Rapport final

Finding Strongly Connected Components

Sequential Algorithm

Question 1:

In this question, we are supposed to find a sequential algorithm to find identifies the strongly connected components of a digraph. Although we don't need to implement a time-optimal algorithm, we still read carefully the purposed article and implement the Tarjan's algorithm.So the time complexity is O(|E|). In order to implement this algorithm, we have built a the graph data structure ourselves by using the map in the standard library. The vertex is represented simply by long int type. We did this by two concerns. The first is that later we are going to treat a greater number of data, and the long int should be able to represent them. And the second is that in the parallel algorithm, we are going to use MPI, however the MPI can't send a complicate data structure, so the long int type can be communicated easily by MPI. But there are also some inconvenient when using this kind of simple data structure. In the Tarjan's algorithm, each vertex has three additional fields. The first is the id, which starts by zero, it may not be identical to the real name of the vertex, imagine the case that the name of the vertex is not a number but a letter. The second is the lowlink, it is a kind of mark in order to tell if we find the root of the strongly connected component or not. The third is “isVisited”, this field has the same functions as the one in the deep search first algorithm, aiming to void the infinite circle. Unfortunately, our simply representation doesn't have these kinds of field, so we have to create a function to transfer our graph into a graph with these specific fields. After finishing this, our program runs quite well.

Question 2:

The is not a difficult question. In order to generate a random graph, it just need to iterate by to loop all the vertex, named i and j, when the random function generate a number smaller than a certain threshold, we don't create the connection, otherwise, we create the connection. The complexity of this algorithm is O(|E|*|E|). And we have tested our program in the firent the question, and we find it works well. During the test, we witness an interesting result. If the threshold is too big, all nodes form a strongly connected component. But if the threshold is not too small to form a non strongly connected component graph, we can have a graph with one cluster of strongly connected components and the rest nodes are left alone. The threshold who can generate a graph with several strongly connected components lives in a really narrow window. We think this is provable mathematically.

Parallel Algorithm

Question 3:

This is a simply extension of the sequential algorithm of generating a random graph. If we have n slavers, and we want to generate a random graphe with p nodes. We just need to attribute n/p node to each slaver, and ask them to generate the vertices between them and all the other node. Just on thing that we should pay attention to is that the initialization of the random seeds should be done differently in each slaver so that they won't all generate the same graphe.

Question 4:

Now we have to parallel our algorithm. The biggest problem is how to divide the graph and attribute them. The article 2 and 3 provide us a really genius idea, which ensure the strongly connected components live each in one of the sub graph without interaction with other subgraph, and more important, one of the subgraph(Succ ∩ Pred) is guaranteed to be a strongly connected component of the original graph. But the problem is that this kind of division may cause a unbalanced charge of each slaver machine, some may be attributed with a great number of nodes while others may have just several isolated nodes. In order to deal with this problem, we want to do this division process recursively which means each slaver machine also divides its graph into four parts and send it to another four free slavers machines. Evidently, the capacity of the Salle info is limited, we have only 169 machines that can be used. So if we treat with a really great data, we have to know how to do this recursive process with a constraint of the number of cores available. However, we haven't figured it out. So we have decided to do a much simple way that we divide the graph into four part, and then the master machine sends the other three subgraph to three slaver machine, and the each slaver launch a Tarjan algorithm, with is the time-optimal algorithm. If we are lucky, we should be able to reduce the time complexity to one forth of its original one. This requires us to find a good pivot node. For now we choose the pivot node randomly. We can hope that choosing the pivot node with the middle node which means it has the median number of neighbors among all the nodes. Because neither choosing the one with the most neighbors nor the one with the least will create a huge imbalance in the charge for each machine. In the worst case, we return to the sequential problem.

If we have some more time, we should be able to figure out how to recursively attribute the data, this will ameliorate largely our algorithm.

Appendix

There are two main classes in our programs (Vertex and Graphe). In the class Graphe, there are two function "randGraphe" and "randGraphe_MPI" to realise the ER digraph sequentially and parallel. The algorithm of Tarjan is written in the file "strongconnect.cpp" and the test program is written in the file "TestQ1Q2.cpp" to test the question 1 and 2. The parallel algorithm to find the strongly connected components is written in the file "parallize.cpp" and the test program is in the file "TestQ4.cpp".

We have already generated the executive files of the test. So you can simply use the commands:

./TestQ1Q2 to test the question 1 and 2;
(If you want to change the size of the ER graph, you open the file "TestQ1Q2.cpp" and you change the arguments in main function, then you regenerate the executive file by
mpic++ Vertex.cpp Graphe.cpp strongconnect.cpp TestQ1Q2.cpp -o TestQ1Q2)

mpirun -np 4 ./TestQ3 to test the question 3;
(To change the size of graph, the same thing as before and then
mpic++ Vertex.cpp Graphe.cpp TestQ3.cpp -o TestQ3)

mpirun -np 4 ./TestQ4 to test the question 4.
(To change the size of graph, the same thing as before and then
mpic++ Vertex.cpp Graphe.cpp strongconnect.cpp parallize.cpp TestQ4.cpp -o TestQ4)