Sahith Bhamidipati <bhamis>, Soumya Mishra <mishrs4>, Rithik Mamidi <mamidr>

1. There is no decisive “best” algorithm as each is more appropriate for various sets of processes. If we were to evaluate solely on wait time in relation to the number of processes then the First Come First Serve algorithm would be the “best” algorithm based on our simulation results. Figure 1 depicts the different algorithms’ wait times with a constant seed of 2, lambda of 0.01, context-switch time of 4 milliseconds, alpha of 0.5, a random limit of 256, and a slice time of 128 milliseconds. Figure 2 mirrors the results in Figure 1 with the same parameters except with data on the turnaround times. Although the preemptive algorithms seem to fall short in comparison to the non-preemptive algorithms as the number of processes increase, it is clear that in the portion of the figure under 5 processes the preemptive algorithms dominate due to the lesser wait time. From this and other examples, we can see that different conditions cause different algorithms to be the “best”. If there were fewer processes to complete, one may choose SRT or RR as these algorithms would be more efficient, whereas a non-preemptive algorithm such as FCFS or SJF would be more appropriate for more processes. Another notable point is that, with the max number of processes, SJF has a lower wait time than FCFS, so it is possible that as the number of processes exceeds that of 26, SJF may have a lower wait time in comparison to FCFS.

The First Come First Served algorithm would be the best for CPU-bound processes. SJF and SRT prioritize processes based on their CPU burst times, hence a process with an extensive CPU time would not enter the CPU for a long time and therefore would cause starvation. FCFS and RR mitigate the negative effects of having a very long CPU bound process as they circulate through the processes with a fixed amount of time. So, essentially all processes are given the same priority. The only difference between FCFS and RR is the time slice in RR, which makes

Round Robin a preemptive algorithm. Between the two, FCFS would most likely be more efficient because the preemption of RR causes more context switches especially in the case of longer CPU burst times. More context switches lead to more inefficiency on the part of the RR algorithm. In conclusion, FCFS would be the best algorithm for CPU-bound processes with RR as a close second. This result is corroborated by the data presented in Figure 1 which shows the efficiency of FCFS in comparison to other algorithms.

The Shortest Remaining Time algorithm would be the best for I/O processes as the main issue with SRT is that it can lead to starvation. Starvation occurs when a CPU bound process with a high CPU time is pushed to the end of the queue. However, with I/O-bound processes SRT would avoid starvation as processes spend more of their time in the I/O subsystem. This means fewer processes are competing for CPU usage, which leads to less preemption in the CPU and thus less starvation of processes. This is supported by Figure 1 which shows the least waiting times for SRT with fewer than 5 processes to complete.

2. Based on our simulation results having rr_add set to END is “better” as the results show lesser wait times and turnaround times for the same simulations with the only difference in rr_add. Figures 3 and 4 are based on the data from running the program with a constant seed of 2, lambda of 0.01, context-switch time of 4 milliseconds, alpha of 0.5, a random limit of 256, and a slice time of 128 milliseconds. The various data points are from running various different numbers of processes.

From those two graphs, it is clear that when rr_add is set to END the wait times are consistently lower than those of rr_add set to BEGINNING. Based on Figure 3, setting rr_add to

END consistently outperforms the same simulation with rr_add set to BEGINNING in terms of the turnaround time. These results prove that changing rr_add to END makes RR more efficient.

3. An alpha value of 0.1 or 0.2 produces the best results for the SJF and SRT Algorithms. Those values result in the least waiting time and turnaround time in comparison to other values of alpha across various tests we ran. Figure 8 shows the different values of the waiting time for SJF when run with a constant seed of 2, lambda of 0.01, context-switch time of 4 milliseconds, a random limit of 256, and a slice time of 128 milliseconds. The figure depicts the different results for a changing number of processes and various values of alpha. The trend shows that lower values of alpha provide better results. This is consistent with Figure 7, Figure 9, and Figure 10, which all show that lesser values of alpha lead to lower waiting times and turnaround times. A possible explanation may be due to the scheduling and preemptions of SRT and SJF. A lower alpha value ensures that the next CPU prediction is based more on its previous prediction and less on the actual CPU burst of the process itself. Larger processes won't necessarily be held to the far back of the queue and starve as their predictions are more lenient.

4. From our data, it is clear that changing from SJF, a non-preemptive algorithm, to SRT, a preemptive algorithm, was more efficient given fewer processes. As shown in Figure 5, the wait time for SRT was smaller until the number of processes exceeded 5. Once there were more than 5 processes in the simulation, SJF was more efficient and had a lesser wait time. This is also reflected in the turnaround time for both algorithms, depicted in Figure 6. For fewer processes, the preemptive algorithm, SRT was more effective while SJF had a lesser turnaround time as the

number of processes increased in comparison to SRT. These results are most likely because having more processes in the simulation leads to too many context switches to justify the usage of preemptions, which leads to longer elapsed time and less efficiency for SRT.

5. One limitation of our simulation as compared to a real-world operating system is that a real-world system might not have as many fixed parameters as we take in from the user input, such as alpha, tau, lambda, and other given variables. Another limitation is that the context switch times for our simulation is specified to be an even integer, which may not be the case for various operating systems. Lastly, our simulation is limited to a maximum of 26 processes based on the alphabet. A real-world operating system would handle many more processes than the minimal 26 we simulated through this project.

6. Our own design for a priority schedule would be to incorporate the idea of the SRT and SJF algorithms with priority based on a CPU burst time estimation in addition to priority based on the amount of time the given process would spend outside of the queue. A major drawback of using SRT and SJF is that both algorithms lead to starvation. Starvation occurs in SRT and SJF as they sort the processes with higher tau values to the end of the queue and hence those processes take much longer to reach the CPU to complete their CPU bursts. This is apparent in Figure 1, where the average wait times of the algorithms over a number of tests are shown. The wait times of SJF and SRT are consistently longer than those of FCFS and RR, proving that processes, in general, take longer to reach the CPU for these algorithms. Our design would mitigate this issue by also taking into account the amount of time the process spent outside of the

ready queue. With this addition to the algorithm, the processes with smaller times have less priority so processes that would previously be stuck at the end of the queue would have the ability to be further up in the queue and thus enter the CPU burst faster. The priority can be recalculated at each specified interval denoted as t in milliseconds. For example, given a value of 100 for t, the priorities would be recalculated every 100 milliseconds. The priority could be calculated at the end of each interval by first subtracting the read queue previous exit time from the current time and then subtracting that value from tau.

Some disadvantages to our scheduling algorithm would be that situations can occur where processes with smaller burst times are waiting for a process in the CPU with a larger burst time since the priority is not only based on the estimated CPU burst times. Also, recalculation would need to happen more often, so the algorithm could be slower because of that.

Despite these drawbacks, our algorithm still favors less CPU burst times and implements a priority scheduling algorithm that helps avoid starvation, which are considerable advantages when compared to the other scheduling algorithms.

Figure 1

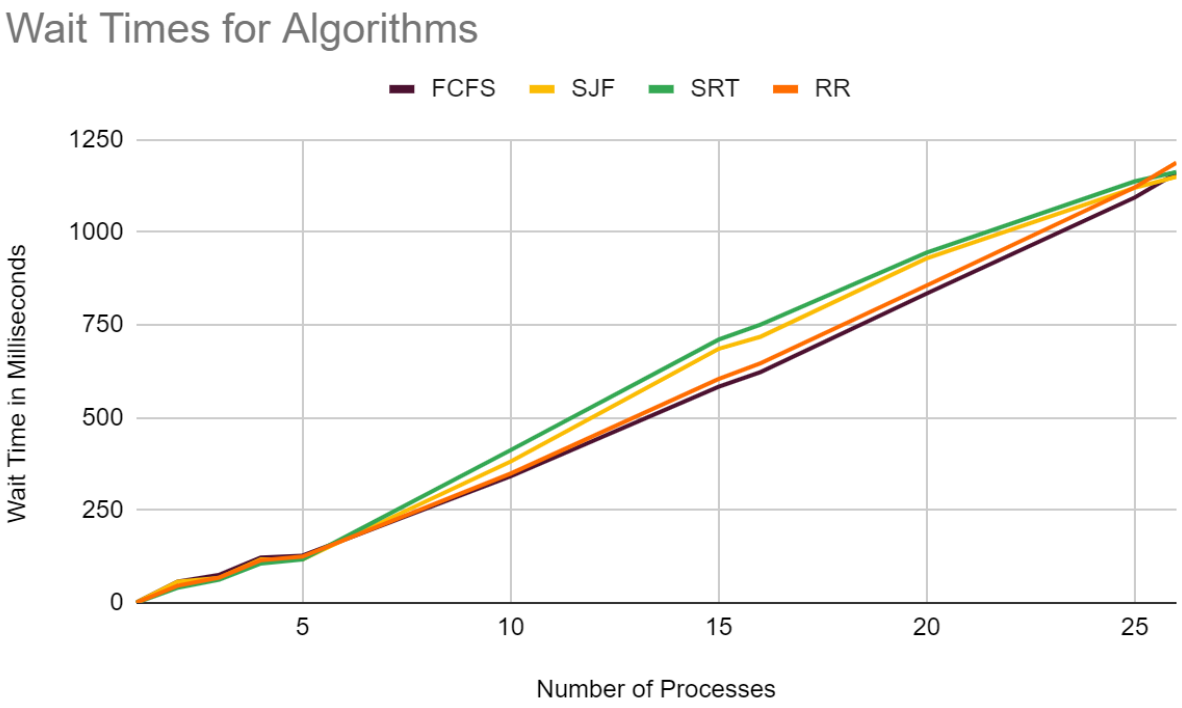

Figure 2

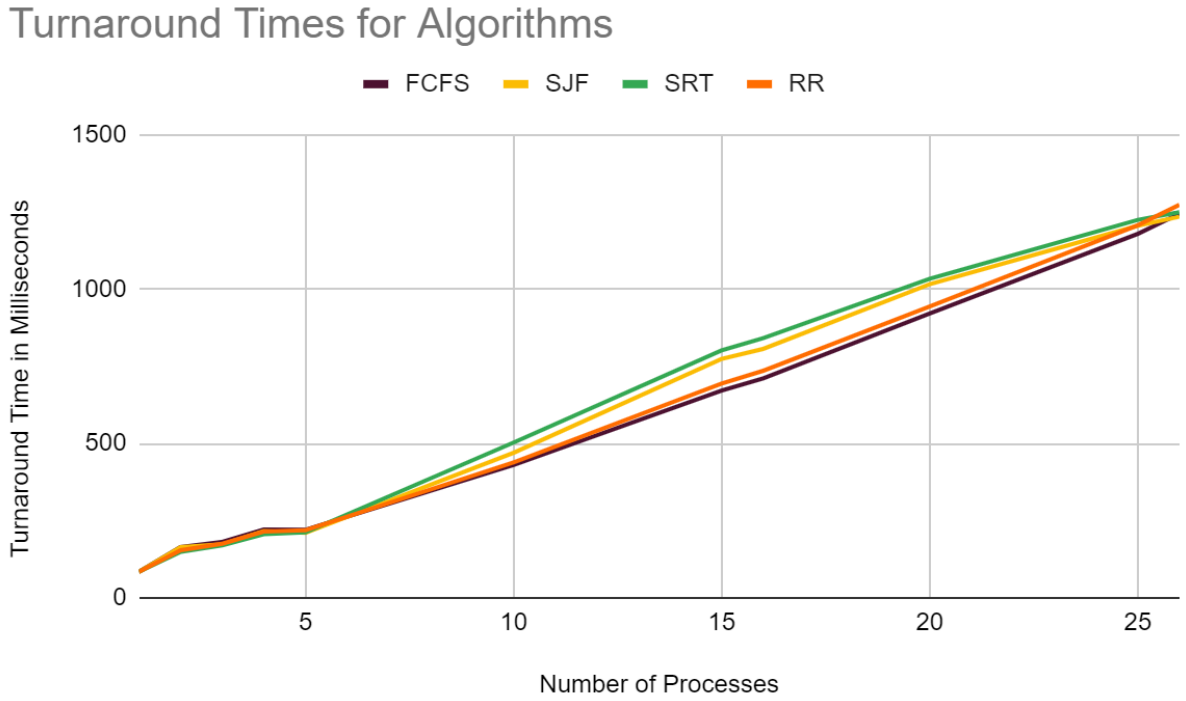

Figure 3

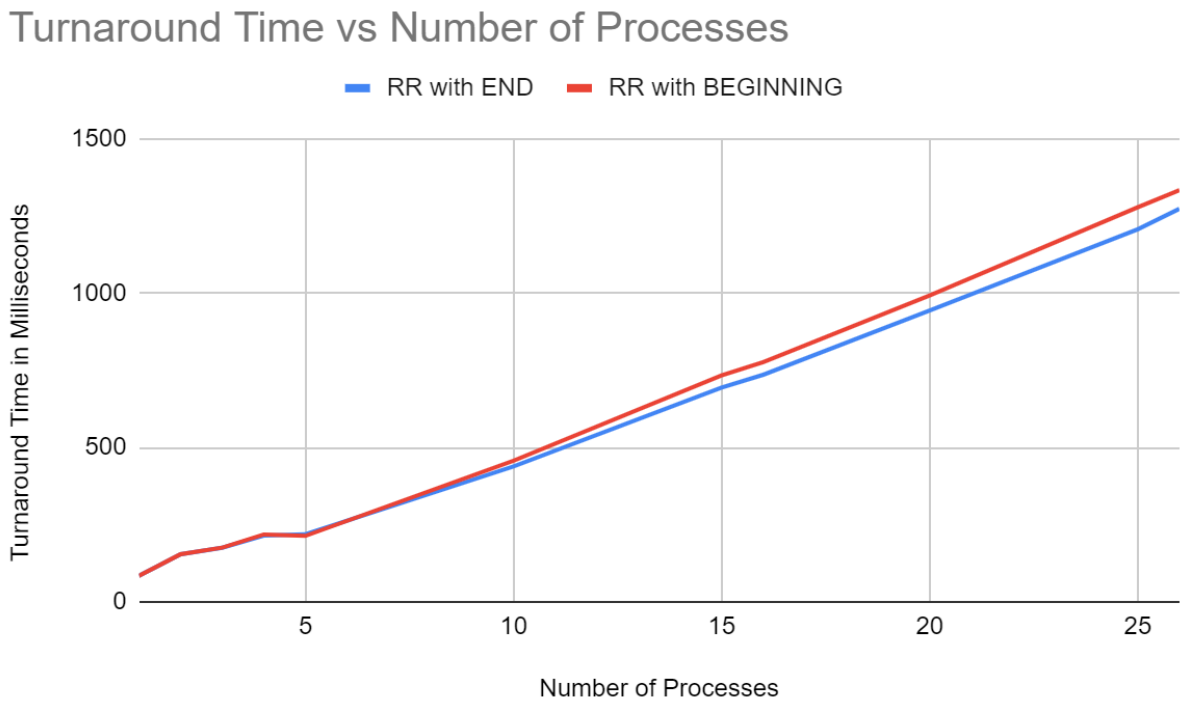

Figure 4

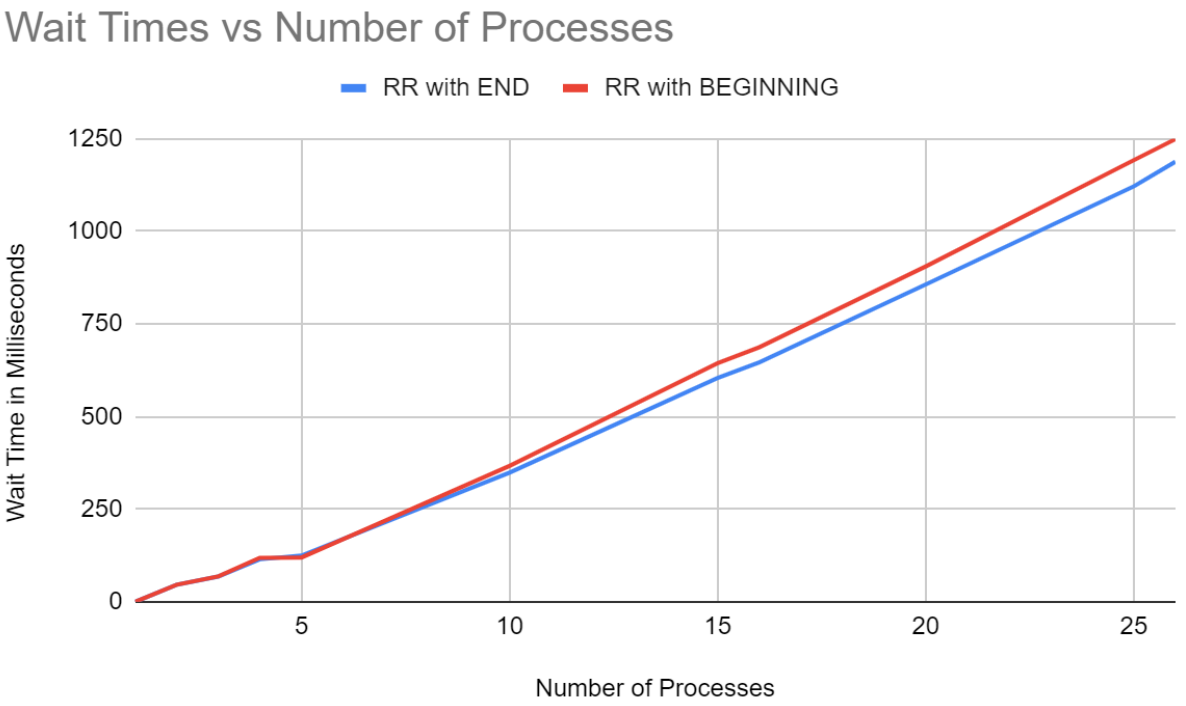

Figure 5

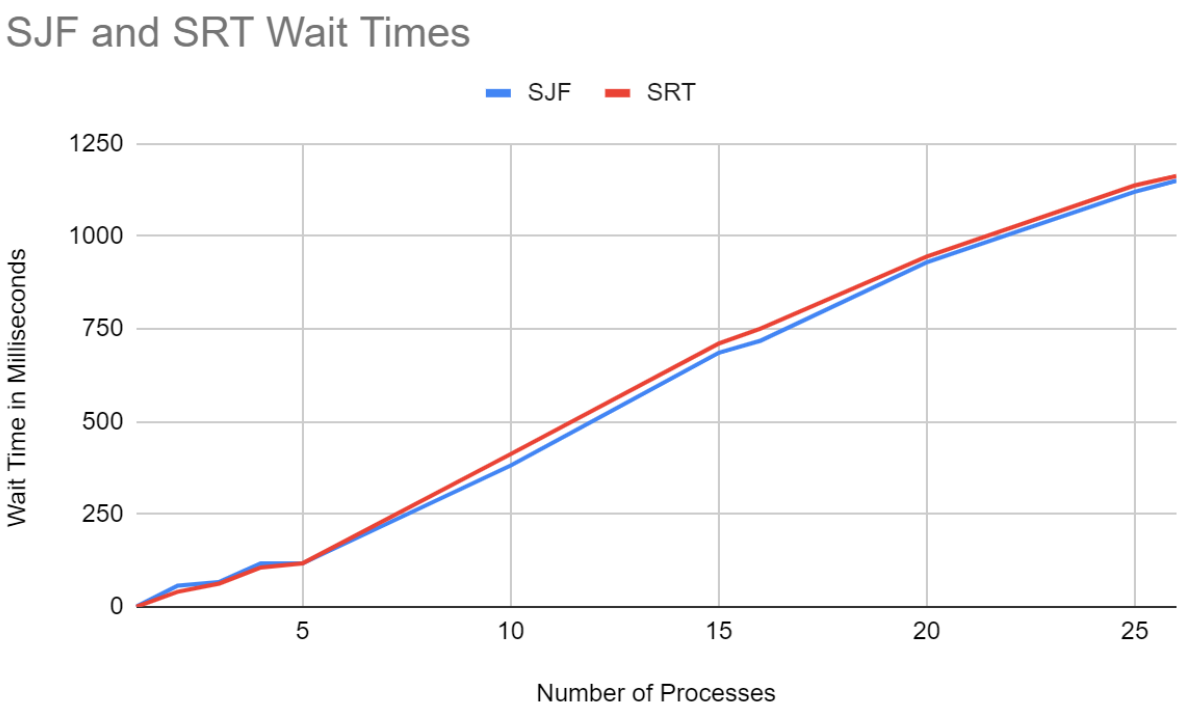

Figure 6

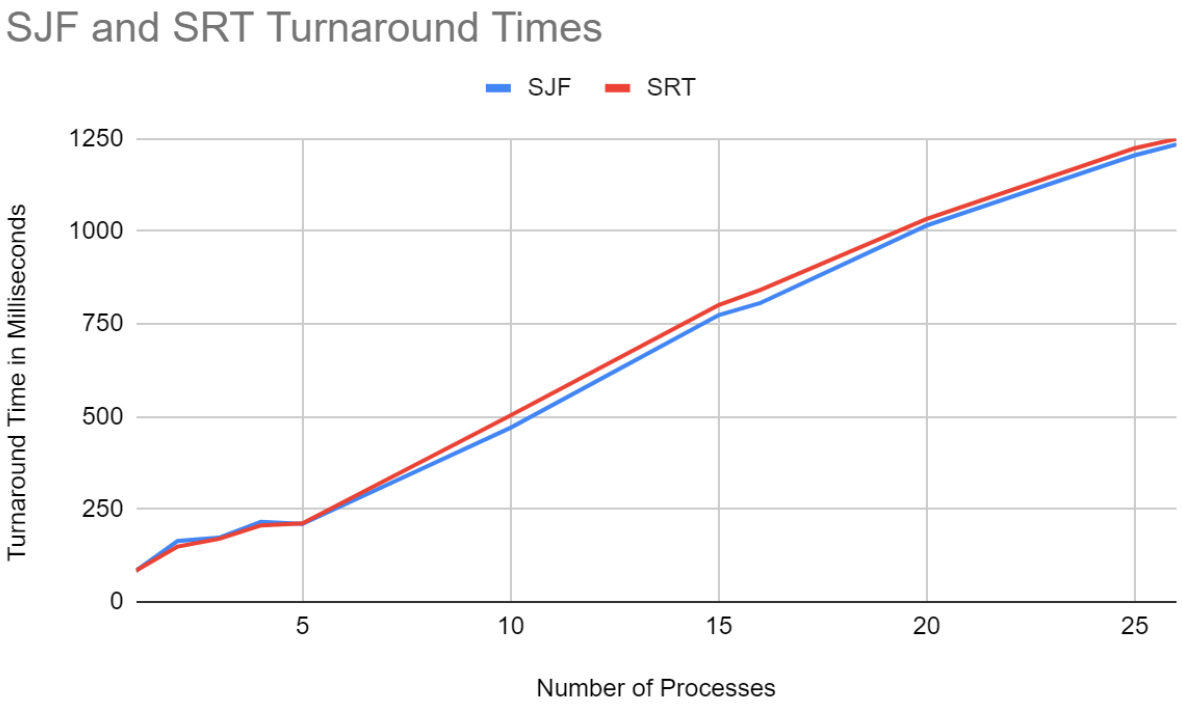

Figure 7

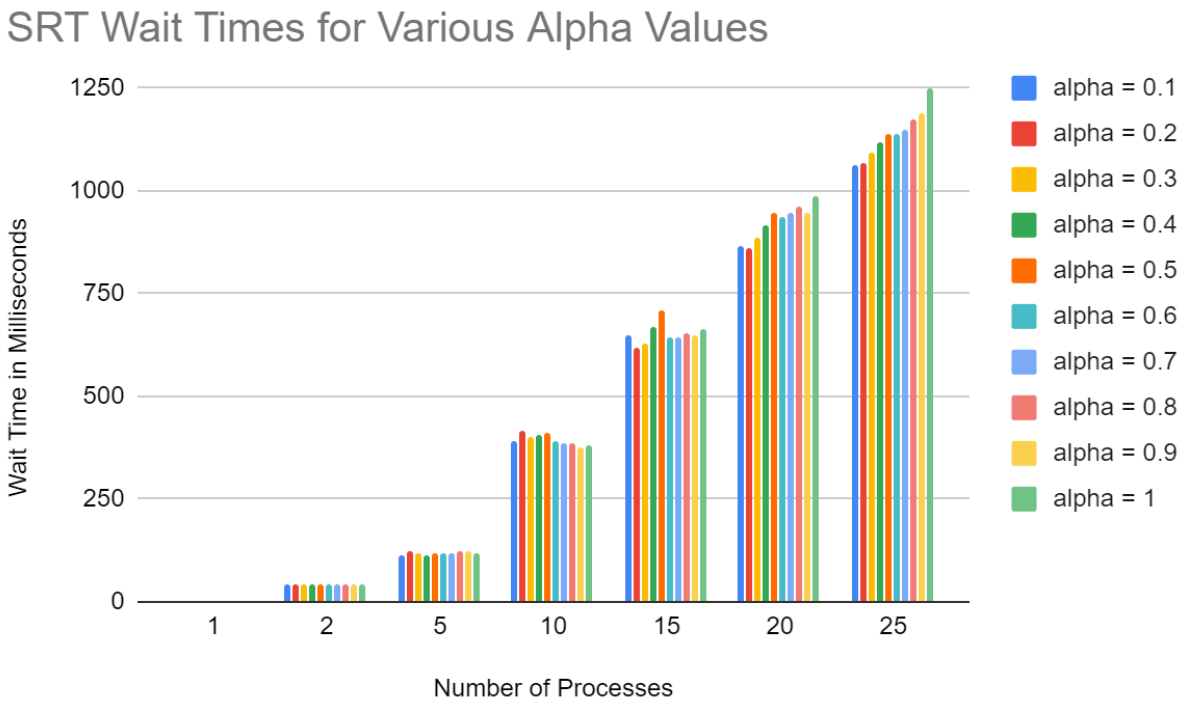

Figure 8

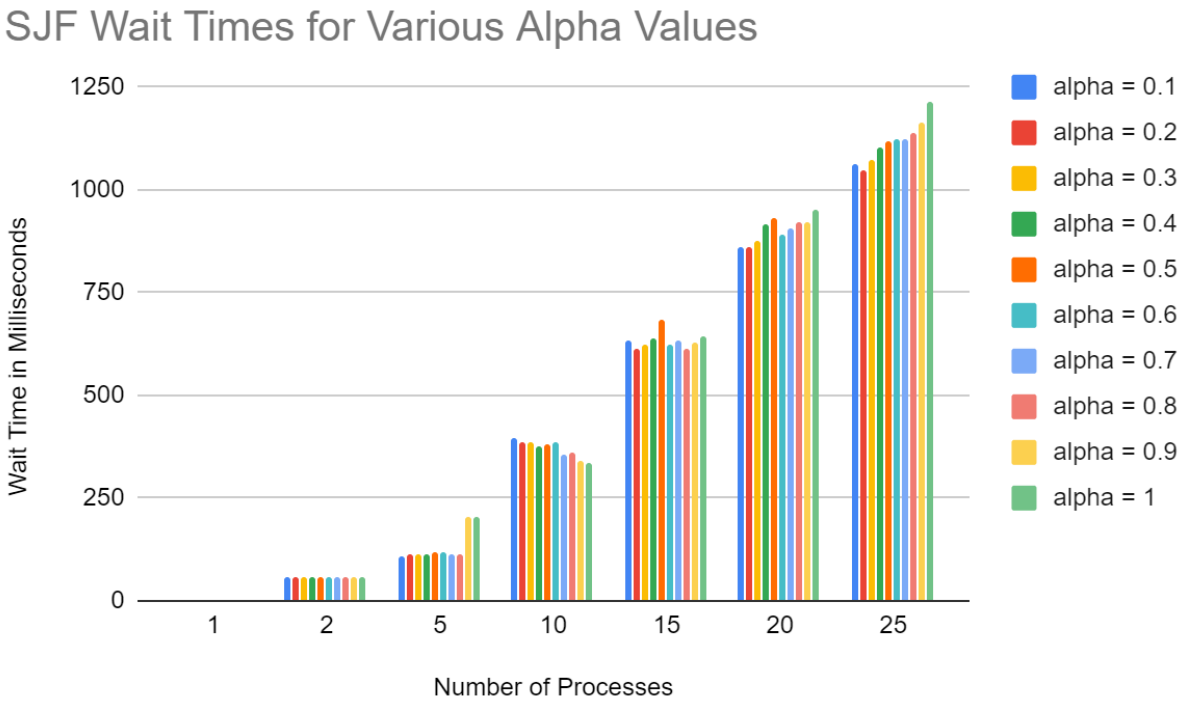

Figure 9

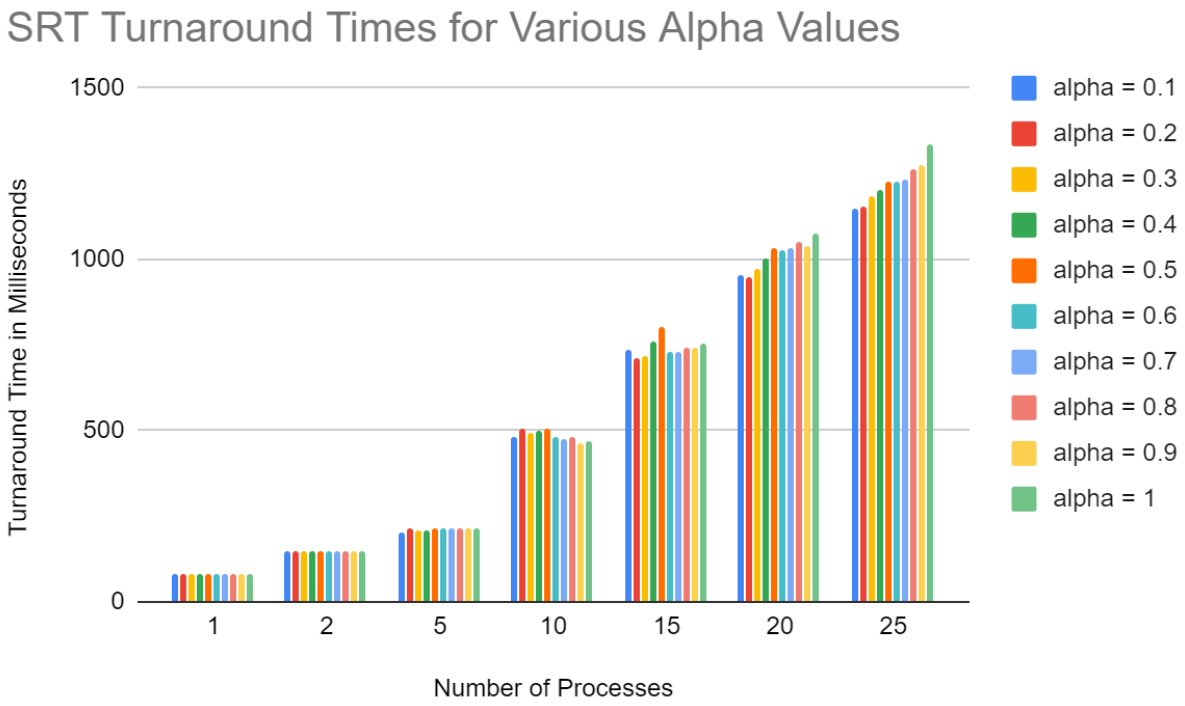

Figure 10