CSU33014 Annual PartB Code Explanation

Can Zhou 19234118

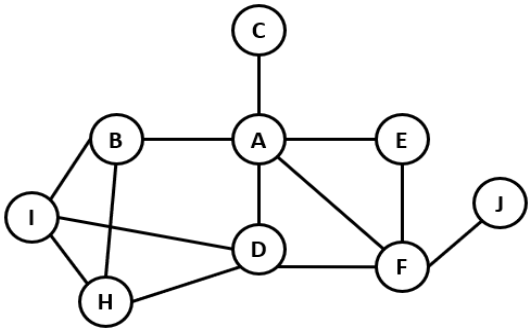


Figure 1 sample graph

Part B.1

The reason why there has redundant work in original code

The recursive function will visit the same nodes of the graph again and again, even though they have already been visited before. For example, by using the graph above, assuming A is start point, C is reached from A, but in C point's recursion, point A will be revisited again which will waste time.

My algorithm for reducing redundant visits

1. The algorithm I used is a modified version of DFS(Depth First Search).
   a. This question is based on an undirected graph and each edge has same weight. So, the degree of separation is the number of edges between two nodes (persons).
   b. I have added a new array *distance*[] to record the number of steps between two nodes. And initializing all the distance to -1 (I am using -1 instead of infinity for initialising the elements of the array. For using infinity as initializer, I need to import additional libraries which I did not feel necessary).
   c. Instead of using traditional DFS to find the goal node, my modified DFS algorithm will terminate when the depth is equal to the degree of septation.
   d. Assuming the start point is A, so the value of point A in distance array is 0. When B will be visited next from A, point B begins to search the points connected. When the B searches A, it can find the value of point A in distance array is 0 not -1 which means point A has been visited, so B will not revisit the node A. In this way, we can reduce redundant visits.
   e. However, we need to consider that DFS perhaps cannot find the minimum distance between two nodes. For instance, assuming the degree of separation is 2 and the start point is A, the point F can be found in this route: A->D->F

and because the distance[F] is 2, J cannot be reached in this route, but after DFS's backtrack, F can be directly reached by A. In this situation, we can update distance[F] to 1, so point J can be reachable under this route: A->F->J.

f. As I mentioned above whether the node is accessed depends on two conditions. One is if the node has not been visited before, then we visit it. Or the new distance between this node and start node is less than the value recorded in the distance array which means this node has potential to reach more nodes.

g. Reason why I have not deleted the bool * reachable in original code which's function can be replaced by int * distance (if distance is -1, then this node hasn't been visited).

   A Bool value only take up 1 bit, an Int takes 64 bits on a 64-bit machine. Bool array (reachable) is much faster than int array (distance) when counting the number of visited node.

2. I have used recursive version instead of iteration version.

   In C programming, iteration should faster than recursion, so first, I tried to implement the algorithm in iteration way by using stack data structure based on linked list. However, I found iteration version slower than recursive version as the result below.

```
Correct count 184464, Time: 187051
Less redundant count (recursion) 184464, Time: 136085
Less redundant count (iteration) 184464, Time: 722381
```

   I think the main explanation is I need to malloc the memory from heap for linked list stack each time which has a high time overhead and recursion allocates the memory from stack which is faster. So, I use the recursive version instead of iteration version.

Time complexity

Assuming the number of total people is n and the degree of separation is k.

Original code's time complexity: $\boldsymbol{O(n^k)}$.

In the worst case, all the nodes are connected to each other, so in k degree, the code will run $\boldsymbol{n^k}$ time.

My less redundant code: $\boldsymbol{O(n^k)}$.

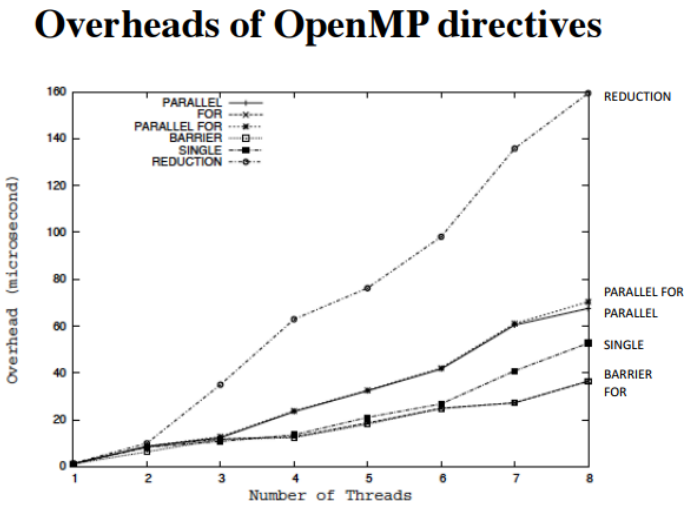
Similar reason as above. However, although less redundant code has same big O time complexity with original code, the constant of $\boldsymbol{n^k}$ is very tiny compared to original code.

Part B.2

The code I used to parallelize is original code. Here is the reason why it is

difficult to parallelize my less redundant code:

In my algorithm, when the recursion is deeper, there will have less nodes need to be visited. However, OpenMP directives have overheads and when the number of threads is increasing, the overhead of OpenMP is increasing as well (see the reference figure below).



(Reference: https://stackoverflow.com/questions/32088211/openmp-slower-than-serial-codes)

When the data set is small, it is not appropriate to use OpenMP as it takes more time to generate and handle more threads which not really do anything. In my less redundant code, the data set is small and OpenMP cannot show its optimal power (overheads of OpenMP directives will cost more than the time that multithread saved). So, I used original code to parallelize instead of my less redundant code.

My parallelization strategy

1. #pragma omp parallel if(total_people > 10000)
   a. #pragma omp parallel

      As I have use parallel for each for loop and recursive function, so I only set one parallel region. Parallel overhead can be reduced slightly by having only one parallel region.
   b. if condition

      Since OpenMP has overheads, when the dataset is small, it is not useful to use OpenMP and may cause longer execution time. Hence, I used an if condition, if the dataset is small (total_people <= 10000), OpenMP won't be used.
2. #pragma omp parallel for

   It can divide the iterations of a for loop between the threads. This has been used in initializing the reachable array and counting the visited people number (when counting, I also use atomic to make sure result is correct and I have mentioned it below).

3. #pragma omp task && #pragma omp single
   Parallel task can be used to parallelize the recursive algorithm which are added to a pool of tasks and executed when the system is ready.
   #pragma omp single is used to make sure only one thread executes the initial recursive function.
4. #pragma omp atomic
   An atomic update can update a variable in a single, unbreakable step. So, with #pragma omp atomic, we are guaranteed that when counting visited people, the value of count is correct.

Time complexity

I used original code and as I have mentioned above, the time complexity for original code is $\boldsymbol{O}(\boldsymbol{n^k})$. If there are P parallel processing cores, in ideally, the parallel code's complexity is $\boldsymbol{O}\left(\frac{\boldsymbol{n^k}}{\boldsymbol{P}}\right)$, where n is the number of total people, k is the degree of separation and p is the number of parallel processing cores.

Timing

When the dataset is small ./graph 500 8

```
Correct count 483, Time: 8011
Less redundant count 483, Time: 186
Parallel count 483, Time: 7833
```

Parallel code has not use OpenMP because of if condition, so it has similar execution time with original code.

When the dataset is large ./graph 800000 10

```
Correct count 714020, Time: 1771759
Less redundant count 714020, Time: 331147
Parallel count 714020, Time: 547692
```

```
Correct count 480932, Time: 520863
Less redundant count 480932, Time: 149322
Parallel count 480932, Time: 391600
```

```
Correct count 584623, Time: 1329039
Less redundant count 584623, Time: 423170
Parallel count 584623, Time: 842479
```

The parallel code is faster than original code.

Note*: sometime parallel code may slower than original code because the random graph connection may cause the dataset which used in OpenMP very small.