Thibault Chatel

Parallel System – 2020/2021

Distributed programming project

Subject : Minimum spanning tree computation in Kn using MPI

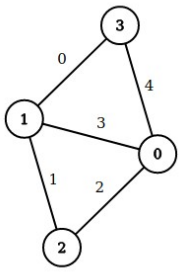
1) Introduction

Minimum spanning tree (MST) computation is a classical operation on weighted graph, for example it is used as a step in the Christofides' algorithm which is an approximation one for the Traveling Salesman Problem.
We will only consider complete graphs Kn as input, which is not restrictive as given any weighted connected graph, adding edges with weight = max of the existing weights +1 so that we obtain Kn, we get the same MST as output.

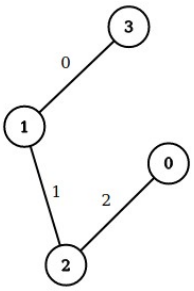
2) The idea

The idea I had was that, Kruskal algorithm has a complexity of O(Elog(E)), so considering complete graph, a time-complexity of $O(N^2 log(N^2))$. We could reduce this complexity by the fact that minimum spanning tree are matroids, meaning objects suited for greedy algorithm so a global solution can be built from local solutions.
Actually, we could split the vertices between our nodes, compute minimum spanning tree of these sub graphs and then consider only the edges of these minimum spanning trees for Kruskal algorithm. That way we would have a complexity of O(Nlog(N)) for Kruskal, the number of edges to be considered would be really reduced (considering we work on complete graphs).

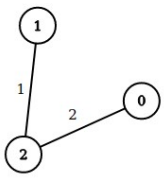
Example :

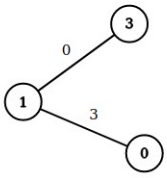


Consider this graph. It's MST is :



Now suppose we split the vertices between 2 nodes :
node 1 in charge of vertices 0,1,2 ; node 2 of vertices 0,1,3. Computing MST on these 2 will give :





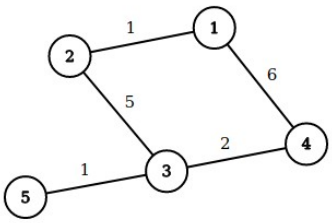
The MST of the whole graph is well contained in the union of the edges of these 2 "sub-MST".

3) Kruskal and Boruvka algorithms

Let's first present Kruskal algorithm.
What we do is first sorting all the edges to obtain the ones of lightest weight in priority. Then, once this is done, we pick edges until we get a spanning tree (of size number of vertices -1), being cautious that we don't pick an edge which would produce a cycle with the ones already selected.

What is Boruvka algorithm now.
The idea here is we first initialize by putting each vertex in its own component. Then we have a loop while we have more than 1 component and in it, at each step we select for each component its lightest incident edge. We add these edges to the MST and we contract the newly linked components into a single one and repeat the loop if needed.

Let us illustrate with an example :



We start with MST={}. For each vertex we look for its lightest edge. What we get with the example graph is :

1→2 ; 2→1 ; 3→5 ; 4→3 ; 5→3

We had so the 3 distinct edges to MST = {12, 35, 34}

After that we contract each edge and we get the second graph on the left. Note that there was 2 edges connecting the 2 nodes, one of weight 5 the other of weight 6, we kept the one of minimal weight corresponding to 23 in the initial graph.

We do the same procedure and we add the edge 23 to the MST = {12, 35, 34, 23}. Contracting the components we get a graph with 1 node so our MST is computed.

4) Boruvka implementation
I saw on this Wikipedia page :
https://en.wikipedia.org/wiki/Parallel_algorithms_for_minimum_spanning_trees
that computing MST using a parallel version of the Boruvka algorithm could be made very efficient in a shared memory programming context. To use this result, I decided to use rayon, and the Rust's part of the course to implement it. Doing so, we get an hybrid algorithm, exploiting both distributed and shared memory programming concepts.
Rayon is based on work-stealing so time complexity is given by : $t_t = W/t + O(D)$; with t the number of threads, W the work and D the depth.
Let's detail the implementation, with N denoting the number of vertices of the input sub graph, and C the number of components :

First we build a slice object from a pointer and a size, corresponding to the input subgraph.

```
let edges: &mut [i32] = unsafe{slice::from_raw_parts_mut(c_array, length as usize)};
```

This costs 1 as a slice in Rust is just a pointer and a length.

Then comes the while loop, stopping when we are left with one component.

```
while composants > 1 {
```

We loop at most log(N) times as C is at least divided by 2 between 2 iterations (which is the case we consider for this computation).

In the loop we have first a search for the lightest edges between distinct components, which is done using a parallel iterator. So, using the above formula, we get a time of $C^2/t+O(\log(C^2))$.

After that we assign new component to the old ones, meaning we build a HashMap, which given the index of an old component, gives the index of the new one. The cost of this part is of $3C/t+O(\log(C))$.

We then do various filter on the edges to only keep the one relevant for the next iteration, like removing edges within the same component etc. The time it takes is :
$C^2/t+O(\log(C^2)) + C + C^2 + C^2/t+O(\log(C)*C)$

Finally we update the HashMap,

```
for k in 0..size {
    belongs_to.insert(k,concordance[&pred[&belongs_to[&k]]]);
}
```

telling us to which component each vertex belongs to, which takes N (we could use here a concurrent HashMap so that we would get $N/t + O(\log(N))$).

Considering that C is divided by 2 at each iteration (which is the worst case), and noticing that in every timing due to work-steal use, the W term is always dominant, we get a sum for the whole while loop :

$$\sum_{i=0}^{\frac{\log(N)}{\log(2)}-1} \left(\left(\frac{2}{t}+1\right)\left(\frac{N}{2^i}\right)^2 + \frac{\left(\frac{N}{2^i}\right)^2}{t} + \frac{3N}{2^i t} + N \right)$$

After computation, we will approximate it by $4N^2/t + 4N^2/3 + N\log(N)$.

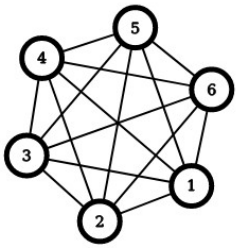
5) Pseudo code of the algorithm
The whole algorithm can be described in few lines :

```
split the original graph between the nodes
each node apply Boruvka algorithm to compute a MST
sort the edges from gather the sub MSTs
apply Kruskal algorithm
```

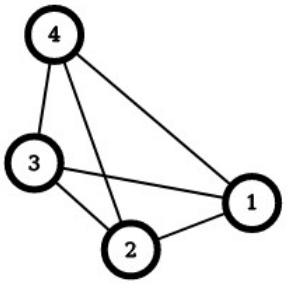
For the correctness of the algorithm : existence of spanning tree ⇔ connectivity, the gather step ensures we get a connected set (assuming we kept in the main processor the edges not within a sub graph) and the spanning trees respecting greedy algorithms, this combination of local solution will give the correct solution.

6) Complexity
We consider complete graphs Kn having so N*(N-1)/2 edges.
Using Kruskal algorithm directly would be of complexity $O(N*(N-1)/2 * \log(N*(N-1)/2))$.
Let us present the complexity with 3 nodes, each node having 4 threads, as it is what I used during my test.

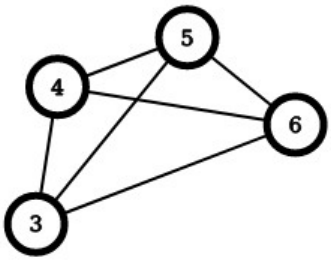
With 3 nodes, in order to have all edges well distributed between the nodes we have to consider sub graphs of 2n/3 vertices. We can illustrate with an example on a graph with 6 vertices, considering sub graphs of 2*6/3 = 4 vertices :



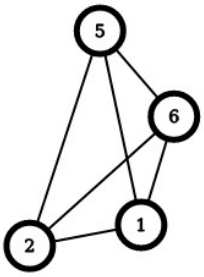
node 1 would get :



node 2 would get :



node 3 would get :



and like that we have well distributed all the edges among our 3 nodes !

I implemented this step of distribution using MPI_Send, MPI_Recv with node 0 being the only one having the whole graph and so in charge to distribute the sub graphs to the other nodes. The cost would be so $3*(2N/3)^2$ (for each node we have to build the sub graph which is a complete one so 3 : number of node ; (2n/3)^2 : size of sub graph) $+ 2*(L+(2N/3)^2/B)$ as node 0 then sends the size of sub graphs to node 1 and 2. A possible improvement could be to use MPI_Scatter for this step.

Once we have the graph distributed we can apply Boruvka algorithm. The implementation I proposed took me a long time to obtain and I am sure we could improve it. Nonetheless, as we have seen, the complexity of this step is $4(2N/3)^2/4 + 4(2N/3)^2/3 + (2N/3)\log(2N/3)$. (reminder : t = 4).

After that we first gather all the sub MSTs. Every node returns a MST made of 2N/3 -1 edges. Gathering them can be done in $2L + 2(2N/3 -1)/B$. Then we have to sort it, I made node 0 use the C function `qsort` for this task. Another improvement would be to explore the possibility of sorting using all the nodes or even simpler, return the MST of the sub graphs in a sorted order to make the sort at node 0 faster. Anyway, I assume `qsort` runs in O(nlog(n)) for an entry of size n so the cost for sorting the edges we get would be $O(3*(2N/3 -1)*\log(3*(2N/3 -1))) = O(2N\log(N))$ as the number of edges remaining is 3*(2N/3 -1).

Finally, applying Kruskal on the sorted edges is linear so $O(3*(2N/3 -1)) = O(2N)$

Considering everything now, we have a t_par(N) =
$3*(2N/3)^2+2*(L+(2N/3)^2/B) + 28N^2/27+(2N/3)\log(2N/3) + 2L+2(2N/3 -1)/B+O(2N\log(N)) + O(2N)$

As we see the most critical part of the algorithm comes from the early distribution of sub graphs. We are also really sensitive to bandwidth given the amount of data we are transferring and that's why, doing my test on my 2 computers like I did and not a supercomputer could be quite bad.

As an indication, the cost on p nodes and t threads with sub graphs of N' vertices is :
$p*N'^2+(p-1)*(L + N'^2/B) + 4N'^2/t+4N^2/3+N'log(N') + 2L+2(N'-1)/B + O(2Nlog(N)) + O(2N)$

An important approximation I made during this complexity computation is that all nodes compute at the same speed, which is not true especially given my configuration during my tests.

7)Tests

I will first describe my setup for the test as it took me a long time to make it work.
I've got 2 computers, both on same WiFi network. One will be called BIG the other SMALL. Both have a user account named "tibo".
First I setup a ssh-server on BIG that I launch with command `sudo service ssh start` (I can verify the result with command `sudo service ssh status`). I can connect via ssh to BIG from SMALL using command `ssh tibo@ip_adress_of_BIG` but it requires a password which is annoying. I use so on SMALL `ssh-keygen` to create a private key id_rsa and a public one id_rsa.pub and then I use the command `ssh-copy-id tibo@ip_adress_of_BIG` to put the public key on BIG. Like that I don't need to input password anymore to connect from SMALL to BIG, I can directly use `ssh ip_adress_of_BIG`.
Now what I want is a "common" directory. To do so I use NFS, I installed `nfs-server` on SMALL and `nfs-client` for BIG, via ssh being on a terminal on SMALL. I created a directory /home/tibo/nfs on both computers. Then, on SMALL I edited /etc/exports adding a line :

/home/tibo/nfs *(rw,sync)

I restart after doing so the NFS server using `sudo service nfs-kernel-server restart` (yet on SMALL).
Now we have to mount this directory to BIG.
I edited, on BIG, /etc/fstab adding a line :

ip_adress_of_SMALL:/home/tibo/nfs /home/tibo/nfs nfs

after that on BIG we do `sudo mount -a`
(it can be useful to define owner of the /home/tibo/nfs directory using `sudo chown tibo /home/tibo/nfs`)
After doing all this, we can see that creating a document in SMALL in the shared directory create the same at same location on BIG!
Then I put my C and Rust code in this directory. The Rust boruvka function being external I modified on both computers the LD_LIBRARY_PATH variable to store the path to rustc/target/debug in my case. After that I compile on SMALL (which will also do it for BIG thanks to NFS) with command :

```
mpicc compute_mst.c -o compute_mst -lrustc -L./rustc/target/debug
```

Finally I run the program using :

```
mpirun -np 3 -H ip_adress_of_SMALL,ip_adress_of_BIG,ip_adress_of_BIG ./compute_mst
```

(thanks a lot to https://www.youtube.com/watch?v=gvR1eQyxS9I)

Here are the timings I managed to get :

Number of nodes	Time for MPI algorithm	Time for sequential Kruskal
6	0.03	<0.01
12	0.93	<0.01
30	4.87	0.02

timings are expressed in seconds

For the timing in MPI, I used 2 MPI_Barrier, one at the very beginning of the main and one at the very end. After each barrier call I used the MPI_Wtime function and like that I get the time of execution doing start-end.

We can see that the performance is quite poor, which could be expected given the fact that my little cluster is clearly not optimized, I should have used ethernet cables, given the bandwidth sensibility we have noticed in the complexity computation.

Another remark is that my run attempt often failed and that’s especially true for graph with more than 50 nodes. I should probably design my code to have better resistance to error, like I would want to be sure that what a node received is what the other node wanted to send as it happened that even for 6 nodes, the function complete but the result is half good, having 3 out of 5 correct edges and 2 incorrect ones in the output. By putting prints in the Rust function, I saw that it happened that some weights data were corrupted, having sometimes a value of 1+the expected value or even random number that I can’t explain. Maybe this is due to the part of unsafe code in this function or it could be that the data is not well send.
I never encountered such behavior running the test on a single computer.