Graph Search Application Project

Elia Piccoli (621332)

July 21, 2021

Final project for the
Parallel and distributed systems: paradigms and models
course

University of Pisa
Artificial Intelligence
A.Y. 2020/2021

Contents

1 Introduction

In this report the main target is to analyze how parallel computation can be exploited to obtain a possible speedup during a custom task such as *graph search*. The task consists in visiting a *Directed Acyclic Graph* starting from a given *node* and counting the number of nodes that have a given *value*. The visit follows a *breadth first* fashion introducing a barrier at the end of the computation of each level. This synchronization process will play an important role during the parallel computation since all threads need to finish the computation of *level k* before proceeding to *level k+1*. For each level a *frontier* of all the nodes to be visited will be exploited to define the amount of work. The process ends when all leaves are reached.

It is easy to see that the computation of each node in the frontier is independent, so the task can be parallelized in order to achieve a speedup. Even though, at the end of the day, things are not that easy since many factors influence the results.

2 Graph generation

In order to correctly create the graph a simple generation script was implemented. The graph topology is strictly related to the following constants:

- **MAX/MIN_BREADTH** : *max/min range* of nodes for each level
- **MAX/MIN_DEPTH** : *max/min range* of levels in the graph
- **PROB_EDGE** : *probability* of creating an edge between two nodes
- **LEVEL_DISTANCE** : *maximum distance between two nodes* before decreasing edge probability

The generation proceeds extracting randomly the number of levels - depth of the graph - then iterating over them it creates a random amount of nodes for each one. During the generation of $level_k$, the algorithm considers all the nodes at $level_{1:k-1}$ and for each one pick a random number and if it is less then ***PROB_EDGE*** it creates an edge.
Using this basic implementation it is easy to adjust the algorithm to create different graphs. In fact it is possible to penalize edges between far apart nodes or vice versa create a dense graph. It is also easy to vary the graph level by level defining a range for the number of nodes and exploiting the random selections. This variety will be analyzed later, since it can heavily influence the performance of the parallel algorithm.

3 Problem analysis and expected results

As previously mentioned the task of graph search is characterized by a breadth first visit of the graph, that implicitly defines a precise order for the computation: visit all nodes of *current* level before moving to the *next* one. This implicitly creates two frontiers: the *current* one, which contains all the nodes to be processed; the *next* one, made up of all the children of

the current frontier's nodes.
The computation of each node in the frontier is independent. This leads to the use of a *Map* pattern to compute the different nodes and a *Barrier* to wait that all the thread have finished computing nodes of $level_k$, before updating structures and moving to $level_{k+1}$. Given this brief and high level analysis of the problem, it is possible to say that the *expected speedup results* should follow the ones of the Farm pattern.
Completion time for this task can be defined as follows:

$$CT = t_{create\_graph} + t_{graph\_search} \tag{1}$$

The creation of the graph is the *serial fraction* of the application so the speedup can be achieved in the computation of the search. Taking a closer look at the second term of (**1**), the topology of the graph plays an important role in defining the amount of work.

$$t_{graph\_search} = \sum_{i}^{level(G)} |frontier_i| * t_{node} \tag{2}$$

Where $level(G)$ indicates the number of levels in a breadth fist visit of the graph G.
The previous considerations lead to an expected speedup, using *nw* workers

$$speedup(nw) = \frac{t_{graph\_search}}{level(G) * nw * (t_{split} + t_{merge}) + \frac{t_{graph\_search}}{nw}} \tag{3}$$

4 Graph search: implementation

Both implementations share a similar structure and interface that allows to easily test different configurations.

executable_name [path_to_nodes_file] [starting_node] [value] [nw]

- **path_to_nodes_file**: created during graph generation, contains all the couples of nodes that are linked
- **starting_node**: id of the starting node for graph visit
- **value**: value to search during visit
- **nw**: number of workers

4.1 C++ threads

As highlighted in previous sections, the task of *graph search* is an iterative process over the levels in the *breadth-first* visit of the graph. Each level is characterized by a frontier of nodes. In order to visit and analyze it the solution is structured following the Map pattern. During the computation of the *i-th level*, the total number of nodes in the frontier is equally distributed among the workers, therefore using a *static partitioning*. Once all workers have finished, a sequential part of code merges the partial frontiers, computed by the different threads, to create the frontier for *level i+1* and updates the visited nodes. The counting is handled by the threads locally during their computation. Only at the end, all the partial counts will be added to get the *final* and total count of nodes with a particular value.
This implementation avoids problem such as *load balancing*, since the same amount of work is given to all the threads. The same goes for the problem of *false sharing*. The shared structures such as the graph, the current frontier and visited nodes are accessed in read only and the updates of shared variables happens once at the end of the worker computation. The *visited nodes* structure is updated once at the end of the level visit, in this way all possible problems related to synchronization during parallel computation are avoided. One important choice, that lead to a significant speedup for the parallel solution, is to use only structures storing the node-id. In this way a small amount of memory is allocated, paying a small constant time for accessing the map structure.

4.2 FastFlow

As far as concerns the *FastFlow* implementation, it follows a similar structure with respect to the C++ threads exploiting the Farm pattern. In order to create the Farm the choice was to use the *parallel building block*. In this way, exploiting the low level interface of the library, it was possible to create the specific implementation for *Emitter*, *Workers* and *Collector*.
Two different implementation of the problem are given. The **first** one, creates a *"classic" farm* with a feedback channel between the Emitter and Collector. The *Emitter* distributes the tasks among the workers using the default scheduling policy. The *Workers* visit the given node and produce the next local frontier. The *Collector* collects the output from the Workers creating locally the next frontier, and, once all workers have finished, updates the

visited node structure and sends the *next frontier* to the Emitter. The **second** version, instead, implements a *master-worker* structure, merging in one single structure Emitter and Collector. In this case the task that in the previous version was divided between Emitter and Collector now is handled by one single node, while the Workers maintain their structure. Also in these versions of the solution, shared structures are accessed in read-only mode and updated when no possible conflicts can occur avoiding synchronization. In this case the algorithm requires one or two "extra" threads to execute Emitter/Collector. In order to not use a number of threads greater than the available resources a simple check was introduced.

5 Graph search: results

In the following sections four different scenarios will be taken into consideration, in order to analyze and reason about the behaviour of the solution and its speedup. The results have been collected on the shared machine *averaging* the times obtained by 5 different runs of the solution. At the end a small section will analyze the results obtained with my laptop.
The graphs that will be used to analyze the behaviour are the same for the server and local machine. Each solution was tested using an *increasing number of workers* following the power of two, reaching 256 on *XEON PHI* and 16 on *Intel i9-9880H*. Correctness was tested checking the results obtained with the sequential implementation. Times for the serial fraction of the solution - graph creation - are reported in **Table 5**. In **Figure 1-3** instead is reported **speedup**, **scalability** and **efficiency** for the different tasks.

$$sp(nw) = \frac{T_{seq}}{T_{par}(nw)}$$
$$scalab(nw) = \frac{T_{par}(1)}{T_{par}(nw)}$$
$$\varepsilon(nw) = \frac{T_{seq}}{nw * T_{par}(nw)}$$

5.1 10k nodes

First scenario that will be analyzed is a graph of *10k nodes*. It was created in order to be a dense graph, meaning that all the nodes have the same probability of having an edge with a node in the successive levels. This implies that the visit of the graph will be completed using only few iterations.

To gather results node with **id 0** was chosen as starting node and this resulted in a ***six** level search*. The sizes of the frontiers were respectively: *1, 186, 7324, 2196, 97, 6.* In **Table 1** the averaged results for this test case are reported for the three solutions. Taking a look at the size of the frontiers and the results in the table it is possible to make some considerations. First of all, using just **one worker** highlights the overhead due to the creation of the thread, especially in the FastFlow versions that requires more than one thread. Increasing the number of workers the FastFlow versions have a smaller time until **8** workers, after that point the C++ version takes the lead, as we can see in **Figure 1**.
Taking a look at the output of the program, where it is reported the computational time for each level, it is easy to observe that **levels 3/4** are the one that impact more the times. Up to **8** workers FastFlow is faster, after that point almost all levels becomes slower and this leads to worse performances. For both implementation **16** represents the *knee point*, from that point on the overhead caused by the smaller levels and the not so big speedup in the bigger levels decrease significantly the performances of the solutions, as we can also see in the scalability plot (**Figure 2**).

Graph Search 10k (μsec)			
ParDegree	**C++ Threads**	**FastFlow v1**	**FastFlow v2**
Seq	153356	153356	153356
1	156366	181443	178537
2	103553	102144	100876
4	58742	56631	54004
8	35970	37918	36218
16	30639	36588	35751
32	42131	44306	45207
64	67911	72376	71849
128	93413	109712	117694
256	132799	215848	195717

Table 1: Average time on XEON PHI machine 10K nodes

5.2 50k nodes

Also in this case a dense graph is provided as input, so the same considerations hold in this scenario. As a matter of fact, using **node 0** as starting point a ***five** level search* is performed with the following frontiers sizes: *1, 1126, 52527, 2344, 22.* Differently from the 10K nodes case the sizes increase, so theoretically it suggest that the solution should scale better. In fact, as reported in **Table 2**, and **Figure 1-2**, is possible to catch this behaviour. Both versions performs well and, more or less, with the same times up to **32** workers, after that the C++ version reveals to behave way better that the FastFlow ones. This holds up to **64** workers, that marks the *knee point* for this scenario. To better understand why C++ threads perform better than FastFlow, using with 64 workers, is important to take a look at level's computation time. In the case of C++ implementation *level number 2* requires *almost half* of the time with respect to the FastFlow versions, that even if performs better in later levels, it is not enough to overcome the gap created. This behaviour obviously extends and becomes even bigger using a higher number of workers. This gap is probably related to the *different* implementation of the *Farm.* C++ threads handles a range of nodes in the current frontier to create a single local queue. This implies more work for the single worker but a faster merge since only *#workers vectors* need to be merged. Instead, in the FastFlow implementation each node in the frontier represents a task, so *#nodes vectors* have to be composed by the Collector, or Master, to locally create the next frontier. Given the fact that from 1126 nodes we get a next frontier of 52527 nodes this difference becomes significant.

Graph Search 50k (μsec)			
ParDegree	**C++ Threads**	**FastFlow v1**	**FastFlow v2**
Seq	12195552	12195552	12195552
1	12195552	12543560	12397940
2	7203391	7168818	7184056
4	3442903	3651416	3668038
8	1643104	1842510	1866226
16	817977	944083	949796
32	523801	551275	531448
64	396229	484147	449298
128	380480	536545	520755
256	403826	689603	637788

Table 2: Average time on XEON PHI machine 50K nodes

5.3 100k nodes

After the interesting results obtained from the the previous case, a new scenario with double the number of nodes was analyzed. But not all that shines is gold, in fact two different graphs sharing the ***same*** *number of nodes* but ***different topology*** will be analyzed. The first graph, as previous ones, will be a dense, while in the second one only nodes that are in a small range of levels can have edges. This implies a "longer" - more levels - visit but also smaller frontiers. These two versions have been taken into consideration to better understand and analyze how the topology of the graph plays a huge role in the results of this application.
Starting the analysis with the first graph the task is solved with a ***six*** *level visit* with the following frontier sizes: *1, 1951, 95518, 2237, 97, 6*. In **Table 3** are reported the times, that as one might expect given the considered graph, decrease nicely while growing the number of workers with a knee at **128**. As for the previous case the C++ and FastFlow versions follow a similar behavior up to **32/64** workers. Further increasing the number of workers highlights the problem that was already analyzed in the previous section, creating a gap between the solutions. In this case the difference is amortized since we have a high number of workers, but still we can see a small difference when comparing the level's computation time.

Graph Search dense 100k (μsec)			
ParDegree	**C++ Threads**	**FastFlow v1**	**FastFlow v2**
Seq	42798560	42798560	42798560
1	43116518	44876760	44708200
2	26478429	25448100	25378400
4	12953215	13433660	13279080
8	6373446	6647696	6674260
16	3176209	3323652	3390624
32	1972228	1876790	1888920
64	1383594	1482734	1483730
128	1212535	1380670	1425216
256	1229456	1529044	1565692

Table 3: Average time on XEON PHI machine 100K nodes *dense* version

Moving to the second graph we have a completely different situation. In this case, given the topology of the graph, a ***twelve*** *level search* is performed

where the levels have respectively the following sizes: *1, 475, 18695, 10474, 9991, 9995, 9996, 9989, 9990, 9993, 9988, 154*. Comparing the sizes with the previous cases the values are smaller and the number of nodes in the frontiers is more or less constant providing the same amount of work.
Considering the results reported in **Table 4** and the speedup/scalability plots (**Figure 1-2**), compared to the other 100k version of the graph this one performs poorly. In this case the amount of work remains constant across the levels while previously one big frontier had to be computed. This is a *key factor* for the performances results. The big frontier plays the role of bottleneck of the application and while we increase the number of workers the speedup given by the parallel computation dominates other overheads. Having instead the same small amount of work for all the levels implies that we gain some speedup using more workers, that eventually will have smaller and smaller work, but at the same time an increasing overhead. This can be observed in the plots were the speedup remains more or less constant highlighting the trade-off between more workers and more overhead. The FastFlow versions even in this case suffers the previously analyzed problem, and as a matter of fact computation time for each level is bigger with respect to the C++ version creating the gap between the two.

Graph Search sparse 100k (μsec)			
ParDegree	**C++ Threads**	**FastFlow v1**	**FastFlow v2**
Seq	7832274	7832274	7832274
1	7855229	11028920	10952080
2	5184445	5945956	6157658
4	3213317	3051964	3123908
8	2169737	1968538	2086768
16	1633289	1842420	1952514
32	1532741	1793472	1910448
64	1504910	2448796	2500828
128	1588359	3401744	3489118
256	1751640	4143240	4153062

Table 4: Average time on XEON PHI machine 100K nodes *sparse* version

5.4 Local machine

Just for fun and to analyze a different machine, the 50/100k dense graphs were tested on the local machine. Results shown in **Table 6** and **Table 7** give an idea on the difference between a laptop CPU, more general purpose and powerful in single thread computations and a CPU specialized in multi threading. The local results shows a 22x faster results but a limited degree of parallelism and speedup as we can see in **Figure 4**.

6 Conclusions

This project shows how a custom task such as graph search can be easily implemented exploiting a simple parallel pattern. Three different solutions were proposed using the standard C++ threads but also the parallel building blocks implemented in the FastFlow library.
Results exhibits how this task is strictly related to the topology of the graph: number of levels, number of children per node, size of frontiers etc. The problem revealed to be not that easy and predictable, so a careful and precise analysis was needed in order to give a correct explanation to results.

7 How to run

In the project folder a **Makefile** is provided.
One can simply run *make all* to create all the executables or can use the single rules to create a specific one.

- *make gg*: generate graph
- *make gs*: graph search C++ threads version
- *make ffgs*: graph search FastFlow version

Important: to correctly compile the FastFlow version update the path for the FastFlow library include.
In **4** the interface of the main program has already being presented more in depth. Only the expected input pattern is reported.

- **generate graph** : *executable_name*
- **graph search** : *executable_name [path] [starting_node] [value] [nw]*

8 Appendix

8.1 Graph generation parameters

In order to generate the same graph here are reported the various parameters:

- **10k**: BREADTH = 50, DEPTH = 200, PROB_EDGE = 5, LEVEL_DISTANCE = 3, SEED = 1234
- **50k**: BREADTH = 75, DEPTH = 750, PROB_EDGE = 5, LEVEL_DISTANCE = 3, SEED = 1234
- **100k dense**: BREADTH = 50, DEPTH = 2000, PROB_EDGE = 5, LEVEL_DISTANCE = 3, SEED = 1234
- **100k sparse**: BREADTH = 100, DEPTH = 1000, PROB_EDGE = 5, LEVEL_DISTANCE = 100, SEED = 1234, use **continue** instead of multiply by 3

[100K dense needs 1.1GB]

Graph Generation (μsec)		
Graph	**XEON PHI**	**i9-9880H**
10K	989237	114726
50K	32277482	4215627
100K#1	109541282	15670317
100K#2	43199933	5622786

Table 5: Average time for creating graph structure

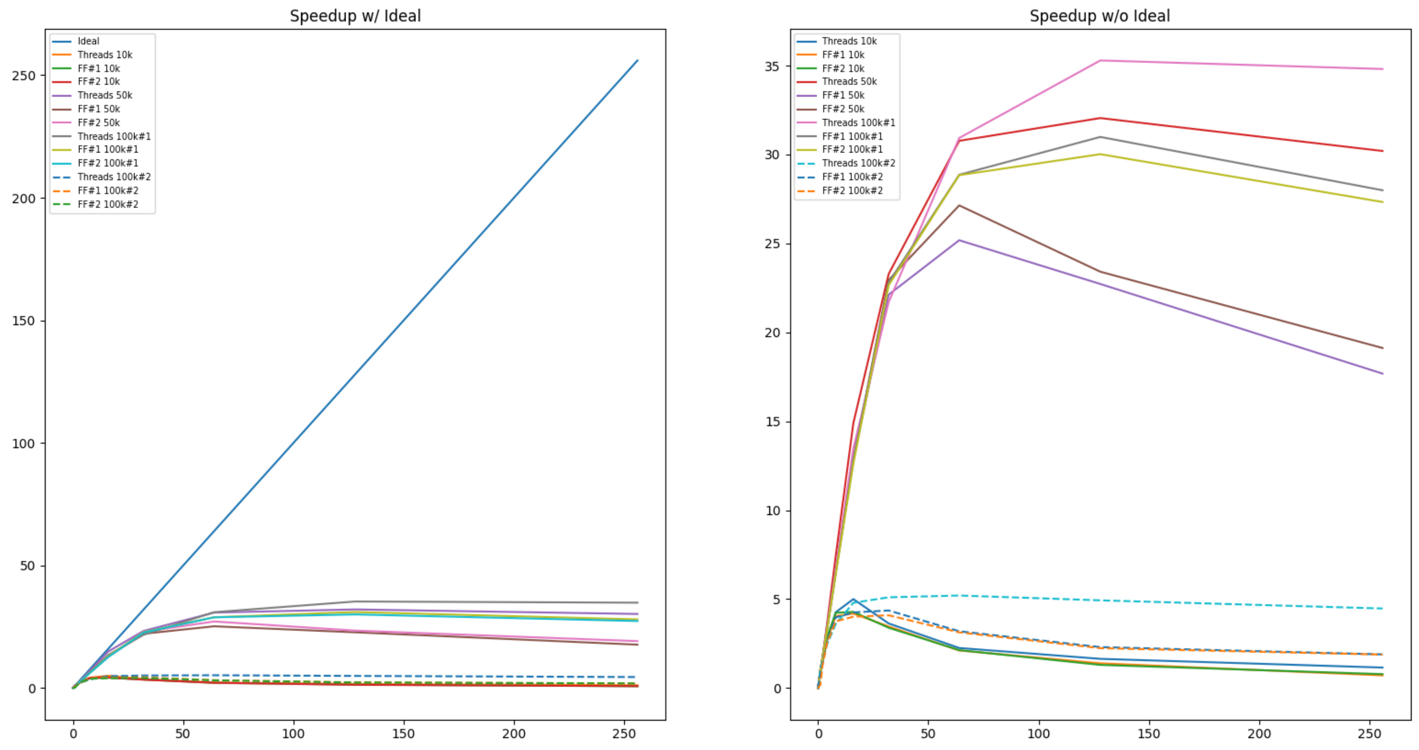

Figure 1: Speedup Graph Search XEON PHI

Graph Search 50k (μsec)			
ParDegree	**C++ Threads**	**FastFlow v1**	**FastFlow v2**
Seq	548047	548047	548047
2	320773	295581	314316
4	200677	223583	201945
8	149875	168963	154976
16	108846	121879	113448

Table 6: Average time on Intel i9-9880H 50K nodes *sparse* version

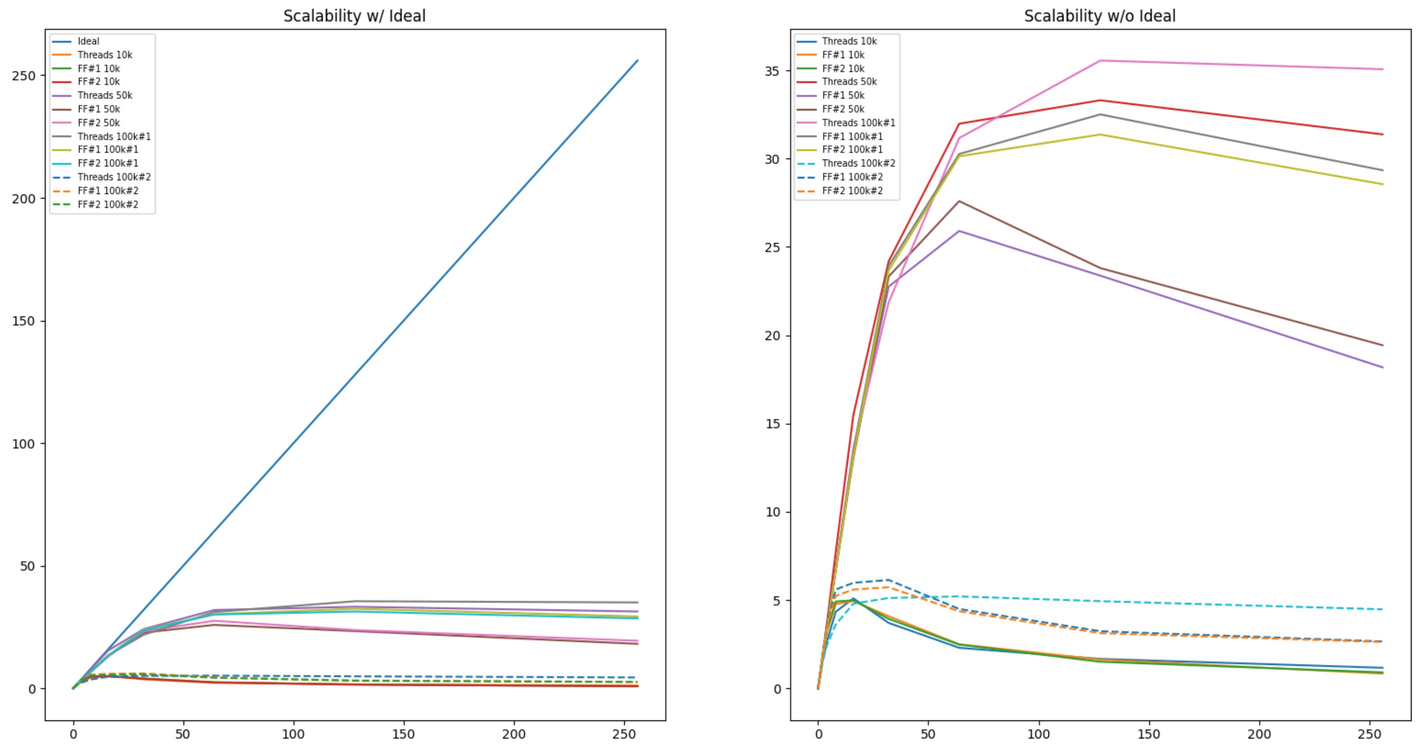

Figure 2: Scalability Graph Search XEON PHI

Graph Search 100k (μsec)			
ParDegree	**C++ Threads**	**FastFlow v1**	**FastFlow v2**
Seq	1973776	1973776	1973776
2	1197080	1175578	1052400
4	642030	638513	733634
8	465218	489368	479579
16	336081	362623	352784

Table 7: Average time on i9-9880H 100K nodes *sparse* version

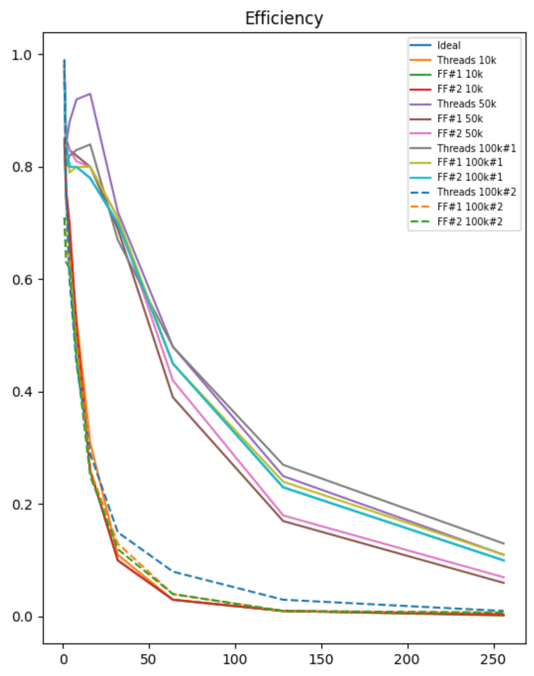

Figure 3: Efficiency Graph Search XEON PHI

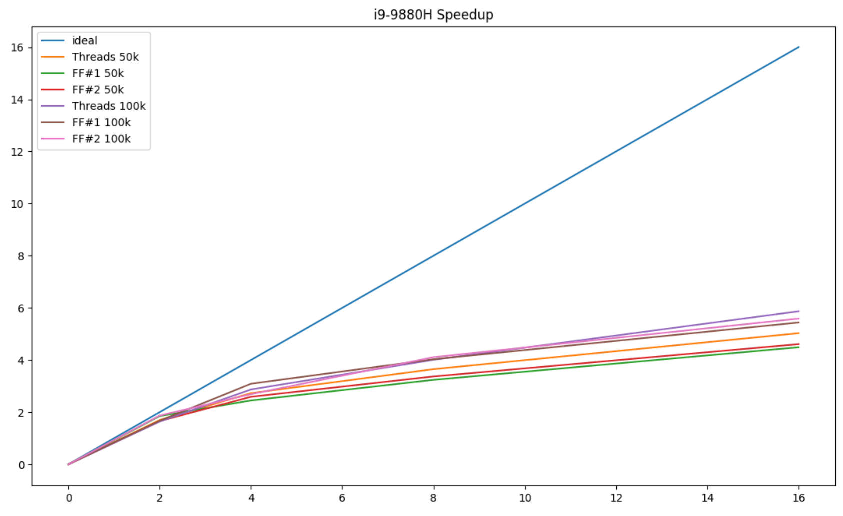

Figure 4: Speedup Graph Search i9-9880H