MUHAMMAD AZMAT KAMAL KHAN - 2020/2021 - MSCS (DK) MATRICOLA: 629042

PARALLEL AND DISTRIBUTED SYSTEMS

FINAL PROJECT - DAG

Introduction:

The report consists of the details regarding the parallel pattern followed and the experiments outputs taken from the implementation of the Graph Search problem using Breadth First Search on an Acyclic Graph.

The project description is: "*A graph is described by a set of nodes N (with an associated value) and a set of arcs (oriented pairs of nodes). The application should take a node value X, a starting node S and must return the number of occurrences in the graph of the input node X found in the graph during a bread first parallel search starting from the node S. The graph to be searched is assumed to be acyclic.*"

To solve the problem we have first implemented a sequential version of the problem and then moved to parallelize it using threads and a separate version using FastFlow. The experiments were performed on Intel Xeon PHI machine provided by the university. At the end some conclusion graph is drawn with some data values.

The task here is to implement the parallel approaches using C++ threads, mutexes and another version using FastFlow library to find the occurrences of the input value starting from a given Node.

Files Structure:

The code zip file or github repo contains the code of this project. There are three main CPP files that are SEQ.CPP, PAR.CPP and FF.CPP. The SEQ.CPP contains the sequential version of the problem solution. The PAR.CPP contains the parallel version of the problem solution using C++ threads. The FF.CPP contains the FastFlow version of the problem solution. The rest of the files are being using as the sub-files to these three main files.

Code Details:

In this section we will see the code implementation details and working of the code. So we can start with sequential approach first.

In the sequential approach we need to execute following command to run a program with the params.

```
./seq 0 50 20000 100 400
```

So, the first param is the starting node which in this case is 0, the second param is the value to find, which is 50 in this case. The third param 20000 is the number of nodes, 100 and 400 stating the minimum and the maximum number of edges respectively. The params 20000 100 400 looks for the graph in the graph_data folder with file name graph_data_20000_100_400.txt

The sequential algorithm works as:

We define a vector to check if the node is visited or not. We add the starting node to a NodeQueue and also mark a vector location of the stating node as visited. We then make a while loop and check id the NodeQueue is not empty, and for each in this loop we take pop the NodeQueue and check its value, if matches we increase the count. We also make a sub-loop for the edges of the node item from the NodeQueue earlier. Then for each edges we check if not visited, we push the edge to the NodeQueue and mark it as visited. So the outer loop find the next Node to visit and in same fashion all the nodes are visited and it works as Breadth First Manner. **Ref: SEQ.CPP > BFS function.**

In the parallel approach we need to execute following command to run a program with the params.

```
./par 0 50 64 20000 100 400
```

```
./ff 0 50 64 20000 100 400
```

So, the first param is the starting node which in this case is 0, the second param is the value to find, which is 50 in this case. The third param 64 is the number of workers. The fourth, 20000 is the number of nodes, 100 and 400 stating the minimum and the maximum number of edges respectively. The params 20000 100 400 looks for the graph in the graph_data folder with file name graph_data_20000_100_400.txt

The parallel algorithm works as:

The parallel algorithm work the same way as the sequential one but with slightly different. The two loops, while loop to check is the main Queue is not empty and the inner loop to loop the edges of the current Node, is considered. So, here we implemented two Queues, the current Queue and the Next item Queue. The process works same, with while CurrentQueue is not empty, process the Node and push the edges to NextQueue is not visited. Another thing to consider is to store the value occurrences.

There are two approaches to store the value counts. One is implementing the global variable and protect its value update using mutexes lock and unlock. The issue in this implementation is that when each threads tries to update the value of the occurrences it increases the overhead. So to reduce the overhead we implemented the thread level value counts which counts the occurrences locally and once the thread has finished working we can update the global occurrences at that time resulting less overhead.

Futher this way we can implement the job stealing approach as well where the faster thread steal the work from the slower thread resulting in the faster overall output. So, while the operation is done and the control move to a barrier, we shift the CurrentQueue to NextQueue to move to the next Iteration. At end, all the nodes are visited and system works in Breadth First Manner. **Ref: PAR.CPP > BFS function and FF.CPP > BFS function.**

We can also add –D flag as –DACTIVEWAIT for the Active-wait version of the program. The delay is 5000us.

We can also add –D flag as –DWITHTIME to see the details time each operation within the program took.

Time Calculation:

The algorithm visits each node level and get its edges. Suppose T_{node} is the time cost of visiting the node, T_{edge} is the time cost of vising the edge, T_{sync} is the time cost of overhead generated by synchronization of threads and T_{mut} is the time cost of overhead generated by the mutex. In case of active-wait there is a delay (5ms) at each node which also corresponds in time T_{node}

The cost SEQ T_{total} will:

T_{total} = (Sum of all the T_{node}) + (Sum of all the T_{edge})

The cost PAR T_{total} will:

$T_{total} = T_{sync}$ + [(Sum of all the T_{node}) + (Sum of all the T_{edge}) + T_{mut}] / nw

Experiments:

All the experiments were performed on the Xeon PHI machine provided during the course. The parallel execution were performed using different number of threads like 2, 4, 8, 16, 32, 64, and 128.

We calculated the time of overall computation, the time single loop took, the mutex time, the time to process the edges etc.

The time take by the sequential computation is as follows:

Sequential Input	Avg. time single loop	Avg. time computing edges	Avg. time computing count
10000 Nodes, Edges (min 100, max 400)	33.5758 usec	8.6189 usec	0 usec
20000 Nodes, Edges (min 100, max 400)	34.7905 usec	8.5714 usec	0 usec
20000 Nodes, Edges (min 100, max 800)	46.8824 usec	13.4191 usec	0 usec
30000 Nodes, Edges (min 300, max 800)	53.0348 usec	18.5952 usec	0 usec
50000 Nodes, Edges (min 300, max 1000)	54.9921 usec	21.7422 usec	0 usec

From the above experiment output we can clearly see that as we increase the number of Nodes in the graph the time taken by the single loop increases and same happen while we increase the number of edges the time to compute the edges increases. We also know the single loop contains the internal logic that does many more small things like popping the queue, computing the value count (that is negligible like 0.0000456 usec), also there are operations marking the node as visited. So, from this we can see if we subtract the single loop computation value and the computation time take for processing the edges, we get the computation time for the other operations mentioned.

If we now consider the time take by parallel computation and its details we should have the speedup of ***nw*** using ***nw*** workers. But there is few more things to consider in term of computation time that is time taken by the mutexes locking and unlocking, the time take by the barrier, the synchronization time.

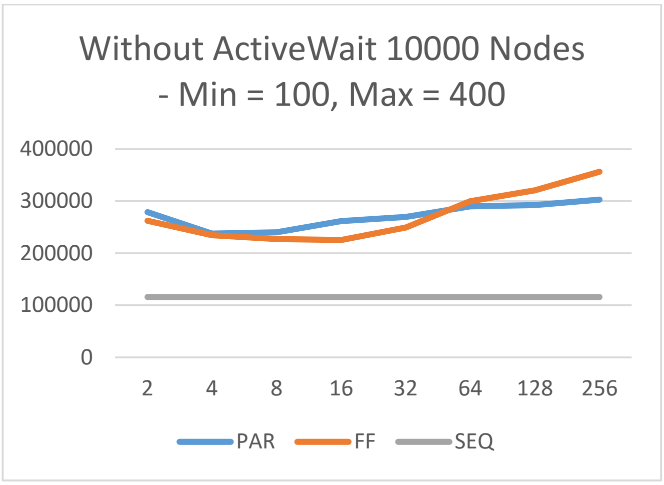
The time taken by the parallel computation is as follows:

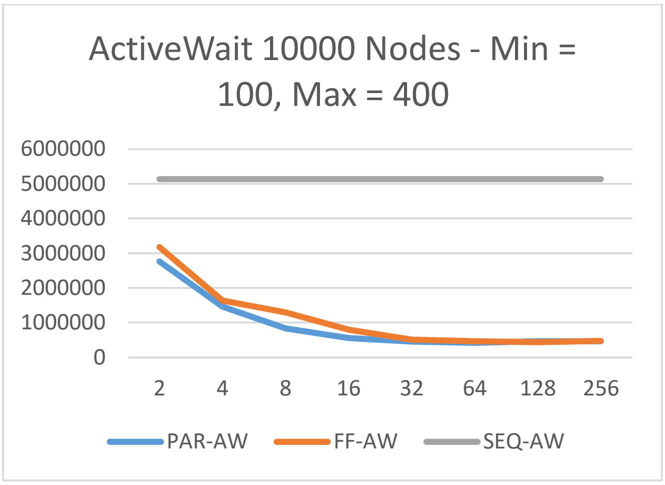
Parallel Input - NW	Avg. mutex time	Avg. time computing edges	Avg. time barrier
10000 Nodes, Edges (min 100, max 400) - 2	3.64944 usec	38.7623 usec	51619.8 usec
10000 Nodes, Edges (min 100, max 400) – 32	30.4582 usec	777.091 usec	48309.5 usec
10000 Nodes, Edges (min 100, max 400) - 64	34.7108 usec	872.666 usec	46149.8 usec
10000 Nodes, Edges (min 100, max 400) - 128	216.036 usec	936.127 usec	30811.9 usec
20000 Nodes, Edges (min 100, max 400) – 2	3.57792 usec	38.7618 usec	111955 usec
20000 Nodes, Edges (min 100, max 400) – 16	12.8104 usec	520.961 usec	97422.5 usec
20000 Nodes, Edges (min 100, max 400) – 32	22.133 usec	975.206 usec	112714 usec
20000 Nodes, Edges (min 100, max 400) – 64	24.2777 usec	1563.39 usec	120722 usec
20000 Nodes, Edges (min 100, max 400) – 128	205.857 usec	1577.32 usec	78028.9 usec
20000 Nodes, Edges (min 100, max 800) – 2	4.9363 usec	45.6347 usec	137291 usec
20000 Nodes, Edges (min 100, max 800) – 16	17.8183 usec	521.004 usec	145079 usec
20000 Nodes, Edges (min 100, max 800) – 32	39.7141 usec	1354.44 usec	162639 usec
20000 Nodes, Edges (min 100, max 800) – 64	61.9289 usec	2276.37 usec	177637 usec
20000 Nodes, Edges (min 100, max 800) – 128	113.243 usec	2329.36 usec	184346 usec
30000 Nodes, Edges (min 300, max 800) - 32	41.3214 usec	1417.76 usec	287215 usec
30000 Nodes, Edges (min 300, max 800) – 128	124.868 usec	3604.76 usec	355166 usec
50000 Nodes, Edges (min 300, max 1000) – 32	37.081 usec	1544.92 usec	547402 usec
50000 Nodes, Edges (min 300, max 1000) - 128	178.081 usec	6393.45 usec	641916 usec

From the above experiment output we can clearly see that as we increase the number of Nodes or Edges, also the number of thread workers, the mutex lock time increases. Since the Edge computation logic also has a mutex lock included where it mark the Node as visited, the computation time also increases as we increase number of Nodes. The barrier time also increases as we increase the number of threads (workers). Instead of decreasing the time for the parallel computation somehow it increases the time due to such locking/unlocking and barrier implementation. So, as the number of threads increases, every thread had to wait for the other threads to finish before the next iteration etc.

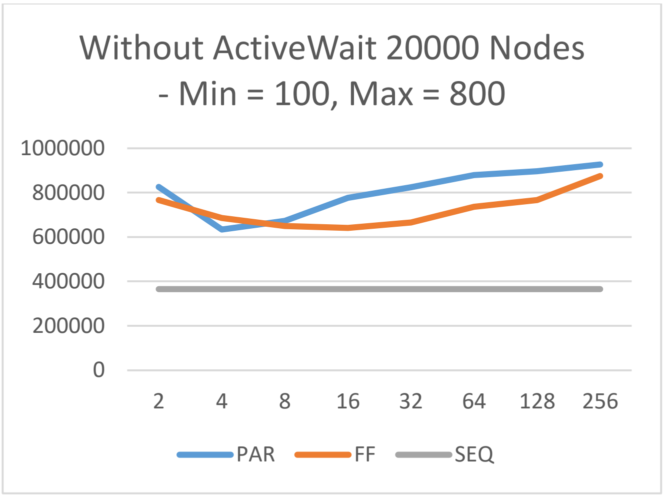
We can now see the graph for the comparison of the sequential computation and the parallel computation.

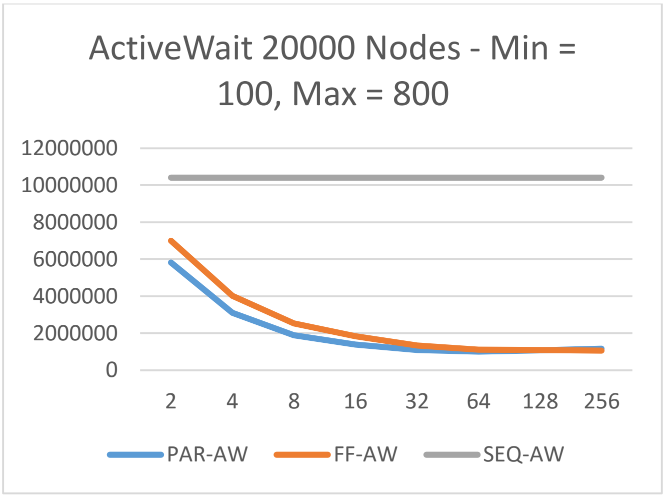
Graph with 10000 nodes with Edges ranging from 100 to 400



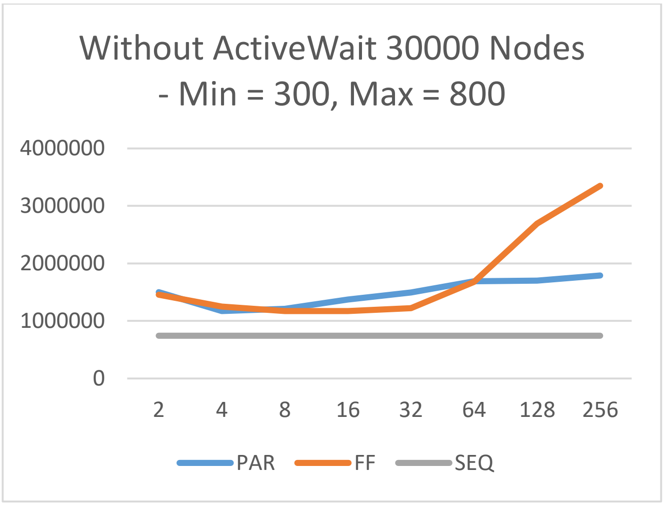


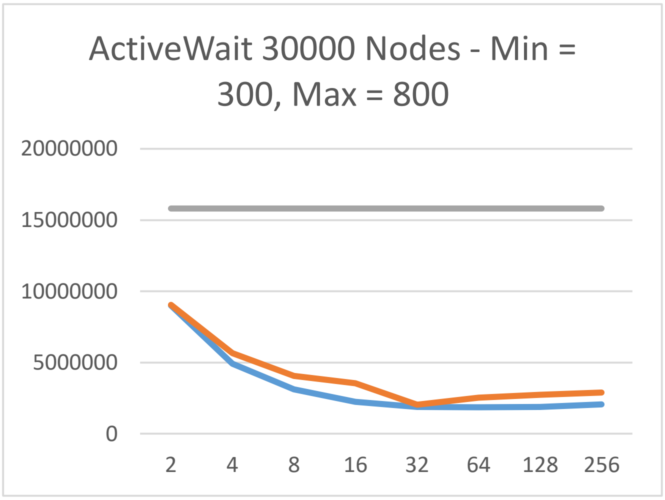
Graph with 20000 nodes with Edges ranging from 100 to 800





Graph with 30000 nodes with Edges ranging from 300 to 800





As we have notice that in without activewait case we have worst performance as we are increasing the number of works, while in the activewait case we can see a good performance and at once stage we have a very good performance (between 64-128 workers) and this is the point, of which we can't achieve more good performance anymore.

As we have the values for the sequential and parallel time, we can calculate the speedup using the formula:

$$Speedup(n) = \frac{Tseq}{Tpar}$$

Conclusion:

The report explains the Breadth First Search algorithm on the graph. We have implemented both sequential and parallel version of the algorithm and noticed the performance and speedup by changing the number of workers. So from the results we can say that it is better to parallelize the algorithm when task given to perform on each node takes more time than the overhead and synchronization cost introduced by using different threads.