Real-Time Operating Systems

Multiprocessing Scheduling Simulator

Siddharth SAHAY
Jordi UGARTE

Prof. Joël GOOSSENS, Yannick MOLINGHEN

December 2024

Contents

Abstract

Multiprocessors are ubiquitous, our phones, laptops, remote servers, all mostly operate on more than a single processor. The ability to execute tasks simultaneously is how we increase the efficiency of our systems. With this definition, tasks can also be divided into smaller parts to improve their execution times. This is referred to as Parallelization, as the technique of dividing a task into smaller independent parts that can be executed simultaneously to improve performance by a multiprocessor.

This project involves the development of a simulator of a multi-threaded tool that checks the schedulability of asynchronous tasks with arbitrary deadlines in the scope of multiprocessor scheduling with several identical cores and according to different real-time multiprocessor scheduling algorithms such as: *Partitioned Scheduling, Global EDF Scheduling* and *EDF(k)*. The project also involves the analysis of the execution times of these algorithms with multiple tasksets and with a variable number of workers.

This report dives into the analysis performed regarding these scheduling algorithms, and how they perform in manners of time, quantity of workers and parallelization and when is it good to implement.

1 Introduction

1.1 Background

Parallelization

Parallelization in Real-Time Operative Systems is the process of executing multiple tasks simultaneously and dividing them into smaller sub-tasks. This is performed to achieve efficiency and performance. Parallelization ensures the following concepts in run-time:

- Concurrency: Simultaneous execution in real-time by multiple cores or processors of the CPU.
- Granularity: Refers to the sizes assigned to the sub-tasks and their communication between them.

The degree to which a task can be parallelized depends on its nature, dependencies, and communication overhead. This boundaries on which tasks can be executed in parallel can be easily denoted by the Amdahl's Law. This law considers the fraction of the tasks that can be parallelized by the following speedup fraction:

$$S(n) = \frac{n}{1 + (n-1) \cdot f}$$

Where n is the number of cores and f the fraction of sequential computations of a parallel program (factor of tasks that cannot be parallelized). When the speedup factor is equals or close to 1, it means that parallelization took no effect on the performance of the execution. Therefore, is not parallelizable because of the dependant or undividable tasks.

Algorithms

Multi-processor algorithms are techniques used to allocate tasks to multiple processors in a computer system to optimize performance, minimize task completion time and efficiently utilize resources. These algorithms are essential in systems where multiple CPUs or cores work together. These techniques are used in multiprocessing computers, distributed systems or super computers. Nowadays, almost every computer has more than one core within their processors. The scheduling algorithms to be studied in this project are the following:

- Partitioned Scheduling
- Global Scheduling

- EDF(k)

These algorithms need to be ran in tasksets that need to be feasible. Feasibility refers to the ability of the system to meet all its timing requirements or deadlines for tasks within given constraints. It is necessary to certify real-time applications and machines so that it can be known in advance that the systems will not fail in real-life situations. We must therefore study these algorithms, determine optimality and figure out applications ahead of time. The algorithms considered are the following:

- **Partitioned Scheduling:** Partitioned algorithms are a foundational concept in parallel and distributed computing. It allows efficient utilization of multiple processors by dividing the problem into smaller manageable parts. To achieve this, the input data is divided into smaller parts assigned to different processors. Tasks are split based on the data partitions, ensuring that each processor handles only its own share of the workload. It is necessary to ensure that the workload is evenly distributed among the processors to avoid bottlenecks caused by some processors being overworked while others remain idle.

- **Global Scheduling:** It is a scheduling approach where tasks are dynamically assigned to processors from a common global queue. Unlike partitioned scheduling, where tasks are statically assigned to specific processors, global scheduling allows tasks to be scheduled on any processor in the system, providing more flexibility and adaptability. Tasks are not statically bound to specific processors. Any processor that becomes idle can take a task from the global queue.

- **EDF(k):** The EDF(k) algorithm is like the base EDF but adapted for multiprocessor systems. It introduces the concept of limiting the number of processors a task can use to improve schedulability and manage overhead in multiprocessor environments. Each task is limited to being scheduled on at most k processors simultaneously. The value of k can vary depending on the task and the system. If k is 1 the task behaves like in a partitioned system. If k equals to the number of processors, the task can potentially execute on all processors, as in global EDF.

2

Methodology

2.1 Implementation Overview

For this multi-processor simulator project, we implemented EDF-k, Global EDF, and Partitioned EDF schedulers using Python, reusing components from our previous uniprocessor simulation project for efficiency when it made sense. Core entities such as *Task*, *Job*, and *TaskSet* are implemented in *entities.py*, with *TaskSet* managing tasks defined by attributes like offset (O), computation time (C), deadline (D), and period (T). The main *EDFk* class in *scheduler.py* provides shared functionality for all schedulers, while child classes override behavior to define specific algorithms.

Simulations and visualizations, including execution times and worker requirements, are generated through *MultiProcessorSchedulerAnalysis.ipynb*, allowing for systematic analysis of scheduling performance of all the algorithms in question.

2.2 Implementation of Algorithms

2.2.1 The Scheduler Class

The `EDFk` class serves as an abstract base class for implementing different scheduling algorithms for multiprocessor implementations, encapsulating common functionality and structure for the subclasses representing the mentioned necessary algorithms. This was done because *EDFk* is a special case for *Global*, where $k = 1$ and *Partitioned EDF*, where $m = k$.

The class is initialized with a `TaskSet`, which holds the tasks to be scheduled. The classes or algorithms that inherit from this class are the Partitioned and Global algorithms. It contains and considers important attributes and functions such as:

2.2.1.1 Attributes

- `task_set`: The set of tasks to be scheduled.
- `k`: Number of clusters in the scheduler. In case of *Partitioned EDF*, $k = 1$, and for *Global EDF*, $k = m$.
- `m`: Total number of processors available.
- `heuristic`: The partitioning heuristic used to divide tasks into clusters. This is unused for *Global EDF* since there is only one cluster.

- `clusters`: List of `EDFCluster` objects representing the clusters. This is a class that helps organise each of the clusters during scheduling and also simply just holding the tasks while partitioning.
- `scheduler_type`: Type of multiprocessor scheduler (default: `EDF_K`).
- `num_workers`: Number of workers for parallel simulations. This could potentially be used for parallelising different clusters, since these are, by definition, independent of each other.
- `verbose`: Boolean flag for detailed logging.
- `force_simulation`: Boolean flag indicating whether to force simulation. Meant for debugging only.

2.2.1.2 Methods

- init_clusters(): Initializes clusters based on the number of clusters (`k`) and processors (`m`).
- partition_taskset(): Partitions tasks into clusters using the specified heuristic.
- pretty_print_clusters(): Prints details of each cluster, including utilization and tasks, if verbose mode is enabled.
- remove_empty_clusters(): Removes clusters with zero utilization (i.e., no assigned tasks).

2.2.1.3 Feasibility Checks

- calculate_m_min(): Calculates the minimum number of processors required to schedule the task set.
- check_utilisation_bound(): Checks if the utilization bound condition is met for each cluster.
- pretty_print_clusters(): Prints details of each cluster, including utilization and tasks, if verbose mode is enabled.
- remove_empty_clusters(): Removes clusters with zero utilization (i.e., no assigned tasks).
- check_density(): Verifies the density condition for the task set and clusters.
- is_feasible(): Main method to determine if the task set is schedulable. This method must define how it evaluates the schedulability of the task set according to its unique algorithm.

2.2.1.4 Simulation

- simulate_taskset(): Simulates the scheduling of tasks in a cluster.
- get_simulation_interval_for_cluster(): Determines the simulation interval for a cluster based on task hyperperiods. This will be overridden by each sub class.
- is_task_set_too_long(): Checks if the simulation duration exceeds the allowed limit.

2.2.2 Simulation Function

The `simulate_taskset()` method provides a generalized structure for task set simulation, which any algorithm can potentially use to confirm feasibility by simulating job executions over time. The simulation loop evaluates deadlines, checks for job releases, and simulates scheduling active tasks. This design ensures that all subclasses have access to a consistent simulation process, with customizable logic for each algorithm's unique scheduling strategy.

2.2.3 Feasibility Interval Calculation

The `get_simulation_interval()` method, also abstract, enables each subclass to determine its feasibility interval based on either a fixed period or a dynamic interval such as a busy period. The `is_task_set_too_long()` helper method checks if the interval length exceeds a specified limit, helping in excluding overly lengthy task sets early.

2.2.4 Schedulability of a tasksets

To determine the schedulability of a taskset, first, it is considered the multiprocessor algorithm (Global, Partitioned, EDF(k)). For the EDF(k) and Partitioned Algorithm, it is necessary to attempt to partition the taskset. If this process fails, the taskset cannot be schedulable under these algorithms. Finally the feasibility per cluster is checked on any algorithm as a final process. At the end, the algorithms are tested if they actually succeed to execute its taskset. This process can be represented on Figure 2.1.

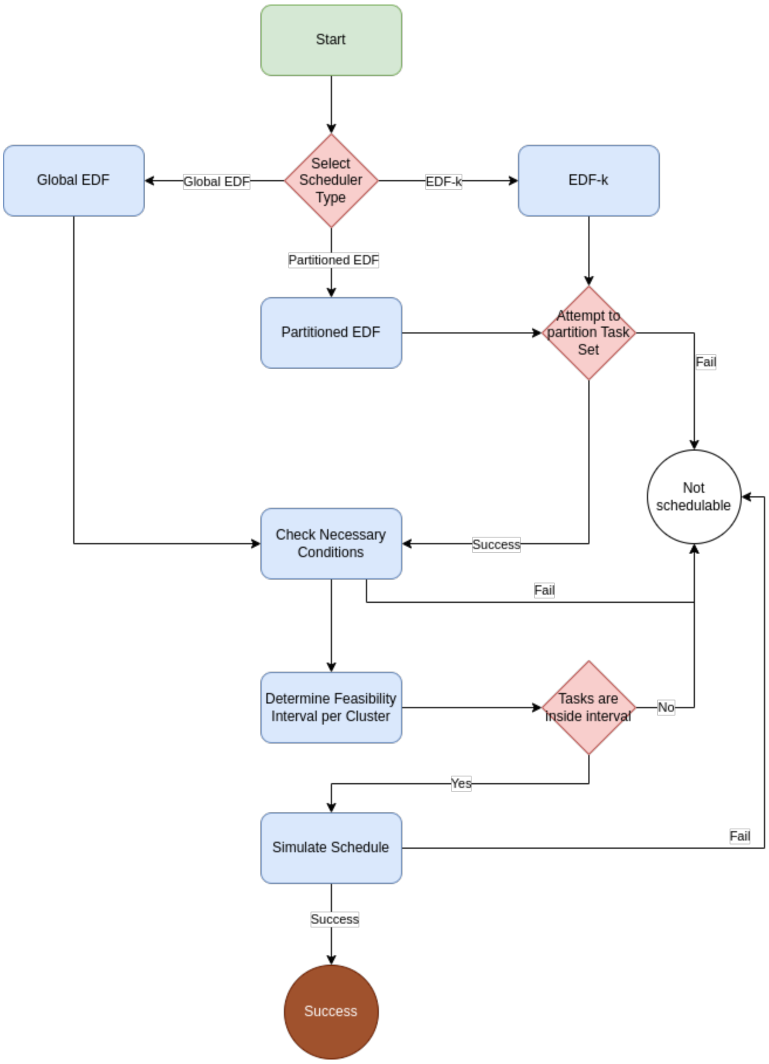


Figure 2.1: Flow diagram to determine the schedulability of a taskset.

3

Experimental Setup

3.1 Usage

To run the code, use the following command:

```
$python main.py <task_file> <m> global|partitioned|<k> [-w <w>] [-H ff|nf|bf|wf] [-s iu|du] [-v]
```

- **Algorithm**: Choose from `global`, `partitioned`, or `EDF(k)`) algorithms.
  - Heuristic (-*H*): First-fit, Next-fit, Best-fit, Worst-Fit.
  - Ordering of tasks (-*s*): Increasing Utilization, Decreasing Utilization.
- **Task Set Location**:
  - If a **directory** is specified, it evaluates all task sets within the folder (recursively) in parallel (subfolders are also considered).
  - If a **single task set file** is specified, the code evaluates just that set of tasks from that file.
- **Number of processors (*m*)**: Number of processors the algorithm uses to schedule the tasksets.
- **Verbose (-*v*)**: Provides detailed output including the whole schedule, if a simulation is done, and a text output instead of just the return code.
- **Quantity of workers (-*w*)**: Number of workers to run the simulation (by default it is the number of cores of the computer, 8 for us).

3.2 Exit codes

The program returns specific codes to indicate the outcome of each task set evaluation and its schedulability. These return codes provide insight into both the schedulability of each task set and whether the evaluation required a full simulation:

- **0**: The task set is schedulable, and the simulation was required.
- **1**: The task set is schedulable because some sufficient condition is met.
- **2**: The task set is not schedulable, and the simulation was required.

- **3**: The task set is not schedulable, and a shortcut was used instead of full simulation. A necessary condition does not hold.
- **4**: Time-out or undecidable, this was added on top of the existing return codes to account for the many task sets that had too long a feasibility interval to do a simulation.

In case of multiple tasksets execution in a folder or subfolders, the output specifies as well each execution time of them. Also it encludes a final count of the task sets that received depending on the exit codes they received from the above table.

3.3 Dataset Structure

The dataset is organized into one main folder. It contains a 1000 task sets in total with no patterns or ordering whatsoever. These tasksets will be tested later by an algorithm testing and analyzing its time execution and performance by varying the number of workers.

3.4 Evaluation Metrics

The key metrics we use to assess algorithm performance are:

- **Schedulability**: Whether a task set meets its deadlines under each algorithm.
- **Success Rate**: Calculated as the ratio of schedulable task sets to total task sets for each algorithm.
- **Execution Time**: Calculate how juch time the algorithm spends on executing all required tasksets.

4 Experiment Results

All the tests were run on a PC with the following specifications:

Intel® Core™ i5-8365U × 8

8.0Gb Memory

4.1 Main performance test - Partitioned EDF

From the task set data, it is required to test the performance of the Partitioned *EDF(k)* algorithm by analyzing its time execution rate for each number of workers considered for the execution of the algorithm. In this case, as required for this assignment, the algorithm is run on 32 cases; 1 to 32 number of workers for each run of the algorithm. By running the simulation with an ascending number of workers, it is possible to analyze the algorithm with different number of workers and to evaluate how this parameter influences the performance. By default, the algorithm runs with an execution of 8 cores and with an heuristic of *Best-Fit* and *Decreasing Utilization* ordering. It is possible to measure how well the algorithm performs over lifetime and which number of workers gives their most optimal situation.

Each test runs 10 times to get a mean value considering the fact that the results can vary depending on the current state of the real CPU being tested on.

4.2 Algorithm analysis - Partitioned EDF

The results of the Partitioned EDF(k) algorithm are shown of Figure 4.1. It was executed with 1000 tasksets, each one with varying quantities of tasks. These tasksets were executed from 1 to 32 workers, each test running 10 times to get a more approximated value by obtaining the mean of the results. The results show that, when increasing from 1 to multiple number of workers *(m > 1)*, it drastically increases its performance by almost 2 seconds. The most optimal point is when the number of workers is closes to the number of cores of the physical CPU of computer where the test was executed. However, the execution times start to slightly increase when the number of workers get greater in quantity than the number of cores of the computer. This is because the system gets saturated because of multiple threads that try to compete for the use of the CPU causing invalidations and slower access to the memory. Generally speaking, from 1 to 32 workers, the tasksets execution lasts approximately 4 seconds.

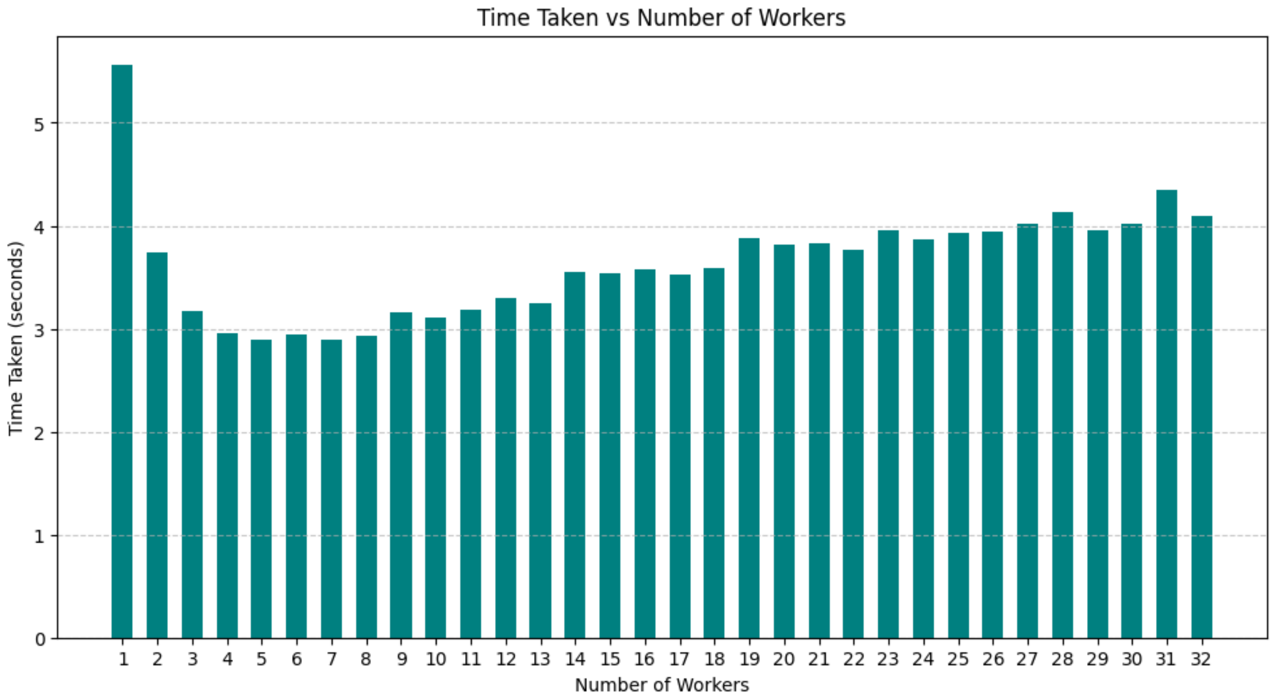


Figure 4.1: Partitioned EDF(k) execution times from 1 to 32 workers assigned.

4.3 Parallelization

There were multiple aspects of the code that could have been parallelised. These can sometimes be overlapping and sometimes not:

4.3.1 At the task set level

Tasksets are independent so we can process multiple tasksets concurrently. This would mean that the number of workers specified by the user can go towards computing feasibility for that many task sets simultaneously. The downside of this choice is that if we decide to use threads (daemonic processes), we cannot further fork into more threads later on and we "limit" parallelization at this level.

4.3.2 At the algorithm level

Within scheduling algorithms, each cluster can be simulated concurrently. This is especially true once the initial conditions have been checked and it's made certain that simulation is the only way to verify feasibility. This is true for *Partitioned EDF* and *EDF-k* since they both have more than one cluster. For *Global EDF* it doesn't make sense to use more than one worker since there exists only one cluster, which cannot be further parellelised.

We have decided to use parallelization at the taskset level since it is a more much more consistent

4.4 Comparison of heuristics

Below we compare partitioning heuristics. This is a known NP-hard problem. We also call these heuristics because there is no known "best" way to partition tasks into clusters. There are some that suit better in some circumstances than others, but in general there is no one winner for all scenarios.

4.4.1 First-Fit

First-Fit is one of the simplest methods, quickly assigning each task to the first processor that can accommodate it. This straightforward approach minimizes overhead and is easy to implement, which makes it practical in time-critical or resource-limited scenarios. However, by not striving for better load balancing, it may lead to cluster imbalances and inefficient utilization of available capacity!

4.4.2 Best-Fit

Best-Fit, in contrast, aims to use the available capacity more effectively by placing a new task into the most heavily loaded cluster that can still fit it. The advantage is that this often reduces fragmentation, potentially leading to more compact packing of tasks. It also leaves processors fully available for other incoming tasks. Yet, this comes at a cost: it can be more computationally expensive due to the need for sorting or scanning, and by pushing tasks into tighter clusters, and as we found, it risks creating "regions" of high load where missed deadlines become more likely! It is indeed not the "best" heuristic we study - it can arguably backfire during simulation since all processors are close to the utilisation of 1, and since the offsets are not taken into consideration while partitioning, it can quickly lead to deadline misses!

4.4.3 Worst-Fit

Worst-Fit tries to avoid concentrating too many tasks in one place by always assigning the incoming task to the least loaded cluster. This often leads to a more even distribution, mitigating the risk of local overload. However, spreading tasks too thin can result in scenarios where no cluster is well utilized, causing inefficient use of resources and limiting potential gains in performance.

4.4.4 Next-Fit

Next-Fit takes a different approach, sequentially placing tasks into clusters and moving on to the next one when the current one cannot handle the load. In our report, we do not study this heuristic very deeply.

In Figure 4.2 we see that all the heuristics perform at roughly the same level while partitioning itself. These comparisons are do not necessarily tell us much about how well the algorithm performs post-partitioning.

4.4.5 Performance of each heuristic for Partitioned EDF

Here we take $m = 12$ since $m = 8$ was found to be slightly low for a good comparison as most of the tasks in our tasksets are infeasible due to high utilisation.
We compare the performance of four heuristics—**Best Fit**, **Worst Fit**, **First Fit**, and **Next Fit**—in determining the feasibility of task sets under the Partitioned-EDF scheduling approach. This is shown in Figure 4.3 The following insights highlight the key findings.

The findings are also summarised in Table 4.1.

- **Worst Fit** emerges as the *most effective* heuristic, successfully making a higher number of task sets feasible. Despite distributing tasks less "greedily," its ability to balance loads across processors leads to significantly fewer failures overall.

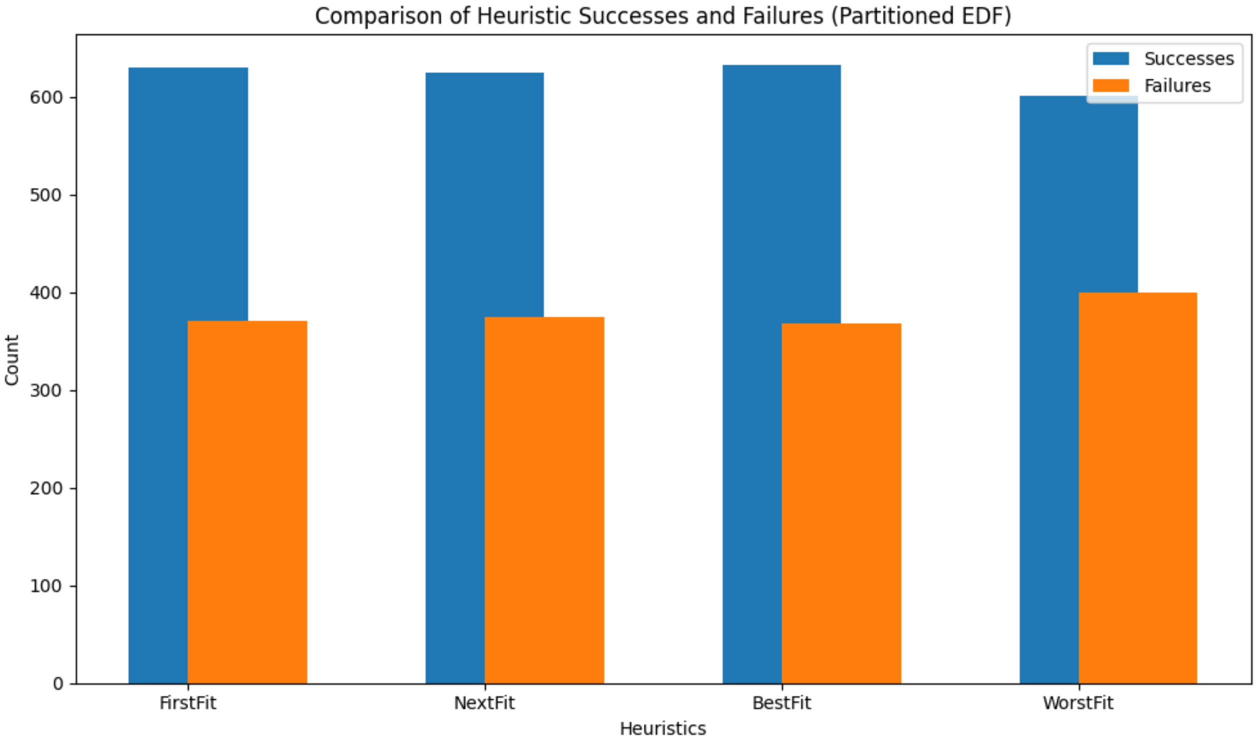


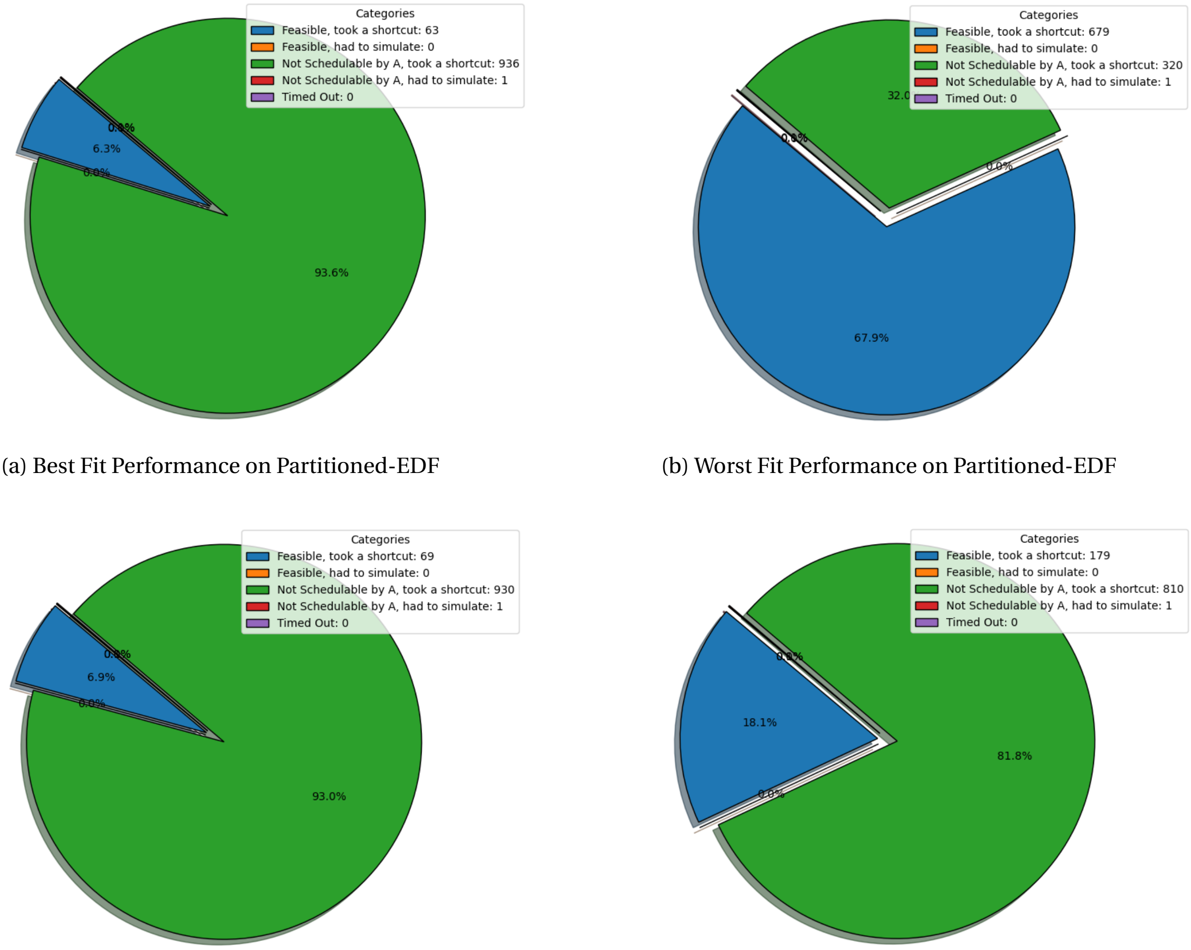
Figure 4.2: Heuristic Performance for Partitioned EDF - Partitioning Only

Table 4.1: Heuristic Performance on Partitioned-EDF Scheduling

Heuristic	**Feasibility (%)**	**Failures (%)**
Best Fit	6.3	93.6
Worst Fit	**67.9**	**32.0**
First Fit	6.9	93.0
Next Fit	18.1	81.8

- **Best Fit** and **First Fit** perform poorly. Their aggressive strategies for tightly packing tasks often result in overloaded processors, making a vast majority of task sets unschedulable.
- **Next Fit** shows moderate improvement over Best and First Fit but still falls short compared to Worst Fit. While it partially avoids early overloads, it struggles to achieve optimal load balancing.

This is all to say that there is no best heuristic for any given problem and they all have their pros and cons.



(a) Best Fit Performance on Partitioned-EDF

(b) Worst Fit Performance on Partitioned-EDF

(c) First Fit Performance on Partitioned-EDF

(d) Next Fit Performance on Partitioned-EDF

Figure 4.3: Comparison of Heuristic Performances on Partitioned-EDF

5 Summary

This project explored and evaluated the performance of three multiprocessor scheduling algorithms—**Partitioned EDF**, **Global EDF**, and **EDF(k)**—using a custom scheduler simulator. Each algorithm represents a unique approach to task prioritization, task selection, and workload distribution. By analyzing these algorithms under various configurations, including different heuristics, task orderings, and worker setups, we uncovered valuable insights into their performance and applicabiliity.

Partitioned Algorithm: The Partitioned Algorithm, tested with a range of 1 to 32 workers and the best-fit heuristic with decreasing utilization, showed intriguing results. Execution times fluctuated depending on the number of CPU cores available and the number of workers configured. Despite this variability, the Partitioned Algorithm consistently demonstrated robustness and better performance when dealing with high task loads. Its static task allocation made it particularly effective for systems where tasks can be well-partitioned initially.

Global Algorithm: In contrast, the Global Algorithm required more time to execute. This was primarily due to its dynamic allocation of tasks and the overhead associated with task migrations. While this dynamic nature allows for greater flexibility, it also introduces additional decision-making overhead, making it less optimal for systems with heavy, static workloads. However, Global EDF shines in environments with highly dynamic or uneven task distributions where adaptability is crucial.

EDF(k) Algorithm: The EDF(k) algorithm serves as a very versatile middle ground between Partitioned and Global EDF. By tuning the number of clusters k, EDF(k) enables trade-offs between the strict load balancing of Partitioned EDF and the flexibility of Global EDF. This adaptability makes it suitable for systems where both static partitioning and dynamic scheduling are necessary to achieve optimal performance.

Heuristic Comparison: When comparing the heuristics (Best Fit, Worst Fit, First Fit, and Next Fit), several key observations were made:

- **Worst Fit** (despite its name) proved to be the most effective heuristic, successfully achieving the highest task set feasibility. By evenly balancing loads across processors, it mitigated local overloads and performed consistently well.
- **Best Fit** and **First Fit**, while computationally simple, performed poorly. Their "greedy" task packing strategies often led to overloaded processors and high failure rates.

- **Next Fit** offered moderate improvements over Best Fit and First Fit. It avoided some early overloads but still fell short of Worst Fit's balanced performance. It was able to distribute the tasks decently well across processors/-clusters.

To sum up, while each algorithm and heuristic has its strengths and weaknesses, their effectiveness depends heavily on the nature of the workload. For static workloads, the Partitioned Algorithm paired with Worst Fit emerges as the clear winner. For more dynamic systems, Global EDF and EDF(k) provide the flexibility needed to adapt to changing conditions.

Ultimately, this study highlights that there is no one-size-fits-all solution. The real challenge lies in understanding the system's requirements and selecting the right combination of algorithm and heuristic to achieve optimal performance. Future work could explore adaptive heuristics that dynamically balance load distribution and task migration to further improve scheduling outcomes. This seems to be the best way to partition tasks (dynamically).

In conclusion, the beauty of multiprocessor scheduling lies in its flexibility. The right algorithm and heuristic can make all the difference, transforming a system's performance and unlocking its full potential!