Multicore Programming Final Project Report

Team member 1: 0010129李冠瑜; Team member 2: 0016040 楊凱盛

BFS:

1. Final result

|  |  |  |  |
| --- | --- | --- | --- |
|  | Original | Modified | Speedup |
| Execution time | 15.136085 s | 5.106758 s | 2.964 |

1. First version (total: 11s, build graph: 9s, BFS: 2s)

In this version, we **use multi threads to run BFS**. In the bfs part, after checking a level of nodes, I'll check if there are more than a certain number (4 or more, depends on the hardware) of nodes in next level. If there are, I will create the corresponding amount of threads to deal with each node in the queue.

1. Second optimization (total: 11s, build graph: 6s, BFS: 5s)

In this version, we **rebuild the data structure** of the graph as an array and run with only one thread. In the first version, most of the time is spent on building the graph. Therefore, we thought the time could be reduced by changing the data structure of the graph. We have tried to store the graph in array and vector. It turns out that the vector in C++ is not as efficient as the array.

We use array to store each node's neighbors.

* + First, we had to find out the amount of nodes and maximum amount of a node's neighbors.
  + Second, we could build a 2D array, which looks like Array[# of nodes + 1][maximum # of neighbors + 1]. Array[i][0] is used to store the number of neighbors of the node i.
  + For example, if node 532 has four neighbors: 533, 534, 535, and 536, row 532 of the array looks like this:

A[532][0] = 4, A[532][1] = 533, A[532][2] = 534, A[532][3] = 535, A[532][4] = 536

Later, we realized that allocating memory for a 2D array takes a lot of time. So we use a 1D array instead.

1. Third version (total: 8s, build graph: 6s, BFS: 2s)

In this version, we combine version 1 with version 2. Although the data structure is different, we still use the same way to distribute the workload of bfs to each thread and get a similar result.

1. Forth version (total: 5s, build graph: 3s, BFS: 2s)

In this version, we **use multi threads to build the graph with new data structure and run bfs**.

There are three steps in building the graph:

* + 1. Find the number of nodes.

We traverse the input arrays (tail and head) to find the biggest node number. It costs 0.05 second and get a similar result when multi-threaded.

* + 1. Get the maximum amount of neighbors a node may have.

We traverse the input array to calculate the largest number of edges of a node. The number is used to construct the array. It costs around 1 second and could be reduced to 0.75 second when multi-threaded.

* + 1. Store the graph in a 1D array.

We traverse the input arrays (tail and head) and store all edge information into a 1D array. It costs around 5 seconds and could be reduced to around 2 seconds when multi-threaded.

Needleman-Wunsh:

1. Final result

|  |  |  |  |
| --- | --- | --- | --- |
|  | Original | Modified | Speedup |
| Execution time | 13.665s | 1.888s | 7.237 |

1. Unsuccessful tries
   1. Pthread:

For an input data size of N, I create N threads to handle N rows. When doing the main computation, you have to compute elements (j-1, k) and (j, k-1) before computing element (j, k). Therefore, I use barriers to make sure that each thread will wait for each other so all elements could be computed in the right order. However, due to the limitation of hardware (e.g. only 4 physical threads per CPU) and the huge overhead caused by barriers, there is not much speedup using pthread. The computing time is almost as same as the original time.

* 1. CUDA:

If using CUDA, we have to face the similar problem mentioned above. I use syncthreads() to guarantee the correct computing order. Since syncthreads() could only synchronize the threads in a same block, I could only deploy one block a time. This limited the number of threads. As a result, the computing power of GPU is not fully utilized. The other problem is that the memory of GPU is not big enough to accommodate the whole N\*N array. The solution is to divide the array in several parts and compute each part in turns. In this way, we have to copy data between the host and the device several times, which is not very efficient. Also, this makes the code much more complicated. Due to the synchronization and memory problems, the result is not very favorable.

1. Final optimizations
   1. Way of traversing the 2D array:

In the original version, the array is traversed diagonally. Since the array in c is stored in a row major way, the most efficient way to traverse an array is traversing row by row. This simple optimization makes a big improvement. The speedup is around 4.

* 1. Cancel the initialization of array:

Since each element will be filled up with the maximum of three neighboring numbers, we do not need any initial value to do the computation. Therefore, we could cancel the initialization of array.

* 1. Combine the computation of reference array with the computation of input\_itemsets:

Since the loop for computing reference array and the loop for computing input\_itemsets array are exactly the same, we could combine these two loops into one. As a result, we only have to traverse the N\*N array for one time. Also, we only have to calculate the index for each element for one time.