CSE 587 FA2024: Parallel Computing
Assignment 3: Report on OpenMP program

Yichen Lu nechy@umich.edu

October 18, 2024

1 Result

Processes	**Execution Time** (s)	**Max value**
9	62.3388	3.33333
18	31.1786	3.33333
36	17.0395	3.33333

Table 1: The Number of Cores, Execution Time and Maximum Value

2 Description

Based on the methodology provided in the document, I divided the interval into smaller subintervals to parallelize the BFS. Each thread is responsible for computing the maximum within its assigned subinterval, and the number of subintervals is determined based on the number of threads.
I used a the global maximum variable stored in shared memory. A queue is used to store the subinterval values as tasks, and it is initialized with the intervals $[0, \frac{1}{2}]$ and $[\frac{1}{2}, 1]$ at the beginning. I employed **'task_lock'** and **'status_lock'** to ensure that every modification to the shared variables, **'tasks'** and **'status'**, is not affected by concurrent thread operations.
Each thread computes the local maximum within its assigned subinterval, and once all threads complete, the global maximum represents the maximum value of the function over the entire interval.

3 Timing

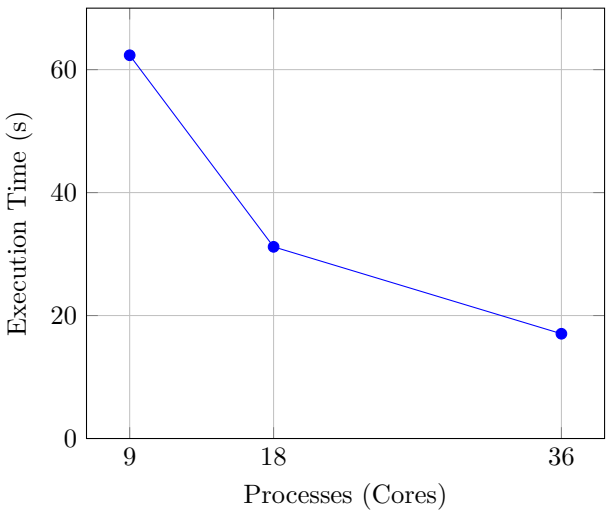

Figure 1: Execution Time vs. Number of Processes

I tested the program on the Great Lakes supercomputer with $np = 9, 18, 36$. The results show that as the number of cores increases, the computation time decreases. However, the performance improvement diminishes beyond a certain point.

This reduction is due to several factors:

1. Increasing the number of cores raises the overhead of locking and unlocking shared resources, particularly when updating the global maximum, which limits performance gains.

2. During execution, the task queue grows as BFS progresses. However, at the beginning, the queue is short, and some threads may be idle, leading to reduced core utilization. The size of the BFS levels also limits the number of available tasks for parallel processing.

3. The speedup from parallelization is inherently limited by the serial portions of the computation. As more threads are utilized, the relative impact of these serial sections becomes more pronounced.

In summary, while increasing the number of cores initially improves performance, the program's scalability is constrained by synchronization overhead and task limitations as more cores are added.