CMPT 770/479: Parallel & Distributed Computing (Fall 2021)

Assignment 4 - Report

Instructions:

- This report is worth 45 points.
- Answer in the space provided. Answers spanning beyond 3 lines (11pt font) will lose points.
- Input graphs used are available at the following location.
  - live-journal graph (LJ graph): **`/scratch/input_graphs/lj`**
- All your answers must be based on the experiments conducted with 4 workers on slow nodes. Answers based on fast nodes and/or different numbers of workers will result in 0 points.
- All the times are in seconds. Unless otherwise stated, use default values for parameters.

---

1. [3 point] Run Triangle Counting with dynamic task mapping on LJ graph. Update the thread statistics in the table below.

Triangle Counting on LJ: Total time = 18.58605 seconds.

thread_id	num_vertices	num_edges	triangle_count	time_taken
0	1218551	17265287	170654322	18.585241
1	1204183	17203720	170335771	18.585213
2	1221300	17375060	171866686	18.585161
3	1203537	17149706	170177107	18.585009

2. [10 points] Run PageRank with dynamic task mapping on LJ graph. Update the thread statistics in the table below. What is your observation on barrier times compared to the barrier times in question 5 from Assignment 3 report? What is your observation on the time taken by each thread compared to time taken by each thread in question 5 from Assignment 3 report? Why are they same/different?

Answer:

The time taken by both of the barriers is reduced and there is no longer a skewness in the task distribution among threads. Both of the barrier take almost no time. Similarly, the total time taken by the threads is also almost identical, that is because thread gets assigned the task as soon as it is free.

PageRank on LJ: Total time = 69.094287 seconds. Partitioning time = 23.848571 seconds.

thread_id	num_vertices	num_edges	barrier1_time	barrier2_time	time_taken
0	24200725	345092859	0.004671	0.004125	69.093459
1	24193332	344381674	0.004671	0.004165	69.093425
2	24305859	345447905	0.004671	0.004256	69.093403
3	24251504	344953022	0.004671	0.004210	69.093330

3. [10 points] Run PageRank with dynamic task mapping on LJ graph. Obtain the total time spent by each thread in `getNextVertexToBeProcessed()` and update the table below. What is your observation on the time taken by `getNextVertexToBeProcessed()`? Why is it high/low?

Answer:

The statistics for this question show us that the time taken waiting for the next vertex is quite a lot (it takes most of the time). This is because the function getNextVertexToBeProcessed() is called too many times, and the thread is basically waiting on the task distribution. This should be changed.

PageRank on LJ: Total time = 69.094287 seconds.

thread_id	num_vertices	num_edges	barrier1_time	barrier2_time	getNextVertex_time	time_taken
0	24200725	345092859	0.004671	0.004125	23.848571	69.093459
1	24193332	344381674	0.004671	0.004165	23.848571	69.093425
2	24305859	345447905	0.004671	0.004256	23.848571	69.093403
3	24251504	344953022	0.004671	0.004210	23.848571	69.093330

4. [10 points] Run PageRank with dynamic task mapping on LJ graph with **`--granularity=2000`**. Update the thread statistics in the table below. What is your observation on the time taken by **`getNextVertexToBeProcessed()`**? Why is it high/low?

Answer:

After introducing the concept of granularity, the time is reduced to one third of previous run. This is because the thread does not have to wait to be assigned a task (for 2000 vertices). This makes sure there is a smaller number of concurrent calls to get the next vertex (vertex1 and vertex2 in my code).

PageRank on LJ: Granularity = 2000. Total time = 25.272375 seconds. Partitioning time = 0.016358 seconds.

thread_id	num_vertices	num_edges	barrier1_time	barrier2_time	getNextVertex_time	time_taken
0	24259426	344826756	0.007480	0.004403	0.016358	25.271479
1	24257426	345666947	0.007480	0.004290	0.016358	25.271422
2	24247426	345189403	0.007480	0.004503	0.016358	25.271397
3	24187142	344192354	0.007480	0.004559	0.016358	25.258626

5. [12 points] While dynamic task mapping with **`--granularity=2000`** performs best across all of our parallel PageRank attempts, it doesn't give much performance benefits over our serial program (might give worse performance on certain inputs). Why is this the case? How can the parallel solution be improved further to gain more performance benefits over serial PageRank?

   Answer:

   The two for loops in the code process different areas of the main task. The first loop runs on edges and waits for a barrier and then the second loop runs on the vertices and then the thread waits for the second barrier. If the loops weren't edge/vertex dependent, more performance can be achieved.