PARALLEL AND DISTRIBUTED SYSTEMS

FINAL PROJECT - DAG

Introduction:

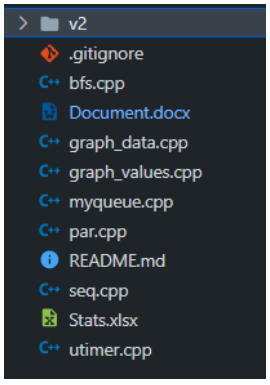
The final project is Graph Search using DAG (Directed-acyclic graph). In DAG each node is represented with some ID and its VALUE. In that, we count the occurrence of a given number at node value in DAG. Multiple nodes can have same or different value. All the nodes are visited in Breadth-first manner.

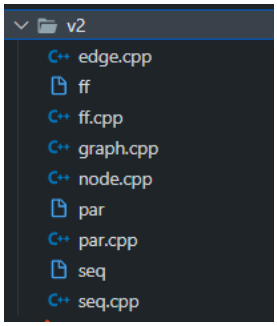
I performed this using programming in C++. Starting with sequential approach and parallelizing it later. The same solution is also done using FastFlow library.

While coding the parallel form, I considered the FARM design pattern with some sort of JOB Stealing mechanism that we studied in the course. A queue is shared with all the threads workers and data is popped and pushed into it for processing and the visited queue take the record of the nodes that have been visited already. This is the overview of the internal working of the system and will be detailed in later sections.

Files Structure:

The code zip file or github repo contains the code of this project. The basic implementation of SEQUENTIAL and PARALLEL is at the root of the folder. The better implementation is in the folder v2 which I ran to evaluate the results, computational time, number of workers used and speedup gained. The folder structure is as follows:





Github repo link: https://github.com/azmatkamal/Directed-Graph-Search-Acyclic

SEQ.CPP contains the **SEQUENTIAL** programming implementation using generic C++ code.

PAR.CPP contains at the root contains the **PARALLEL** programming implementation using generic C++ code.

FF.CPP contains the **FASTFLOW** version of the program implementation in C++.

GRAPH_DATA.CPP contains the nodes and its connected edges data for generating graph.

GRAPH_VALUES.CPP contains the Nodes/Edges value that we need to count the occurrences which is our aim.

GRAPH_DATA.CPP and **GRAPH_VALUES.CPP** is changed between all types of implementations.

Executing Program:

To execute a program we need C++17 for parallel code implementation as we are making some use of atomics. The sequential part can run in standard C++. The FastFlow library in included in the LIB directory of C++ compiler which can be cloned from the following link: https://github.com/fastflow/fastflow

To compile a program there are some instructions written in README.md file at the root of the project files. Here are the details as well:

Compile sequential program: `g++ seq.cpp -o seq`
Run Program: `./seq 0 50`
Compile parallel program: `g++ par.cpp -o par -pthread -std=c++17`
Run Program: `./par 0 50 5`
Compile Fastflow version of a program: `g++ ff.cpp -o ff -pthread -std=c++17`
Run Program: `./ff 0 50 5`

Where 0 is the starting vertex.
Where 50 is the value to find at nodes and get count.
Where 5 is the number of workers.
We can also add –D flag as –DACTIVEWAIT for the Active-wait version of the program. The delay is 5ms.

Program Output:

The output of the program is as follows:

For Sequential:

```
azykamal@Azykamal:/mnt/d/pisa/lectures/practicals/spm/gs-directed-acyclic/v2$ ./seq 0 50
Value to find: 50

50 is found 14 times
SEQ Part computed in 2671 usec
```

For Parallel:

```
azykamal@Azykamal:/mnt/d/pisa/lectures/practicals/spm/gs-directed-acyclic/v2$ ./par 0 50 5
Value to find: 50

50 is found 14 times
PAR Part computed in 3017 usec
```

For FastFlow:

```
azykamal@Azykamal:/mnt/d/pisa/lectures/practicals/spm/gs-directed-acyclic/v2$ ./ff 0 50 5
Starting from: 0
Value to find: 50
No. of Workers: 5

50 is found 14 times
PAR Part computed in 2138 usec
```

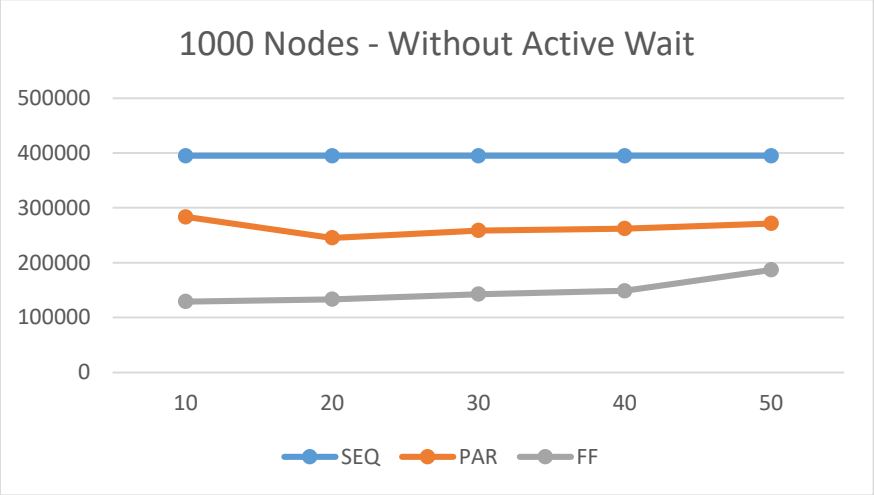
Results:

I executed the programs with different number of workers and different number of Nodes in the graph. The visual graph that shows result of the execution has the time on Y-axis and number of workers on X-axis. The Speedup can be calculated using formula:

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

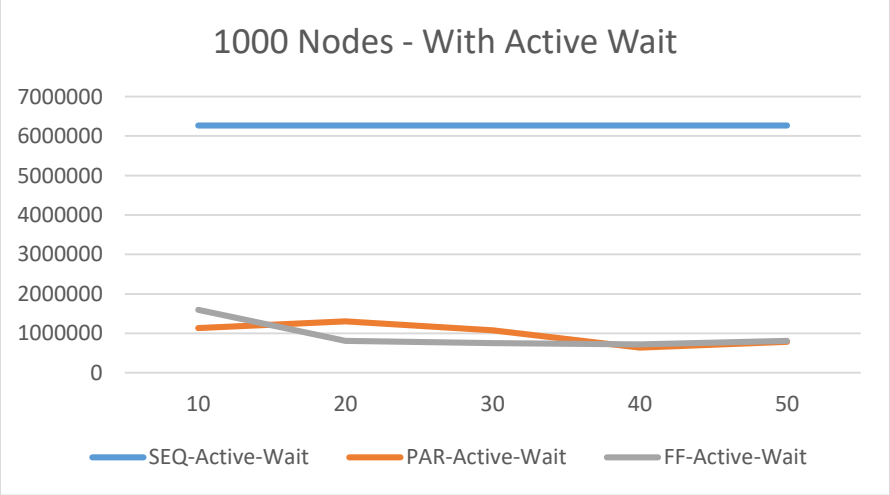
Test 1 – 1000 Nodes without active-wait:

	10	20	30	40	50
SEQ	394974	394974	394974	394974	394974
PAR	283620	245284	258609	262150	271628
FF	129264	133197	142640	148580	187029



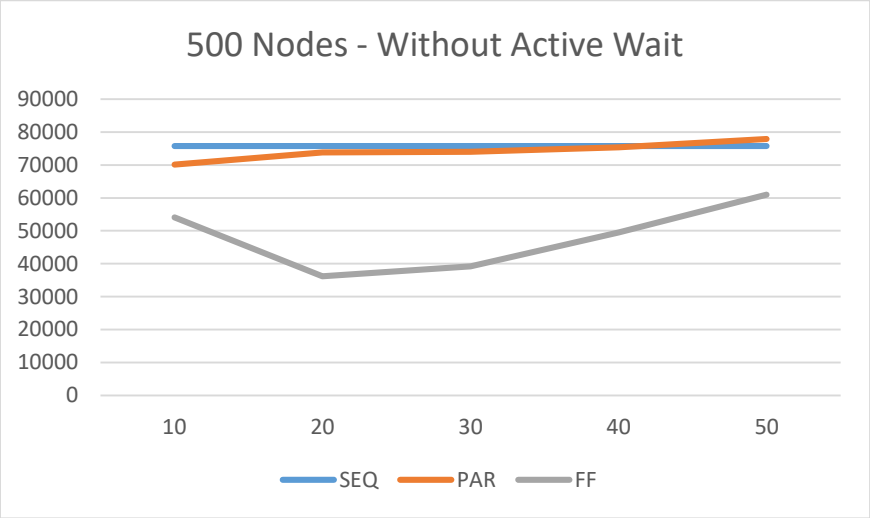
Test 2 – 1000 Nodes with active-wait:

	10	20	30	40	50
SEQ-Active-Wait	6270696	6270696	6270696	6270696	6270696
PAR-Active-Wait	1132782	1304582	1078541	640406	784029
FF-Active-Wait	1599067	807810	757266	723283	811894



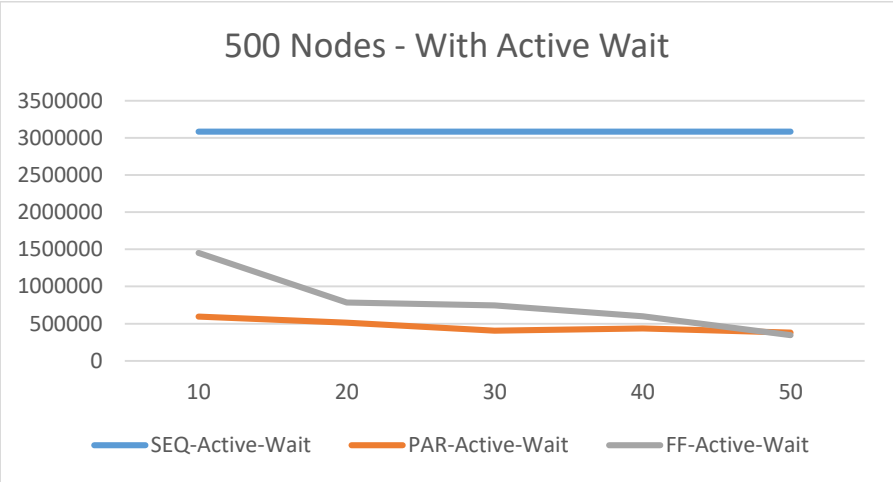
Test 3 – 500 Nodes without active-wait:

	10	20	30	40	50
SEQ	75796	75796	75796	75796	75796
PAR	70126	73802	74042	75340	77899
FF	54076	36207	39248	49477	60963



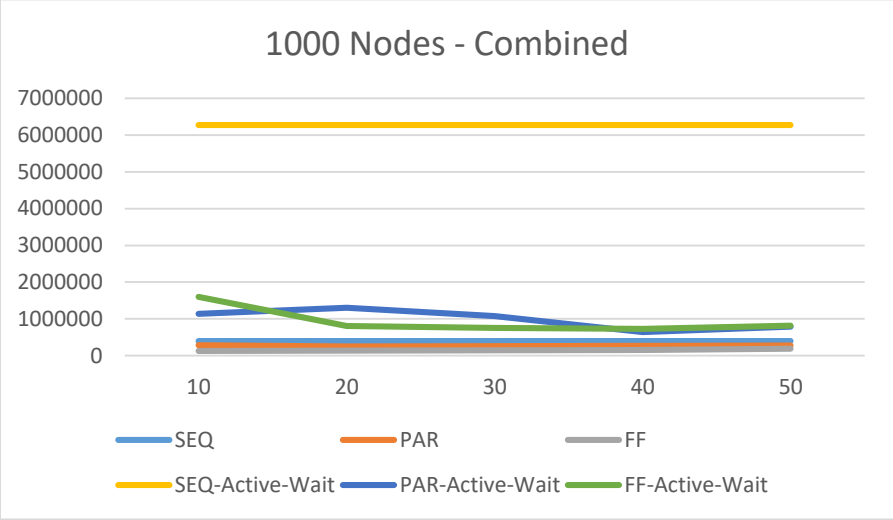
Test 4 – 500 Nodes with active-wait:

	10	20	30	40	50
SEQ-Active-Wait	3081858	3081858	3081858	3081858	3081858
PAR-Active-Wait	596789	512346	407207	435910	381493
FF-Active-Wait	1452768	785872	746563	598458	343752



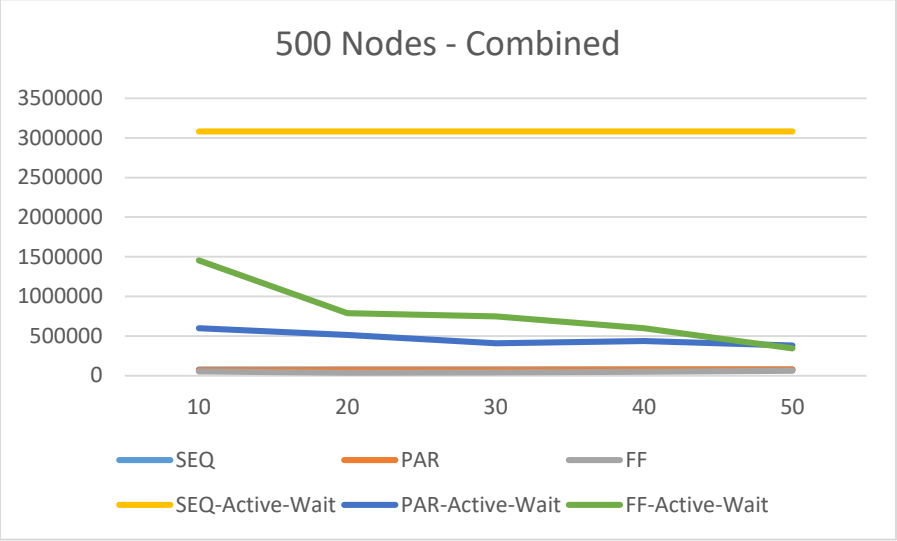
Test 5 – 1000 Nodes combined result:

	10	20	30	40	50
SEQ	394974	394974	394974	394974	394974
PAR	283620	245284	258609	262150	271628
FF	129264	133197	142640	148580	187029
SEQ-Active-Wait	6270696	6270696	6270696	6270696	6270696
PAR-Active-Wait	1132782	1304582	1078541	640406	784029
FF-Active-Wait	1599067	807810	757266	723283	811894



Test 6 – 500 Nodes combined result:

	10	20	30	40	50
SEQ	75796	75796	75796	75796	75796
PAR	70126	73802	74042	75340	77899
FF	54076	36207	39248	49477	60963
SEQ-Active-Wait	3081858	3081858	3081858	3081858	3081858
PAR-Active-Wait	596789	512346	407207	435910	381493
FF-Active-Wait	1452768	785872	746563	598458	343752



Result Summary:

On an average **I have got a speedup of around 2-3 with 10 workers** and **around 7-8 with 50 workers**. The speedup also depends on number of nodes and size of the graph. In some cases with small graph the SEQUENTIAL time is almost equal to PARALLEL time since the size of the graph is very small and the interaction between the threads takes more time than usual.