IPCA
Technology Department
Applied Artificial Intelligence
High Performance Computing

# Parallelization of a JSSP algorithm

Aluno: Celestino Machado 21190
Aluno: Bernardo Neves 23494
Professor: Nuno Alberto Ferreira Lopes

Barcelos
2025

# Contents

# List of Figures

# 1 Introduction

The Job Shop Scheduling Problem (JSSP) is a classical combinatorial optimization problem with significant practical relevance in manufacturing, logistics, and computing systems. It involves scheduling a set of jobs, each consisting of a sequence of operations, across a limited number of machines. The goal is typically to minimize the makespan, which is the total time required to complete all jobs, while satisfying the precedence constraints of operations and ensuring that each machine processes only one task at a time.

JSSP is known for its computational complexity; it belongs to the class of NP hard problems, meaning that the time required to solve it grows exponentially with the size of the instance. Due to this, exact methods become impractical for large instances, and heuristic or metaheuristic approaches are often adopted to find near optimal solutions within reasonable time frames.

Parallel computing offers a promising avenue for accelerating the solution of such computationally intensive problems. By decomposing the problem or its search process across multiple processors, it becomes possible to explore the solution space more efficiently and reduce overall computation time. In this context, various parallelization strategies can be applied, ranging from simple multi threaded dispatching rules to more advanced techniques like task parallelism and Foster's methodology.

This report investigates both sequential and parallel approaches to solving the JSSP, beginning with a hybrid sequential algorithm and exploring parallelization methods for performance improvement.

# 2 Proposed Hybrid Sequential Algorithm

The developed algorithm integrates multiple heuristic strategies to solve the Job Shop Scheduling Problem (JSSP), aiming to minimize the makespan. This section describes the main components of the current sequential algorithm implemented in C.

## 2.1 Heuristic Dispatching Rules

Dispatching rules are classical heuristics that assign scheduling priority to operations based on local criteria. The algorithm supports the following rules:

- **SPT (Shortest Processing Time)**: prioritizes operations with the shortest processing time;
- **LPT (Longest Processing Time)**: prioritizes operations with the longest processing time;
- **SRT (Shortest Remaining Time)**: prioritizes jobs with the least total remaining processing time;
- **MWKR (Most Work Remaining)**: prioritizes jobs with the highest remaining workload;
- **FIFO (First-In First-Out)**: prioritizes jobs in order of appearance;
- **MOR (Most Operations Remaining)**: prioritizes jobs with the most remaining operations;
- **Random**: assigns priorities randomly to diversify scheduling decisions.

Each scheduling step identifies all ready operations and calculates their priority using the selected rule. A struct of arrays format is used to store operation data efficiently in the `operation_batch_t` structure. The best operation is then scheduled based on the lowest priority value. This logic is implemented in the `solve_with_dispatch_rule_sequential(...)` function.

## 2.2 Multi-Pass Heuristic

To improve solution robustness, the algorithm executes multiple passes using different dispatching rules. Each pass builds a complete schedule independently. After evaluating all rules, the solution with the lowest makespan is selected. This approach increases schedule diversity and reduces the impact of any single rule's bias. It is implemented in the `solve_multi_pass(...)` function.

```
switch (rule) {
case SHORTEST_PROCESSING_TIME:
    ready_ops.priorities[idx] = ready_ops.processing_times[idx];
    break;
case LONGEST_PROCESSING_TIME:
    ready_ops.priorities[idx] = -ready_ops.processing_times[idx];
    break;
case SHORTEST_REMAINING_TIME:
    ready_ops.priorities[idx] = ready_ops.remaining_works[idx];
    break;
case MOST_WORK_REMAINING:
    ready_ops.priorities[idx] = -ready_ops.remaining_works[idx];
    break;
case FIRST_IN_FIRST_OUT:
    ready_ops.priorities[idx] = job_id;
    break;
case MOST_OPERATIONS_REMAINING:
    ready_ops.priorities[idx] = -(jss->num_machines - op_idx);
    break;
case RANDOM:
    ready_ops.priorities[idx] = rand() % 1000;
    break;
default:
    ready_ops.priorities[idx] = job_id; // FIFO as fallback
}
ready_ops.count++;
```

Figure 1: Dispatch Rules.

## 2.3 Final Integration and Execution

The `solve_sequential(...)` function acts as the main entry point, invoking the multi-pass heuristic and returning the best solution found. While additional optimization strategies like the shifted bottleneck method or random restart were considered during development, they are not included in the final implementation. Execution time is measured externally using system timers during benchmarking.

```
for (int i = 0; i < num_rules; i++) {
    int makespan = solve_with_dispatch_rule_sequential(jss, &temp_solutions[i], rules[i]);

    results[i].makespan = makespan;
    results[i].rule_used = rules[i];
}

// Find best result
int best_idx = 0;
for (int i = 1; i < num_rules; i++) {
    if (results[i].makespan < results[best_idx].makespan) {
        best_idx = i;
    }
}

// Copy best solution
memcpy(solution, &temp_solutions[best_idx], sizeof(jobshop_solution_t));
return results[best_idx].makespan;
```

Figure 2: Multi Pass Heuristic.

# 3 Parallelization

## 3.1 Objectives of the Parallel Implementation

The parallel version of the algorithm was implemented with the following goals:

- Accepts three command line arguments: the input file, the output file, and the number of threads.
- Enables performance comparison with the sequential version, focusing on the start times of operations and the completion time of the last scheduled operation.

## 3.2 Parallelization Strategy

Instead of parallelizing the internal scheduling logic which is inherently sequential due to dependencies between operations, the parallel implementation explores task level parallelism by evaluating different dispatching rules concurrently. Each rule generates a complete and independent schedule. This strategy ensures correctness and avoids concurrency issues, while effectively leveraging multicore processors.

## 3.3 Program Components in the Parallel Version

**i. Data structures used** The program uses the same base structures as in the sequential version: `jobshop_t` for problem instances and `jobshop_solution_t` for resulting schedules. Additionally, `operation_batch_t` is used to manage schedulable operations at each iteration, and `solution_summary_t` stores intermediate results for each thread.

**ii. Thread initialization code** Threads are created using OpenMP. The number of threads is set based on the input parameter `parallel.c`, function `solve_multi_pass_parallel`).

```
int solve_multi_pass_parallel(const jobshop_t* jss, jobshop_solution_t* solution, int num_threads)
    omp_set_num_threads(num_threads);
```

Figure 3: Number of Threads.

**iii. Code executed by each thread** Each thread applies a different dispatching rule and computes a full solution independently. This is done inside a parallel loop where each iteration is self-contained.

**iv. Final code of the program** After all threads complete, the program compares the results and selects the best one, which is then copied to the final output structure.

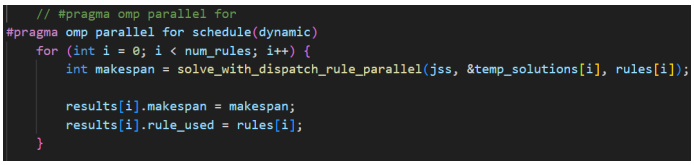


Figure 4: Code in Each Thread.

## 3.4 Program Characteristics

**Shared and private variables**

- **Shared (read-only):** `jss`, `rules`, and `num_rules`.
- **Shared (indexed write):** `temp_solutions[]` and `results[]` — each thread writes to a distinct index.
- **Private:** All loop and local variables, such as scheduling buffers and operation counters, inside the function `solve_with_dispatch_rule_parallel`.

**Critical sections and race conditions** There are no explicit critical sections or race conditions because all data written by a thread is isolated by index. Each thread solves a rule independently and does not share intermediate results with others. As such, mutual exclusion mechanisms are not required.

**Mutual exclusion techniques** None are necessary in this implementation due to the independence of data accessed by each thread. This design guarantees both determinism and correctness without relying on OpenMP's `critical` or atomic directives.

## 3.5 Comparison with the Sequential Version

The core scheduling logic remains consistent across both implementations. However, the parallel version significantly reduces the overall runtime, especially in larger problem instances, by computing multiple dispatching strategies concurrently. While the operation start times vary between strategies, the selected solution always reflects the best makespan found. The difference in performance can be observed in both execution time and solution quality.

# 4 Performance Analysis

## 4.1 Graphs


*Available processors: 12, Instances: 31, Executions per instance: 10*

Figure 5: Benchmark Results

## 4.2 Speedup Analysis

Speedup ($S(p)$) measures how much faster the parallel implementation runs compared to the sequential version, defined as:

$$S(p) = \frac{T(1)}{T(p)}$$

Where $T(1)$ is the time to run with the sequential configuration (SEQ) and $T(p)$ is the time with $p$ parallel instances. Based on the benchmark results (Figure 5), the parallel implementation for configuration P1 shows a slight slowdown ($0.97\times$) compared to sequential. However, speedup improves significantly with more instances, reaching its peak at P16 with a speedup factor of $3.44\times$. Beyond P16, specifically at P32, the speedup slightly declines to $3.38\times$. This behavior suggests that while parallelism is beneficial, excessive instances can introduce overhead that diminishes further performance gains.

## 4.3 Efficiency Analysis

Efficiency ($E(p)$) evaluates how effectively the available processing units are utilized:

$$E(p) = \frac{S(p)}{p}$$

While the graphs do not directly show efficiency values, we can infer from the speedup and the number of instances. The speedup analysis indicates that the system is most efficient around P16. For instance, at P16, with a speedup of $3.44\times$, the efficiency would be $3.44/16 \approx 0.215$ or 21.5%. The decline in speedup from P16 to P32 suggests a corresponding drop in efficiency as the system attempts to utilize more instances than optimal for the given workload and available resources (12 available processors). This is expected as the overheads associated with managing a larger number of instances (31 instances, 10 executions per instance) begin to outweigh the benefits of additional parallelism.

## 4.4 Scalability Discussion

The implementation demonstrates good scalability up to the P16 configuration. Beyond this point (P32), the performance gains plateau and slightly decline, indicating that the system's scalability limit is reached or exceeded under the given conditions. The observed speedup of $3.44\times$ at P16, with 12 available processors and 31 instances, suggests that factors beyond just the number of "dispatch rules" influence scalability. The optimal performance at P16 implies an effective utilization of the available computational resources for the workload. While task-level parallelism is easy to implement, the system appears to reach a point where increasing the number of instances does not yield proportionate returns, possibly due to a combination of fixed sequential portions of the task, communication overheads, or resource contention.

### 4.5 Overheads and Limitations

While task-level parallelism proved effective in achieving significant speedups, particularly up to P16, it is clear that overheads become a limiting factor for higher levels of parallelism (P32). The initial slowdown observed at P1 (0.97x speedup) highlights that even introducing parallelism for a single instance can incur overhead. The decline in speedup from P16 to P32 further reinforces the presence of increasing overheads associated with managing a larger number of parallel instances (31 instances running 10 executions each), potentially related to scheduling, communication, or resource contention on the 12 available processors. Memory usage may also play a role, as per-thread data structures increase linearly. Future improvements could focus on optimizing these overheads and exploring finer-grained parallelism to better utilize resources without incurring excessive management costs.

# 5 Conclusion

This project demonstrated a hybrid sequential and parallel approach to solving the Job Shop Scheduling Problem (JSSP) using multiple dispatching rules. The parallel implementation leveraged task-level parallelism by assigning each dispatching rule to a separate instance. This approach avoids race conditions and ensures correctness.

The parallel solution achieved notable speedups over the sequential version, particularly in large instances. The best performance was observed with the P16 configuration, yielding a speedup of 3.44× over the sequential baseline. Performance at P1 was slightly slower than sequential, and beyond P16 (at P32), the speedup marginally declined, indicating an optimal point for parallelism given the system's characteristics and available resources.

Key design decisions—such as dynamic scheduling and thread-isolated buffers, likely contributed to the observed performance gains and avoidance of synchronization bottlenecks. However, the approach is constrained by factors such as the overhead of managing a large number of parallel instances, particularly evident at P32.

Future work may involve:

- Parallelizing the inner scheduling logic using finer-grained task decomposition
- Exploring dynamic load balancing using work-stealing
- Reducing memory overhead via buffer pooling or shared partial solutions
- Integrating metaheuristics or reinforcement learning to guide rule selection

Overall, the project highlights the effectiveness of simple parallelism in heuristic-based scheduling while exposing the trade-offs between ease of implementation and scalability under specific resource constraints.