Parallel and Distributed Computing Project Report

Network Slack Calculation

Students:

Bravo, Angel Manuel
Tomsic, Alejandro Zlatko

About Our Solution

For addressing the problem, we divided the task in the following steps:

I. **Reading the input file and creating data structures for storing the read data.**

The algorithm reads the file line by line and stores the information about nodes and edges in our defined structures. In the node structure, we also create two arrays to store information about the node's links. Our first approach was using linked lists to store this information, but after some tests we decided to change this because it increased execution time significantly. The current version uses, as mentioned, two arrays per node which store the indexes of the edges connected from and to itself.

II. **Computing each node's delay** (as described in the project description).

III. **Sorting the graph.**

After trying to solve the problem in different ways, we decided to sort the graph. Using this approach, the problem reduced it's complexity significantly and many computations were saved. The algorithm used for this was "Topological Sorting".

IV. **Computing node's arrival time, required arrival time and slack.**

When calculating arrival times, we take each element respecting the order of the sorted graph and update arrival times of its successors.
When calculating required arrival times, we take each element in the reverse order of the sorted graph and update required arrival times of its predecessors.

V. **Writing output file** (as instructed in the project description).

Parallel Version, OpenMP

We have focused our parallelization only on the computation and therefore not in the reading and writing of the file.

In our parallel version, the node's delay and slack computation have been parallelized by using a parallel for because there are no dependencies among iterations.

At the sorting stage, we parallelized the computation within each node (the inner loop of the Topological Sort algorithm). As the maximum number of incoming and outgoing edges to and from a node is 20, we think the speedup here may vary from one architecture to another. In our case, we used a dual core architecture and detected a little speedup. On architectures where there are more than two processors, we think that no benefit would be obtained from parallelization at this point because the overhead of creating and terminating threads when computing each node would be too high.

For the computation of arrival and required arrival times (from now on AT and RAT respectively) we added an array which stores the amount of items to compute at each level. This array is generated when sorting the graph's elements. The elements at each level of the computation are evenly divided among processes using a parallel for clause. Inside parallel threads we detected that the update operation of node's AT and RAT could not be made simultaneously, so we introduced a critical section for these two operations.

Results

After analyzing the algorithm we realized there is much of the code that can not be parallelized because of data dependencies and communication between tasks. Therefore, we did not expect a high speedup. We got a speedup of 1,22 running on a two processors architecture.

OpenMP new Version

After the first delivery, we realized some improvements could be done in our previous approach and modified it. It mainly addresses two problems we had:

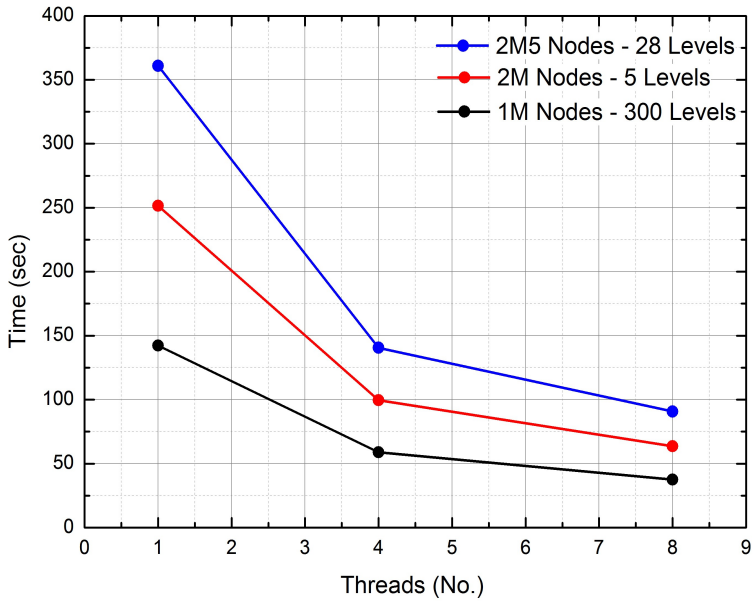
- We were using a critical section for computing AT and RAT at each node because there could be different threads modifying the same values concurrently.
- We were computing RAT after finishing computing AT.

Now:

- We removed the critical section and now only one thread can handle a given node's AT or RAT computation.
- We compute AT and RAT simultaneously in different processors using parallel sections.

Results

With this modifications we got more parallelization and eliminated critical sections. Therefore, we are getting better speed-up. The following picture shows the computation time of our new version of OpenMP using 1, 4 and 8 threads (Quad-Core processor). The table below shows the gotten speed-up.



#Threads/Speed-up	1M Nod. - 300 L	2M Nod. - 5 L	2M5 Nod. - 28 L
4	2,41	2,52	2,56
8	3,78	3,95	3,97

Open MPI Version

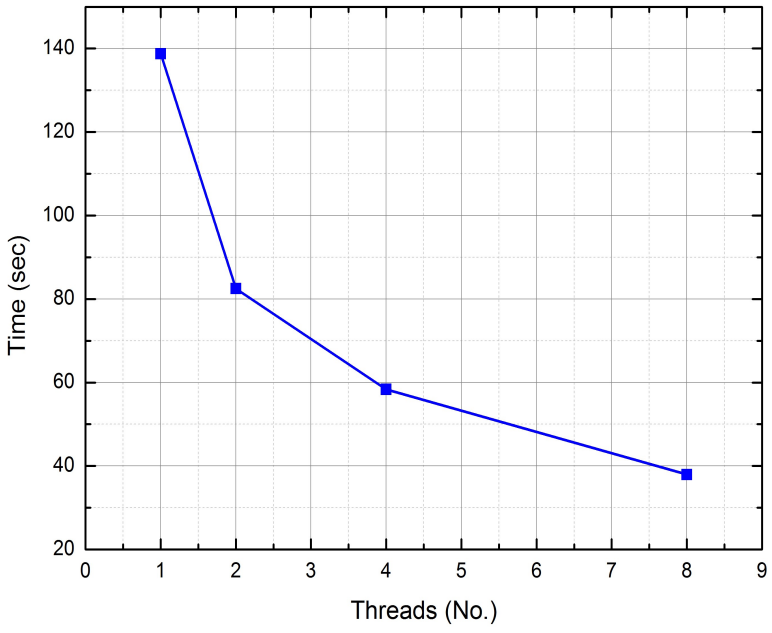
This version is very similar to the last OpenMP version, as the parallelization is mainly the same; we parallelize the computation of each level of the graph, dividing the calculation of a given level evenly among processes, and AT and RAT are calculated simultaneously on different processes.

The master process (id 0) takes care of reading the input file. If there are more than one processes running, it sends all the data structures (nodes and edges) used to represent the graph to process with ID 1. From now on, process 0 handles the computation of AT and process 1 of RAT. If more processes exist, the computation of each level is divided. Process 0 scatters each level's nodes to be computed by processes with even ID and then gathers the results of AT. Process 1 does the same among processes with odd ID. After AT and RAT have been computed, process one sends the RAT results to process 0. the latter receives them and computes slack, and writes the output file.

Results

The results obtained using this approach were not as good as we expected. The main problem this approach has is the initial communication between processes 0 and 1, as we send all the structures of nodes and edges. When computing a graph with a million nodes, this communication takes around 45 seconds on a 100 Mbps fast ethernet connection, time that is very significant. We got speed-up of 1,37 running on 8 computers (1 thread each).

The following picture shows the computation time of our MPI version using 1, 2, 4 and 8 threads in a computer with a Quad-Core processor (2 threads each core).



#Threads	1M Nodes - 300 Levels	Speed-up
1	138,75 sec	
2	82,49 sec	1,68
4	58,369 sec	2,37
8	37,952 sec	3,66

As the picture shows, the obtained speed-up is higher when we are using a unique machine. This results support that the main problem of our approach is the great amount of the initial communication. We suggest some improvements in the next point of this document.

MPI Further Improvements

After several experiments in the lab, we realized we are getting some extra time because of the huge communication between processor zero and one. As we have explained in the MPI description, we are sending the nodes and edges arrays from zero to one if we have two or more processors. This communication takes a high percentage of the overall computation time. We suggest several improvements based on either making this communication lighter or removing it.

- In our approach, the responsible of getting AT and RAT calculated are process zero and one respectively. One option could be making zero the responsible of both. Using this approach we are removing that huge communication. However, zero will have to send a double amount of messages, so that, we believe zero should not compute any AT or RAT, it would be scattering and gathering the data.

- An easy way of getting a better computation time would be passing a smaller amount of data. The processor one does not need all the information stored for each node and for each edge, therefore, we could create a new structure and just store the needed information. Checking in our program we could reduce the communication time in more than a half.

- The last improvement suggested is based on a merge of the others. We suggest to remove the initial send from zero to one but keeping process one as responsible of computing RAT. Processor zero would be in charge of getting the needed levels for computing RAT before it starts with AT calculation. Each time it gets a new level, it will be asynchronously sent to one. In this case we are just sending the smallest possible amount of data. Each time processor one is receiving a level, it will compute it and it will wait for the next level. This receive has to be asynchronous.

We would need to check this improvements experimentally in order to pick one. A priori, the first and the third one are probably better options.