[14] Kannan Govindarajan, Supun Kamburugamuve, Pulasthi Wickramasinghe, Vibhatha Abeykoon, and Geoffrey Fox. 2017. Task Scheduling in Big Data - Review, Research Challenges, and Prospects. In *2017 Ninth International Conference on Advanced Computing (ICoAC)*. IEEE, Chennai, India, 165–173. https://doi.org/10.1109/ICoAC.2017.8441494

[15] R.L. Graham, E.L. Lawler, J.K. Lenstra, and A.H.G. Rinnooy Kan. 1979. Optimization and Approximation in Deterministic Sequencing and Scheduling: a Survey. In *Discrete Optimization II*, P.L. Hammer, E.L. Johnson, and B.H. Korte (Eds.). Annals of Discrete Mathematics, Vol. 5. Elsevier, Cham, 287–326. https://doi.org/10.1016/S0167-5060(08)70356-X

[16] Oscar H. Ibarra and Chul E. Kim. 1977. Heuristic Algorithms for Scheduling Independent Tasks on Nonidentical Processors. *J. ACM* 24, 2 (1977), 280–289. https://doi.org/10.1145/322003.322011

[17] Jan Janecek and Tarek Hagras. 2003. A Simple Scheduling Heuristic for Heterogeneous Computing Environments . In *Parallel and Distributed Computing, International Symposium on*. IEEE Computer Society, Los Alamitos, CA, USA, 104. https://doi.org/10.1109/ISPDC.2003.1267650

[18] 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 (2013), 682–692. https://doi.org/10.1016/j.future.2012.08.015

[19] Han Lin, Ming-Fan Li, Cheng-Fan Jia, Liu. Jun-Nan, and Hong An. 2019. Degree-of-Node Task Scheduling of Fine-Grained Parallel Programs on Heterogeneous Systems. *Journal of Computer Science and Technology* 34, 5 (2019), 1096–1108. https://doi.org/10.1007/s11390-019-1962-4

[20] C. L. Liu and James W. Layland. 1973. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *J. ACM* 20, 1 (1973), 46–61. https://doi.org/10.1145/321738.321743

[21] Fatma A. Omara and Mona M. Arafa. 2010. Genetic algorithms for task scheduling problem. *J. Parallel and Distrib. Comput.* 70, 1 (2010), 13–22. https://doi.org/10.1016/j.jpdc.2009.09.009

[22] L. Pinedo, Michael. 2016. *Scheduling: Theory, Algorithms, and Systems* (5 ed.). Springer, Cham. https://doi.org/10.1007/978-3-319-26580-3

[23] Robert W Robinson. 1973. Counting unlabeled acyclic digraphs. *New directions in the theory of graphs* (1973), 239–273.

[24] M. Rynge, G. Juve, J. Kinney, J. Good, B. G. B., A. Merrihew, and E. Deelman. 2013. Producing an Infrared Multiwavelength Galactic Plane Atlas using Montage, Pegasus and Amazon Web Services. In *Proceedings of the 23rd Annual Astronomical Data Analysis Software and Systems (ADASS) Conference*.

[25] Chathurangi Shyalika, Thushari Silva, and Asoka Karunananda. 2020. Reinforcement learning in dynamic task scheduling: A review. *SN Computer Science* 1, 6 (2020), 306.

[26] 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 (2002), 260–274. https://doi.org/10.1109/71.993206

[27] J.D. Ullman. 1975. NP-complete scheduling problems. *J. Comput. System Sci.* 10, 3 (1975), 384–393. https://doi.org/10.1016/S0022-0000(75)80008-0

[28] Peter Winkler. 1985. Random orders. *Order* 1 (1985), 317–331.

APPENDICES

A AI STATEMENT

During the preparation of this work the author used ChatGPT in order to enhance the writing of this paper. After using this tool, the author reviewed and edited the content as needed and takes full responsibility for the content of the work.

B PSEUDOCODE

This appendix presents the detailed pseudocode for the algorithms described in the main text. The pseudocode is designed to clearly explain the algorithmic steps and key computations without relying on any specific programming language.

B.1 LBC Pseudocode

This appendix presents the pseudocode for computing LBC scores for all nodes in a DAG.

Algorithm 2 *LBC* Pseudocode.

function *LBC*(Graph G)
  LBC$[v] \leftarrow 0, \quad v \in G.V$;
  $topological\_order \leftarrow$ Topological-Sort(G);
  while $topological\_order \neq \emptyset$ **do**
    $s \leftarrow topological\_order[0]$
    $visited \leftarrow [s]$
    $P[w] \leftarrow [], \quad w \in G.V$;
    $\sigma[t] \leftarrow 0, \quad t \in G.V; \quad \sigma[s] \leftarrow 1$
    $d[t] \leftarrow -1, \quad t \in G.V; \quad d[s] \leftarrow 0$
    for all $v \in topological\_order$ **do**
      if $v \neq s$ and $\sigma[v] = 0$ **then**
        skip to next iteration
      end if
      append $v \rightarrow visited$
      for all outgoing neighbors w of v **do**
        if $d[w] < d[v] + c_v$ **then**
          $d(w) \leftarrow d(v) + c_v$
          $\sigma[w] \leftarrow \sigma[v]$
          $P[w] \leftarrow [v]$
        else if $d[w] = d[v] + c_v$ **then**
          $\sigma[w] \leftarrow \sigma[w] + \sigma[v]$
          append $v \rightarrow P[w]$
        end if
      end for
    end for
    $\delta[t] \leftarrow 0, \quad t \in visited$
    for all $w \in$ reversed$(visited)$ **do**
      for all $v \in P(w)$ **do**
        $\delta[v] \leftarrow \delta[v] + \frac{\sigma[v]}{\sigma[w]} \cdot (1 + \delta[w])$
      end for
      if $w \neq s$ **then**
        LBC$[w] \leftarrow$ LBC$[w] + \delta[w]$
      end if
    end for
    pop $s \rightarrow topological\_order$
  end while
end function

B.2 Source-based LBC Pseudocode

This appendix details the pseudocode for the source-based LBC variant. The pseudocode is identical to the general LBC algorithm with one key addition: at the start of each iteration over the topological order, the algorithm checks if the current node has incoming neighbors. If it does, the node is skipped to ensure only source nodes initiate the traversal. Consequently, the following line needs the be added to the start of the main loop:

Algorithm 3 Early-exit condition to restrict traversal to source nodes only.

if incoming neighbors of $v \neq \emptyset$ **then**
  return *LBC*
end if

B.3 LBC-SRL Pseudocode:

This appendix presents the pseudocode for the ranking phase of the LBC-SRL algorithm, which is one of the fundamental stages in the LBC-SRL heuristic.

Algorithm 4 Task Ranking using *LBC* Scores.

function LBC-SRL-Ranking(Graph G)
  $ranking \leftarrow []$
  while $G.V \neq \emptyset$ **do**
    $G \leftarrow$ Add-Super-Node(G)
    $scores \leftarrow LBC_s(G)$
    $G \leftarrow$ Remove-Super-Node(G)
    $ready\_tasks \leftarrow \{v \in G.V \mid in\_degree(v) = 0\}$
    $task \leftarrow \max_{v \in ready\_tasks} (scores[v])$
    append $task \rightarrow ranking$
    Remove-Node$(G, task)$
  end while
  return *ranking*
end function

B.4 Score-Guided Topological Sorting Pseudocode

This appendix presents the pseudocode for the score-guided topological sorting, which is used in most of the proposed heuristics to generate a valid task ranking based on precomputed scores, while respecting all dependency constraints.

Algorithm 5 Score-Guided Task Ranking Using *LBC* Scores.

function Score-Guided-Ranking(Graph G, Scores S)
  $ranking \leftarrow [\,]$
  $ready\_tasks \leftarrow \{v \in G.V \mid in\_degree(v) = 0\}$
  $heap \leftarrow$ construct heap for $[S[v], v] \in ready\_tasks$
  while $heap \neq \emptyset$ **do**
    $v \leftarrow$ highest-priority node from *heap*
    append v to *ranking*
    for all successors w of v **do**
      remove edge (v, w) from G
      if w now has no incoming edges **then**
        insert w into *heap* with priority $S[w]$
      end if
    end for
  end while
  return *ranking*
end function

B.5 Insertion-based EFT Scheduling Pseudocode

This appendix presents the pseudocode for the scheduling phase of the heuristics, which assigns tasks to processors using an insertion-based EFT strategy.

Algorithm 6 Insertion-Based EFT Scheduling.

function SCHEDULE(Graph G, Ranking R, Processors P, Costs C)
  $task\_finish[t] \leftarrow 0, t \in G.V$
  $schedule \leftarrow [\,]$
  $proc\_slots[p] \leftarrow [\,], p \in P$
  for all $t \in R$ **do**
    $ready\_at \leftarrow \max\{task\_finish[d] \mid d \in \text{predecessors}(t)\}$
    $EFT \leftarrow \infty$
    for all $p \in P$ **do**
      $start\_time \leftarrow$ find first gap after $ready_at$ on p
      $finish\_time \leftarrow start\_time + C[t]$
      if $finish\_time < EFT$ **then**
        $best\_proc \leftarrow p$
        $EST \leftarrow start\_time$
        $EFT \leftarrow finish\_time$
      end if
    end for
    Reserve slot $[start, end]$ on processor $best\_proc$
    $task\_finish[t] \leftarrow EFT$
    Append $\{task : t,\ proc : best\_proc,\ start : EST,\ end : EFT\} \rightarrow schedule$
  end for
  return $schedule$
end function

C ILLUSTRATIVE EXAMPLE OF LONGEST BETWEENNESS CENTRALITY

This Appendix provides a concrete example demonstrating how the Longest Betweenness Centrality (LBC) metric is computed on a small weighted DAG. This example is intended for readers who need further clarification beyond the formal definition provided in the main text or for those who wish to develop a more intuitive understanding of the metric through a step-by-step application.

LBC Example:
The DAG, shown in Figure 4, contains six nodes and seven directed edges, each annotated with a weight that represents the execution time of the task.

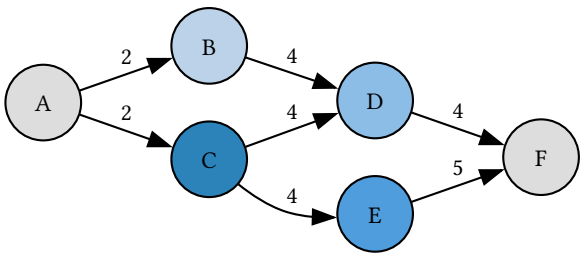

Fig. 4. Example DAG with node colors representing LBC scores. Darker nodes indicate higher centrality, reflecting more frequent occurrence on longest weighted paths.

1: Longest Paths from Source A to Sink F
The LBC metric quantifies how often a node lies on the longest weighted paths between pairs of nodes in the DAG. Consequently, we must consider all longest paths between all source-destination pairs in the DAG. We will first consider the contribution to the LBC from the longest path between A and F

(1) A → B → D → F with total weight: 2 + 4 + 4 = 10
(2) A → C → D → F with total weight: 2 + 4 + 4 = 10
(3) A → C → E → F with total weight: 2 + 4 + 5 = 11

The longest path among these is the third, so only nodes C and E contribute to LBC for this path.

2: Additional Longest Paths From Node A To B, C, D and E
We now consider all longest paths originating from node A, assigning points to all nodes that lie along these paths. When multiple longest paths exist, the points are evenly distributed across all such paths.

- To D:
  – $A \rightarrow B \rightarrow D$ and $A \rightarrow C \rightarrow D$. Both are equal-length longest paths, so B and C each receive 0.5 points.
- To E:
  – $A \rightarrow C \rightarrow E$. Node C gains 1 point.
- To B or C: Paths to B and C are direct from A, so no intermediate nodes contribute to their LBC.

Thus, only considering all longest paths from source node A, the intermediate LBC scores are described in the following table:

Node	LBC Score
A	0
B	0.5
C	2.5
D	0
E	1
F	0

3: Computing the Final LBC Scores
To compute the final LBC scores, the same process is repeated for all valid source–target pairs in the DAG. The cumulative scores obtained are shown in the following table:

Node	LBC Score
A	0
B	0.5
C	2.5
D	1
E	2
F	0

This example illustrates how LBC identifies structurally significant nodes that frequently appear along critical long execution paths in the DAG.