

Global Predecessor Indexing: Avoiding Binary Search in Weighted Job Scheduling

Amit Joshi

amitjoshi2724@gmail.com

amit.joshiusa@gmail.com

Independent Researcher

Abstract

We present an improved solution to the Weighted Job Scheduling (WJS) problem. While the classical dynamic programming (DP) solution runs in $O(n \log(n))$ time due to comparison-based sorting and per-job binary search [1], we eliminate the binary search bottleneck. In its place, we introduce a novel multi-phase preprocessing technique called *Global Predecessor Indexing (GPI)*, which computes the latest non-overlapping job (i.e., the predecessor) for all jobs via a two-pointer linear-time pass. GPI enables direct use in the classical DP recurrence. When combined with linear-time sorting, GPI yields a complete $O(n)$ solution. Even with comparison-based sorting, GPI significantly outperforms the classical solution in practice by avoiding repeated binary searches.

Keywords: Weighted Job Scheduling, Interval Scheduling, Dynamic Programming, Linear Sorting, Two Pointers, Preprocessing

1. Introduction

The Weighted Job Scheduling (WJS) or Weighted Interval Scheduling (WIS) problem is a classic combinatorial optimization task where each job has a time interval and weight (or value), and the goal is to select a subset of non-overlapping jobs that maximizes total weight. This problem arises in compiler register allocation, where weighted intervals represent variable lifetimes; the goal is to assign variables to registers without conflicts and prioritize the most critical variables. It also appears in server backup scheduling, where jobs have time windows and priority weights; the objective is to select non-overlapping jobs that maximize the total priority-weighted data secured.

The classical solution runs in $O(n \log(n))$ time for n jobs. It first sorts jobs by end time. Then, for each job i , it performs a $O(\log(n))$ binary search to find the latest job that ends before job i starts, known as its predecessor $p(i)$ [1] and accesses this index during the DP recurrence. The per-job binary search is a key bottleneck in the classical solution.

Our method avoids per-job binary search by precomputing all predecessor indices in a two-pointer $O(n)$ pass after sorting twice. We call our approach *Global Predecessor Indexing (GPI)*. GPI achieves theoretical $O(n)$ performance when paired with a linear-time sort, and yields strong practical speedups even with general comparison-based sorting.

The final DP step then runs in $O(n)$ time using these predecessor indices. We also discuss the parallelizability of our approach. To our knowledge, this is the first $O(n)$ algorithm for weighted job scheduling when job times admit linear-time sorting, such as bounded integers or smooth distributions.

2. Background

2.1. Classical DP Framework

Each job i has a start time s_i , end time e_i , and weight w_i . Two jobs i and k are compatible if their intervals do not overlap, i.e., if $e_i \leq s_k$ or $e_k \leq s_i$. All jobs in the chosen subset must be

pairwise compatible, and our aim is to find the optimal subset to maximize the sum of its intervals' weights.

The classical solution sorts jobs by increasing end time and computes an array dp , where $dp[i]$ stores the maximum achievable total weight using at most the first i jobs [1]. For each job i , it finds the latest job k such that $e_k \leq s_i$, defining $p(i) = k$. This is done with an $O(\log(n))$ binary search over the sorted end times. Over all n jobs, this yields $O(n \log(n))$ time complexity. Lastly, we have the DP recurrence:

$$dp[i] = \max(dp[i-1], w_i + dp[p(i)])$$

where $p(i)$ is the predecessor index of job i . If no predecessor exists for a job i (i.e., if no job ends at or before s_i), we define $p(i) = 0$. The base case is $dp[0] = 0$, corresponding to an empty job set. $dp[n]$ stores the final answer. Once dp is fully computed, the actual set of jobs comprising the optimal solution can be recovered in $O(n)$ time via standard backtracking. Starting from $i = n$, we compare $dp[i]$ to $dp[i-1]$: if they are equal, job i was excluded; otherwise, job i was included, and we continue the trace from its predecessor $p(i)$. The selected jobs can be stored in reverse and then returned in sorted order.

Algorithm 1 Classical DP Weighted Job Scheduling

Require: Array $jobs$ of n jobs

- 1: Initialize $end_ordered \leftarrow jobs$ sorted by increasing e_i
 - 2: **function** FINDPREDECESSORBINARYSEARCH(i)
 - 3: Binary search for largest $k < i$ such that $e_k \leq s_i$
 - 4: **return** k or 0 if no such k exists
 - 5: **end function**
 - 6: Initialize $dp \leftarrow [0, 0, \dots, 0]$
 - 7: **for** $i = 1$ to n **do**
 - 8: $k \leftarrow$ FINDPREDECESSORBINARYSEARCH(i)
 - 9: $dp[i] \leftarrow \max(dp[i-1], w_i + dp[k])$
 - 10: **end for**
 - 11: **return** $dp[n]$
-

2.2. Linear Sorting Algorithms

Standard comparison-based sorting algorithms run in $O(n \log(n))$ time. However, in many practical cases, job times can be sorted in linear time such as when they are bounded-width integers or floats, drawn from uniform distributions, or drawn from a small domain. In such scenarios, specialized linear-time sorting algorithms can achieve $O(n)$ time in the worst case or expectation. We briefly describe several below.

Counting Sort [2] runs in $O(n + K)$ time, where K is the largest input value. It counts occurrences of each value and reconstructs the sorted array stably. It works best when $K = O(n)$, and is inefficient when the input domain is large or sparse.

Radix Sort [2] achieves $O(d(n + b))$ time, where d is the number of digits and b is the base. It uses Counting Sort as a stable subroutine per digit. Radix Sort behaves linearly when d is small and can also handle fixed-width decimals.

Bucket Sort [3] divides input into n evenly spaced buckets, sorts each bucket (using a comparison-based sort), and concatenates the results. It has expected $O(n)$ time for roughly uniform inputs, but degrades to $O(n \log(n))$ or worse under skew. It is a good choice when $K \gg n$ or the input range is sparse.

Spreadsor [4] is a hybrid sorting algorithm that combines the linear-time performance of radix sort with the generality of comparison-based sorts. It achieves $O(n)$ in practice by adapting to non-uniform input distributions and degrades to $O(n \log(n))$ only in adversarial cases. It operates on the bit-level representations of numbers, recursively partitioning them by their most significant differing bits. The number of bits used per pass is chosen dynamically to balance recursion depth and bucket size. When buckets become small or indistinct, the algorithm falls back to comparison sort.

3. Our Approach

We present a simple, practical, and efficient algorithm for the weighted job scheduling problem. Our approach eliminates the bottleneck of per-job binary search by introducing a novel preprocessing technique, which we call *Global Predecessor Indexing (GPI)*. After sorting the job intervals twice, GPI computes all predecessor indices in a two-pointer linear pass.

When job times admit linear-time sorting (e.g., bounded-width, uniformly distributed), our full pipeline runs in $O(n)$ time. Even with general comparison-based sorting, GPI significantly outperforms the classical $O(n \log(n))$ solution in practice since per-job binary search is slower than sorting.

In the first phase, we sort the jobs by increasing *end* time and expand each job state to include its ordering within this sort. In the second phase, we sort by increasing *start* time on the array with the expanded jobs. We use both orderings to find all predecessor indices with a clever two-pointer $O(n)$ algorithm in the third phase.

Finally, we use these precomputed predecessor indices with the standard DP from Section 2 to compute the final numeric answer: the maximum achievable weight from a subset of non-overlapping job intervals. We can also easily find the subset of jobs themselves by backtracking. This eliminates the $O(\log(n))$

per-job binary search overhead with the classical DP, while using the same recurrence.

3.1. Sorting Jobs by End Time, Reordering, and Job State Expansion

We first sort jobs by increasing end time, and ties may be broken arbitrarily. Then, we reorder the given jobs by this sorted order. In other words, j_1 is the first job in this order, j_2 is next, and so on. We refer to index i as the “end-order” of j_i . By design, $\forall i < n : e_i \leq e_{i+1}$. As we are reordering, the initial ordering may be discarded.

Lastly, we append each job state j_i with its end-order.

$$j_i \leftarrow (s_i, e_i, w_i, i)$$

And we now refer to the array $[j_1, \dots, j_n]$ as the *end_ordered* array. For clarity, we also use “job i ” to refer to j_i , the job with end-order i . Additionally, $p(i)$ refers to the end-order of the predecessor job of j_i .

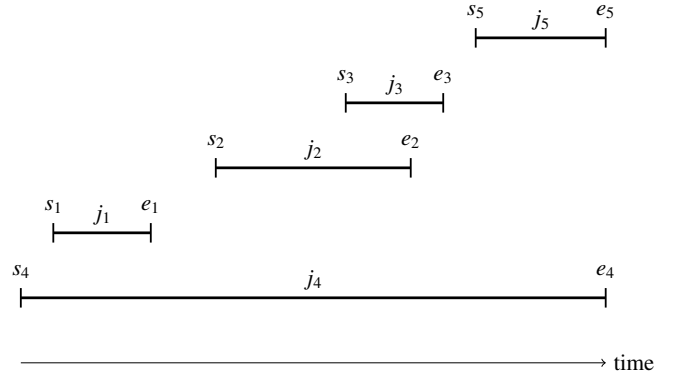


Figure 1: Example job intervals reordered by end time.

3.2. Sorting Jobs by Start Time

Unlike classical DP, we sort once again. This time we sort by increasing start time into a new array since we will need both orderings for the next phase. Similarly, we refer to a job’s index in the start time ordering as its “start-order.” By sorting again, we implicitly define a permutation π over $\{1, \dots, n\}$ which maps from a job’s start-order to its end-order. By design, $\forall i < n : s_{\pi(i)} \leq s_{\pi(i+1)}$. We refer to $[j_{\pi(1)}, \dots, j_{\pi(n)}]$ as the *start_ordered* array. By construction, $\pi(i)$ is the fourth component of the i^{th} job in the *start_ordered* array — due to the job state expansion from Section 3.1) For clarity, we also use “job $\pi(i)$ ” to refer to $j_{\pi(i)}$, the job with start-order i .

start-order i	end-order $\pi(i)$
1	4
2	1
3	2
4	3
5	5

Table 1: Permutation $\pi(i)$ mapping start-order to end-order of the jobs from Figure 1, created by sorting again by increasing start time.

3.3. Precomputing All Predecessor Indices in $O(n)$ time

Next, we describe our two-pointer $O(n)$ pass (Algorithm 2) to precompute all predecessors, avoiding the per-job binary search from the classical solution. We store predecessor indices in array p , where $p[i]$ holds the predecessor $p(i)$ for job i .

We find the predecessor for each job in decreasing start-order, i.e., find the predecessor for job $\pi(n)$, then job $\pi(n-1)$, down to job $\pi(1)$. For this, we use `start_index` to traverse backward through `start_ordered` in our outer loop. We use `end_index` to traverse backward through `end_ordered` in our inner loop. Before the outer loop begins, we use a single binary search on `end_ordered` to initialize `end_index` with $p(j_{\pi(n)})$: the predecessor index of job $\pi(n)$. This is the highest end-order of any predecessor job since no job starts later than job $\pi(n)$.

Within the outer loop, the inner loop decrements `end_index` until we find a job that ends no later than job $\pi(\text{start_index})$ starts. Since we traverse *backwards* through `end_ordered`, the first such job is guaranteed to be $p(j_{\pi(\text{start_index})})$: the highest end-order job which ends before job $\pi(\text{start_index})$ starts. If `end_index` hits 0, then any remaining jobs with unassigned predecessors remain as such since they start before any other job ends. We ensure their entries in p are 0. Once the outer loop finishes, all predecessors have been correctly assigned.

Job i (end-order i)	Predecessor index $p(i)$
1	0
2	1
3	1
4	0
5	3

Table 2: Predecessor indices $p(i)$ corresponding to the intervals in Figure 1.

Time Complexity

We analyze the time complexity of Algorithm 2: Precompute Predecessors. First, we perform a single binary search to find $p(\pi(n))$ which takes $O(\log(n))$ time. Next, although the while loop is nested within the for loop, the crucial observations are that:

- `start_index` decrements monotonically and visits each index from $[1, \dots, n]$ at most once, and so the number of iterations of the outer loop is at most n .
- `end_index` is initialized before the outer loop, decrements monotonically, and visits each index from $[0, \dots, n]$ at most once. The *total* number of iterations of the inner loop across the function is at most n .

Each iteration of the inner loop performs a fixed number of $O(1)$ operations: evaluating `ENDSNOATERTHANSTARTS` (two array accesses and a comparison) and decrementing `end_index`. Each iteration of the outer loop also performs a fixed number of $O(1)$ operations: evaluating the comparison on line 16, the array assignment on line 19, and decrementing `start_index`. All of these operations—including arithmetic updates, logical

Algorithm 2 Precompute Predecessors

Require: Array `end_ordered` of n jobs. Appears as $[j_1, \dots, j_n]$ where each (s_i, e_i, w_i, i) sorted by increasing e_i .

Require: Array `start_ordered` of n jobs. Appears as $[j_{\pi(1)}, \dots, j_{\pi(n)}]$ where each $(s_{\pi(i)}, e_{\pi(i)}, w_{\pi(i)}, \pi(i))$ sorted by increasing $s_{\pi(i)}$

```

1: function ENDSNOATERTHANSTARTS( $k, i$ )
2:   Check if  $j_k$  ends no later than  $j_i$  starts
3:   return true if  $e_k \leq s_i$ , false otherwise
4: end function
5: function FINDPREDECESSOR( $i$ )
6:   Binary search over end_ordered for largest  $k < i$  such
   that  $e_k \leq s_i$ 
7:   return  $k$  or 0 if no such  $k$  exists
8: end function
9: function COMPUTEPREDECESSORS( $n$ )
10:  Initialize  $p[1 \dots n] \leftarrow [0 \dots 0]$   $\triangleright$  predecessor array
11:  end_index  $\leftarrow$  FINDPREDECESSOR( $\pi(n)$ )
12:  for start_index =  $n$  to 1 do
13:    while end_index  $\geq 1$  and
      (not ENDSNOATERTHANSTARTS(end_index,
       $\pi(\text{start\_index})$ )) do
14:      end_index  $\leftarrow$  end_index - 1
15:    end while
16:    if end_index = 0 then  $\triangleright$  Remaining jobs
      with start-order  $\leq \text{start\_index}$  do not have predecessors,
      so their entries in  $p$  remain 0
17:      break
18:    end if
19:     $p[\pi(\text{start\_index})] \leftarrow \text{end\_index}$ 
20:  end for
21:  return  $p$ 
22: end function

```

comparisons, and array accesses or assignments—are primitive and take constant time, i.e., $O(1)$. Since the total number of `start_index` and `end_index` decrements is bounded by $2n$, and each decrement involves only $O(1)$ work, the total time complexity is $O(n)$.

Correctness

We prove correctness of our novel preprocessing step via a Loop Invariant over the outer (**for**) loop on line 12.

Loop Invariant. At the beginning of line 16 during iteration `start_index` = i , `end_index` is the predecessor of job $\pi(\text{start_index}) = i$. Although the loop begins at line 12, this invariant is defined to hold specifically at line 16, just before the predecessor for job $\pi(\text{start_index})$ is assigned on line 19.

Base Case. Before the first iteration, FINDPREDECESSOR uses binary search to initialize `end_index` with $p(j_{\pi(n)})$. Since `start_index` = n in the first iteration and by the correctness of the classical solution, the loop invariant holds true. The inner loop will not decrement `end_index` during the first iteration.

Maintenance. Assume the invariant holds at the start of line 16 during iteration $\text{start_index} = i$. Hence, end_index equals $p(\pi(\text{start_index}))$. Now, we show that the invariant also holds true in the next iteration where $\text{start_index} = (i - 1)$. First, we prove that $p(\pi(i - 1)) \leq p(\pi(i))$:

Proof. By way of contradiction, assume the opposite, so $p(\pi(i - 1)) > p(\pi(i))$. By definition of predecessor, job $p(\pi(i - 1))$ ends no later than job $\pi(i - 1)$ starts. By construction, job $\pi(i - 1)$ starts no later than job $\pi(i)$ starts. Then, by transitivity, job $p(\pi(i - 1))$ ends no later than job $\pi(i)$ starts.

Due to our assumption for contradiction that $p(\pi(i - 1)) > p(\pi(i))$, $p(\pi(i))$ can not be the predecessor for job $\pi(i)$ since there is a higher end-order job, job $p(\pi(i - 1))$, that ends no later than job $\pi(i)$ starts. However, $p(\pi(i))$ is indeed defined to be the predecessor for job $\pi(i)$. \square

Now, we know that the correct end_index for iteration $\text{start_index} = (i - 1)$ occurs at or before the correct end_index from iteration $\text{start_index} = i$. As we are simply doing a linear search backwards through end_ordered , we are guaranteed to find the job with the highest end-order that ends no later than job $\pi(i - 1)$ starts: its predecessor. Lines 13-15 decrement end_index until the invariant is restored or maintained, so that when control reaches line 16, the invariant still holds.

Termination. When the loop terminates after iteration $\text{start_index} = 1$, the invariant has held at line 16 in every iteration $\text{start_index} = i$ that was reached and correctly assigning $p[\pi(i)]$ to end_index on line 19. Any outer-loop iterations that were not reached were because end_index hit 0, which all entries in p were initialized to anyway. Since π is a permutation over $\{1, \dots, n\}$ and start_index looped from n to 1, then $p[i]$ was set at most once for each $i \in \{1, \dots, n\}$ with the correct end_index . Thus, $p[i]$ correctly stores the predecessor for all jobs i .

Parallelizability

The outer loop can be split across multiple threads. Each thread can handle a continuous segment of start_index values in reverse order, say $[u_m, \dots, u_1]$. The cost is one binary search per thread to initialize end_index with the predecessor of job u_m . As each thread can compute the predecessor of many jobs, this is a good tradeoff.

3.4. Dynamic Programming

Assuming the predecessor indices $p(i)$ have already been computed (see Section 3.3) and we have GPI's resulting predecessor lookup table p , we now describe the final DP step from Section 2:

$$\text{dp}[i] = \max(\text{dp}[i-1], w_i + \text{dp}[p[i]])$$

with base case $\text{dp}[0] = 0$ by convention. During the DP, we can retrieve each job's predecessor in constant time via p , eliminating the need for per-job binary search and providing the main speedup of this paper. Since all array accesses and maximum computations are $O(1)$, the total time to compute the DP table,

the final numeric answer, and the optimal subset of jobs themselves is $O(n)$.

Correctness

As in the classical solution, $\text{dp}[i]$ holds the maximum achievable total weight among the first i jobs. We prove correctness by induction on i . The base case $\text{dp}[0] = 0$ is valid as no jobs have been selected. Assume $\text{dp}[i-1]$ is correct. Then $\text{dp}[i]$ either inherits the optimal solution for the first $(i - 1)$ jobs or takes the optimal solution for the predecessor $p(i)$ — $\text{dp}[p[i]]$ —plus w_i . Since $p(i)$ was correctly computed, and since the recurrence takes the maximum of these two cases, the solution remains optimal. Since $\text{dp}[n]$ now represents the maximum achievable total weight among the first n jobs, and there are n jobs total, it holds the final answer after the loop ends.

Parallelizability

Since all predecessor relationships are known in advance, the DP step can be parallelized using Kahn's topological sort [5], where each job depends only on its predecessor job's DP value. The effectiveness of this parallelization depends on how often jobs are predecessors to multiple other jobs.

4. Experimental Results

We benchmark GPI with linear sorting and GPI with Timsort against the classical $O(n \log(n))$ solution. Although GPI achieves theoretical $O(n)$ with linear sorting, GPI with Timsort offers the best practical performance due to Timsort's optimized design. We release our implementation and benchmark code.¹

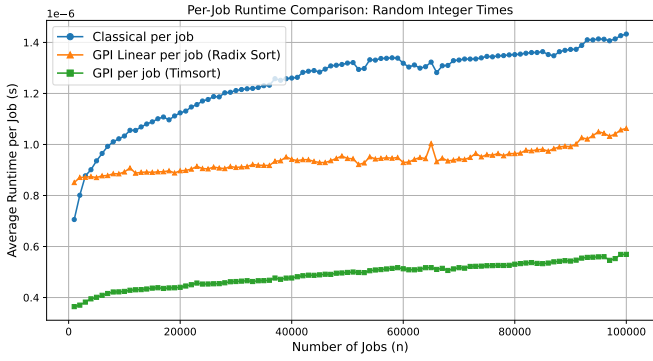
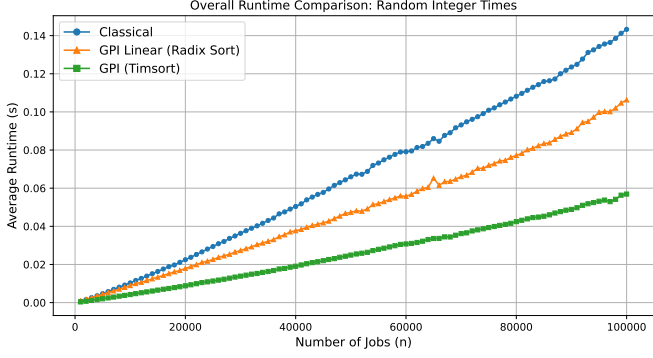
4.1. Benchmark Design

We design four controlled experiments to evaluate GPI performance under different job generation patterns, each capturing distinct structural characteristics of real-world workloads. Both classical and linear-time algorithms are implemented in Python 3 and benchmarked on identical hardware with cross-validated results. Runtimes are averaged over 10 trials per input size to mitigate noise, and garbage collection triggered only before trials. For each experiment, we vary n from 1 000 to 100 000 jobs and report two metrics. The first metric is **Overall runtime (seconds)**, which measures wall-clock time to solve a complete instance. The second metric is **per-job runtime (seconds/job)** which is computed as total time divided by n and is useful for visualizing marginal cost and highlighting logarithmic overhead within the classical solution.

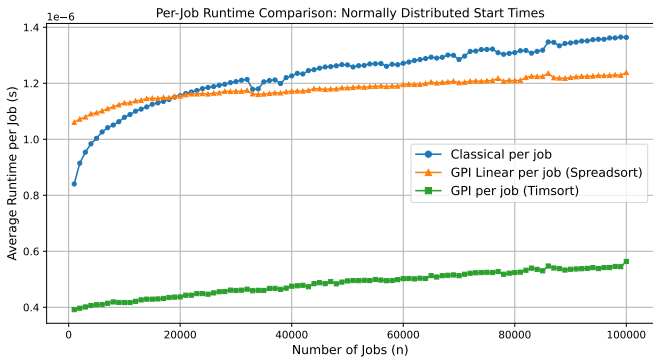
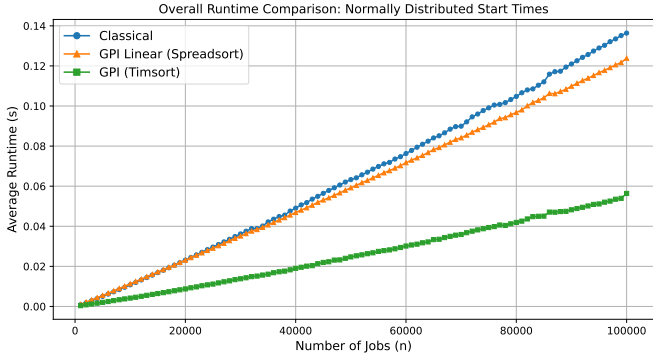
4.2. Experiment Details

Experiment 1: Random Integer Times. Job start and end times are drawn uniformly from $[0, 10^6]$, representing generic conditions with arbitrary durations, random overlaps, and diverse predecessor relationships. This baseline test resembles mixed workloads in general-purpose scheduling applications. GPI Linear uses Radix Sort as job times are bounded-width integers.

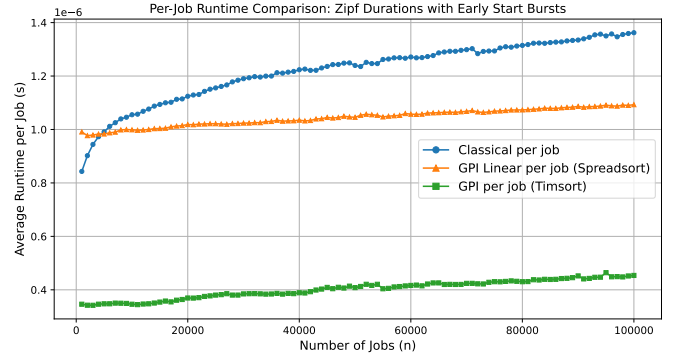
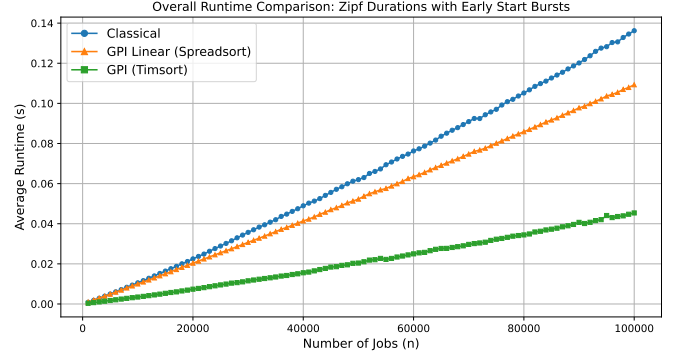
¹<https://github.com/amitjoshi2724/gpi-job-scheduling>



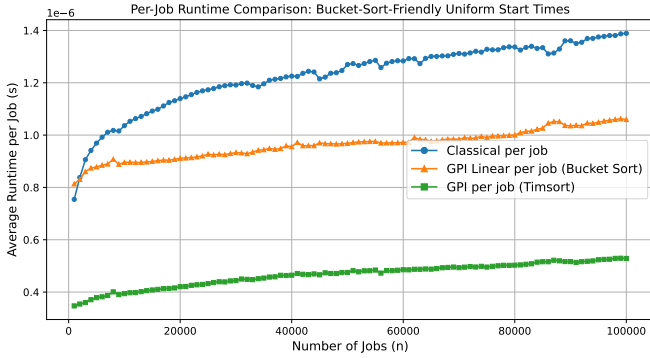
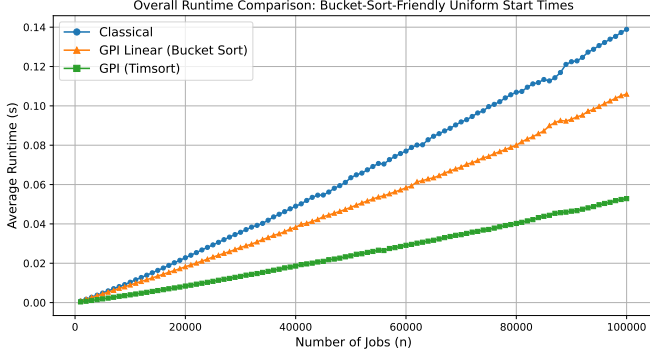
Experiment 2: Normally Distributed Start Times. Job start times are drawn from a truncated normal distribution $N\left(\frac{K}{2}, \left(\frac{K}{10}\right)^2\right)$ within $[0, K = 10^9]$, and float durations are sampled uniformly from $[1, 10^6]$. This models job bursts in real-world systems with smooth, dense clusters. GPI Linear uses Spreadsort, which efficiently handles such non-uniform input distributions.



Experiment 3: Zipf Durations with Early Start Bursts. Job start times follow a truncated exponential distribution with scale $\frac{K}{10}$ (clustered near 0), modeling early start bursts in real-world workloads. Durations follow a scaled Zipf distribution with exponent $s = 2$ (i.e., $d = 100 \cdot Z$ for $Z \sim \text{Zipf}(2)$), capped at 10^6 . This yields mostly short jobs with few long ones, causing some jobs to have nearby predecessors while others have distant ones. This disrupts cache locality during predecessor lookups during the DP. GPI Linear uses Spreadsort for this heavily skewed input distribution.



Experiment 4: Bucket-Sort-Friendly Uniform Start Times. Job start times are drawn uniformly from $[0, K = 10^9]$ so $K \gg n$, and float durations are sampled uniformly from $[1, 10^6]$, so the end times are approximately uniform as well. This produces a well-spread timeline ideal for Bucket Sort, which our GPI Linear uses. The uniform distribution ensures minimal collisions per bucket. Collisions are sorted using Timsort (a hybrid merge/insertion sort that exploits ordered subarrays [6]), highlighting Bucket Sort's strengths when input times are approximately uniformly distributed across a wide domain.



4.3. Discussion

Our results align with theoretical expectations. The classical solution always shows mild upward curvature in the Overall Runtime charts, consistent with its $O(n \log(n))$ time complexity. In the Per-Job Runtime charts, Classical DP displays a clear logarithmic trend, reflecting the increasing marginal cost of processing each additional job.

GPI Linear in Experiments 1 and 4 achieves true linear scaling, producing linear slopes in Overall Runtime charts with flat Per-Job Runtime curves, confirming constant marginal cost per job. While GPI Linear shows a slight upward slant in per-job runtime, this reflects system-level overhead (possibly due to cache misses) rather than algorithmic inefficiency. The scaling outperforms the classical solution with increasing n and supports our linear-time claim.

GPI Linear in Experiments 2 and 3 uses Spreadsor on non-uniform input distributions and achieves good speedups over the classical solution despite Python-C++ conversion overhead via Pybind11. While speedups are less pronounced than in Experiments 1 and 4, performance improves with larger n and Per-Job Runtime curves remain relatively flat, demonstrating near-linear runtime and near-constant marginal costs even under heavily skewed, non-ideal conditions.

GPI with Timsort consistently outperforms all methods by eliminating the per-job binary search bottleneck. Despite a very slight logarithmic curve in per-job runtime, overall performance remains near-linear. At $n = 100,000$, GPI Timsort runs 2-3 times faster than the classical solution—an overwhelming practical speedup despite identical $O(n \log(n))$ theoretical worst-case complexity.

5. Conclusion

We have shown that GPI significantly accelerates WJS in the general case by avoiding binary search, and proven our algorithm’s correctness. When job times permit linear-time sorting, GPI solves WJS in $O(n)$ time, beating the classical solution’s $O(n \log(n))$. Even with comparison-based sorting, GPI greatly outperforms the classical solution in practice, and outperforms GPI paired with linear-time sorts. GPI sorts twice, and then replaces per-job binary searches with two linear-time passes to find all predecessors and uses array-indexed predecessor lookups during DP. Our approach gives a theoretical and practical speedup over the classical solution as well as being conceptually simple, cache-friendly, and even parallelizable. We hope that our technique of sorting by one key, expanding states to store this sort order, and then sorting by another key will be used to solve other problems as well.

References

- [1] J. Kleinberg, Éva Tardos, Algorithm Design, 1st Edition, Pearson/Addison-Wesley, Boston, MA, 2005.
- [2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to Algorithms, 3rd Edition, MIT Press, 2009.
- [3] A. V. Aho, J. E. Hopcroft, J. D. Ullman, Data Structures and Algorithms, 1st Edition, Addison-Wesley, 1983.
- [4] S. J. Ross, The spreadsor high-performance general-case sorting algorithm, in: Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA), Vol. 3, CSREA Press, 2002, pp. 1100–1106.
- [5] A. B. Kahn, Topological sorting of large networks, Communications of the ACM 5 (11) (1962) 558–562. doi:10.1145/368996.369025. URL <https://doi.org/10.1145/368996.369025>
- [6] T. Peters, Timsort: A hybrid stable sorting algorithm, <https://github.com/python/cpython/blob/main/Objects/listsort.txt> (2002).