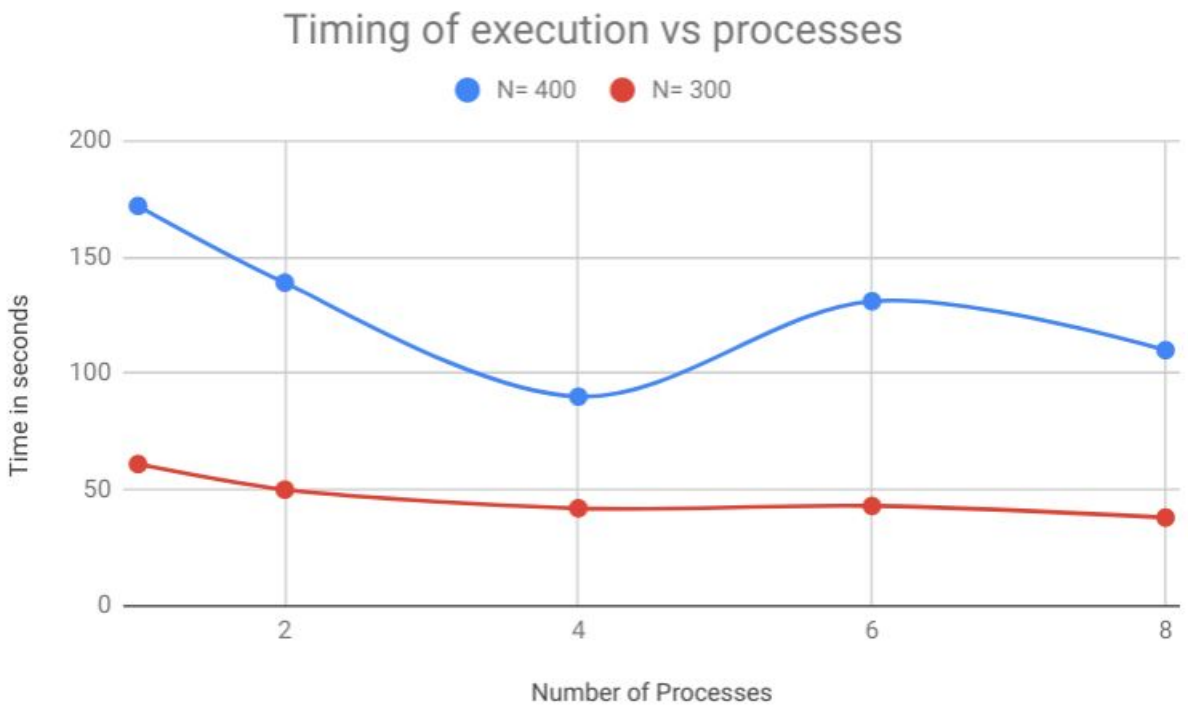
Assignment 2: Operating Systems

Akshat Khare, 2016CS10315

Due date: March 22, 2019, 11:55pm IST

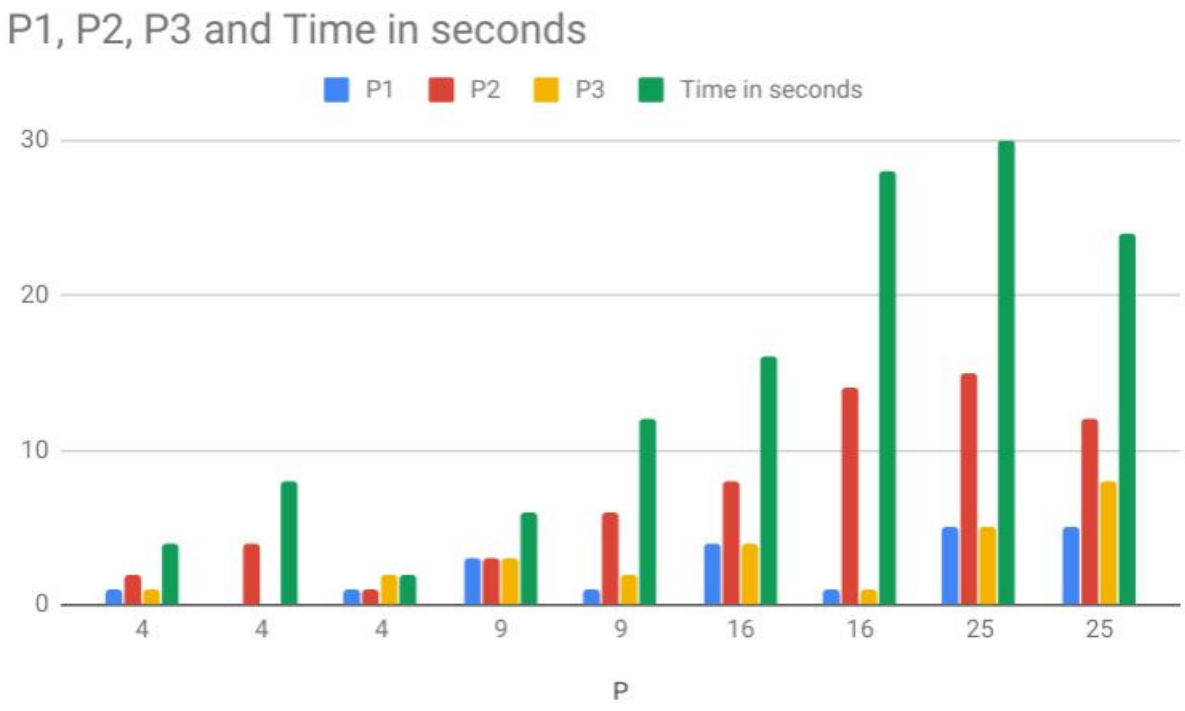
1 Plots and observations

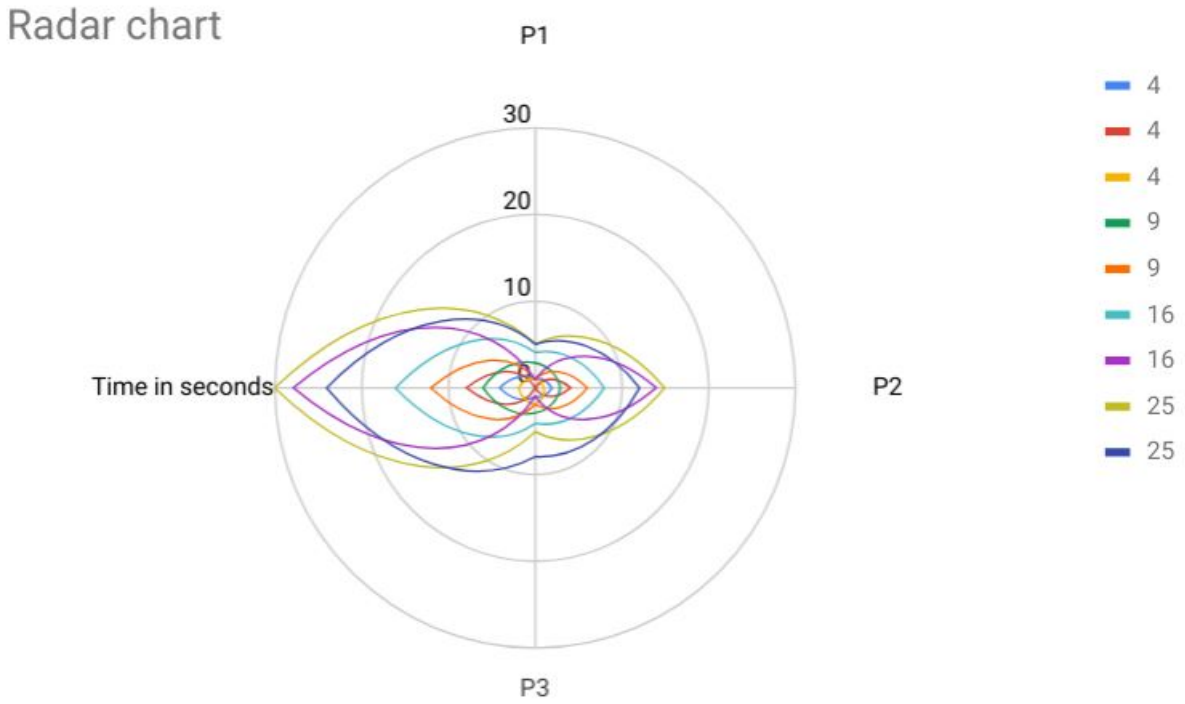
1.1 Jacobi algorithm for steady state heat distribution



Observation and Reason: We see that for $N = 400$ case we see a decrease in time with increase in number of processes. For small cases where $N < 100$ we didn't see much speedup because the time of execution was very less and inter process communication overheads overpowered the benefits of parallel computation. My computer has 4 cores therefore 4 is the optimal number of processes to be spawned for implementation. On increase in number of processes we see that time increased. In this implementation we can see that our implementation is scalable as speedup is evident for large values of N.

1.2 Maekawa algorithm for mutual exclusion





Observation and Reason: We see that time is very independent of $P1$ and $P3$ but time follows the relation $time \approx 2 * P2$, which is true as as only processes of $P2$ kind only contribute to the overall time significantly. From radar chart we can see that Time in seconds and P2 have same profile and in bar char we can see that Time is almost double the P2. In both we charts we can see that Time is independent of P1 and P3.

2 Verifying Correctness

2.1 Jacobi algorithm for steady state heat distribution

For both xv6 and linux following checks were made to ensure correctness:

1. The number of total iterations taken should be same and were found to be same.
2. The final output matrix in both linux and xv6 implementation should be same and were found to be same.
3. For small cases, after each loop the value of $diff$ and matrix should be same and were tested to be same.

2.2 Maekawa algorithm for mutual exclusion

For both xv6 and linux following checks were made to ensure correctness:

1. **Mutual Exclusion**: Each subsequent print of acquired and release lock must not have any other process's print in between. This was tested at least 10 times for 10 different sets of inputs.
2. **Starvation Freedom**: All process who wish to acquire lock must eventually acquire the lock. There should not be a deadlock. This was tested at least 10 times for 10 different sets of inputs.
3. **Fairness**: If there exists a higher priority process that wishes to acquire the lock so a lower priority process must yield the lock upon failing to acquire at lock at that moment, i.e., a higher priority lock must acquire the lock first if it has expressed its intention to do so. This was tested by verifying the order of prints of acquire and release locks at least 10 times for 10 different sets of inputs.