Project 2: Parallel Programming with OPENMP

Yunfei Liu
School of Computer Science, Shanghai Jiao Tong University, Shanghai, Minhang, 800 Dongchuan RD.
liuyunfei@sjtu.edu.cn

***Abstract*—This is the second project report of final assignments of Course Parallel Computing and Parallel Programming Autumn, 2020.**

I. Monte Carlo Algorithm

A. Method

In this task, we implement Monte Carlo algorithm for approximating the area of unit circle (or the value of π). We uniformly generates M points in the square [0, 1] x [0 1]. Then we count the points which fall in the unit quarter circle, denoted as N. Finally we estimate the area of unit circle based on Eq.(1).

$$area = 4 * \frac{N}{M} \tag{1}$$

The task is to parallelize the approximation algorithm with the help of OpenMP. Our strategy is to divide the point sampling process into several threads, and with the help of ”reduction” function of OPENMP, we can sum all the points which fall in the unit quarter circle in every threads.

B. Experiments

We implement this algorithm with C and OPENMP. We run the code on a i5 8400 CPU with 256G memory space. The experiments are conducted with different numbers of threads. The total points number is set 100000 and the result is averaged over 5 runs. Table I shows the results.

TABLE I
The results of Monte Carlo algorithm.

Thread Num	Average Time (s)	Unit Circle Area
1	0.0706	3.141696
4	0.4392	3.142096
8	0.5697	3.141452
16	0.3819	3.142344
32	0.3204	3.142012

As Table I shows, we increase the number of threads, the algorithm is not accelerated. We think the reason behind this is the operation inside ”for” is simple, and communication between threads costs too much time. Meanwhile, we can see that the estimated unit circle area is basically the same under different thread number.

II. Quick Sort

A. Method

Quicksort is a divide-and-conquer algorithm. We select a ’pivot’ element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. The sub-arrays are then sorted recursively.

For the quick sort algorithm, it is difficult and inefficient to parallelize it, since iteration of the current step depends on iteration of the previous step. Thus, we choose to achieve data-level parallelism. The details are as follows.

- First, we divide original large data array into each thread uniformly. Each thread gets a small part of large data volume.
- Second, each thread apply quick sort algorithm to its small data array.
- Third, we implement merge sort algorithm to sort the arrays from different threads. Since the merge sort of the small data array of two threads does not affect other threads, this step can be also implemented in parallel.

B. Experiments

Same as Monte Carlo task, we also implement this algorithm with C and OPENMP. We run the code on a i5 8400 CPU with 256G memory space. We randomly initial a data arrary with 1,000,000 elements. We change the number of threads, run each experiment for 5 time, and report the average time cost, as Table II shows.

TABLE II
Time cost of quick sort algorithm with 1,000,000 elements.

Thread Num	Average Time (s)
1	11.7408
4	0.9511
8	0.3229
16	0.1368
32	0.0887
64	0.1041

From the above table, we can see that increasing number of threads will will greatly reduce the algorithm running time. When the number of threads reaches some value, continue to increase the number of threads, the running time will not be shortened but will increase. e.g. thread number increases from 32 to 64, which means communication between threads cost most time.

III. PageRank

A. Method

For the PageRank algorithm, suppose we have a directed graph, where nodes represent web pages and edge (i, j) means that web page i links to web page j. Then we can build the adjacent matrix based on the graph, which can also be seen as the state transition probability matrix P. And suppose each web page has a initial page rank value, denote page rank values of all web pages as vector pr. Then the page rank value will execute random walk based on the state transition probability matrix P, until it converges. To avoid rank leak and rank sink, we also should add a small constant to each element of matrix P.

The optimal solution is to calculate the corresponding eigenvector when the eigenvalue is 1. However, the number of web pages is very big, and matrix P is also very big, the computational complexity will be huge. Thus, we use the power iteration method. Details as follows.

- First, execute random walk based on $pr_{new} = P * pr_{old}$, then update the page rank value $pr_{old} = pr_{new}$.
- Second, repeat the above step until the distance between pr_{new} and pr_{old} is less than some threshold or reaching the maximum iteration steps.

To parallelize the algorithm, we parallelize the step of random walk and update step. specifically, we use COO sparse matrix to store the sparse adjacent matrix, which make it difficult to initial this graph in parallel. And the iteration steps exist data dependency. Thus, we implemente parallelization to update the page rank value of each web page in each iteration step with the help of ”parallel for” clause of OPENMP.

B. Experiments

Similarly, we implement this algorithm with C and OPENMP. and run the code on a i5 8400 CPU with 256G memory space. We initialize a graph which contains 1,024,000 nodes randomly and the edge count of different nodes ranges from 1 to 10. The PageRank algorithm will run for 100 iterations. We change the threads number and report the average time cost for 5 repeated experiments. (Note that, we use coo sparse matrix to store the huge adjacent matrix, otherwise, we can not store this matrix on the 256G memory space). The results are shown in Table III.

TABLE III
TIME COST OF PAGERANK ALGORITHM WITH 1,024,000 NODES.

Thread Num	Average Time (s)
1	11.1314
4	7.0535
8	4.5209
16	3.2166
32	2.8717
64	2.9882

From Table III, we can see that we can see that increasing number of threads will will greatly reduce the algorithm running time. When the number of threads reaches some value, continue to increase the number of threads, the running time will not be shortened but will increase. e.g. thread number increases from 32 to 64, which means communication between threads cost most time.

IV. DEMO LINK

The demo for this project is upload to jbox, you can follow the link below and check the running process.

- The demo link:
  https://jbox.sjtu.edu.cn/l/iJFKld.