Parallel Graph Coloring

System and Device Programming

Andrea Cavallo
Politecnico di Torino
Torino, Italy
s287965@studenti.polito.it

Gioele Scaletta
Politecnico di Torino
Torino, Italy
gioele.scaletta@studenti.polito.it

September 3rd, 2021

Abstract

The project aims at implementing and analyzing different algorithms to perform graph coloring, comparing the performances of sequential and parallel approaches. The algorithms are taken from Allwright et al., 1995. Different parallelization techniques are experimented.

1 Introduction

1.1 Language and HW support

The project is developed in C++ using Visual Studio 2017 and it is run using Windows. The computer used for the run has the processor i7-8550U, 1.80 GHz with 16 GB of RAM.

1.2 Data structure

Initially, the project was organized in three classes: node, edge and graph. The graph class contained a vector of node objects and a vector of edge objects. However, it became soon clear that this structure was not efficient, as the memory occupancy was high and it was difficult to store even small graphs. Therefore, a different approach was tested.
In particular, the new data structure used to store the graphs is an adjacency list implemented by means of vectors. Nodes are identified by an integer number and for each node in the graph there is a vector (created using the container of the C++ Standard Template Library) containing the identifiers of the neighboring nodes. Additional support structures are used for the colors of the nodes (a vector) and for other information needed by the coloring algorithms. This approach proved to be more efficient than the previous one, and it was therefore selected for the project.

1.3 Algorithms

Four algorithms are used, which are then implemented in different versions. The first one is a greedy sequential algorithm.

Algorithm 1 Greedy algorithm

Given $G = (V,E)$
$n = |V|$
choose a random permutation of the vertices V
for i=1 to n **do**
    select v_i
    C = colors of all colored neighbors of v_i
    c = smallest color not in C
    color v_i with color c
    $U = U - v_i$
end for

Then, the Jones-Plassman algorithm is implemented, which assigns random weights to each node in order to parallelize color assignments.

Algorithm 2 Jones-Plassman

```
assign random weights to each node w
U := V
while |U| > 0 do
    for all vertices v ∈ U do in parallel
        I := {v such that w(v) > w(u) for all neighbors u ∈ U}
        for all vertices v' ∈ I do in parallel
            S := {colors of all neighbors of v'}
            c(v') := minimum color not in S
        end for
    end for
    U = U − I
end while
```

The third algorithm is the Largest Degree First, whose only difference with respect to the Jones-Plassman algorithm is that weights are not assigned randomly to nodes but they are chosen to be the degree of the nodes. This choice is supposed to reduce the number of used colors, even though the time efficiency of the algorithm may decrease.

In conclusion, the Smallest Degree Last algorithm is implemented. With respect to the LDF, the weight assignment follows a more complex procedure, described by algorithm 3.

Algorithm 3 Assignment of weights in SDL

```
k = 1
i = 1
U := V
while |U| > 0 do
    while ∃ vertices v ∈ U with d^U(v) ≤ k do
        S := {all vertices v with d^U(v) ≤ k}
        for all vertices v ∈ S do
            w(v) = i
        end for
        U := U − S
        i := i + 1
    end while
    k = k + 1
end while
```

1.4 Parallelization strategies

1.4.1 Efficient partition of tasks among threads

Concerning the parallelization methods, the first implementation consisted in a single thread coloring one node at a time. However, it was soon clear that creating one thread per node was very inefficient. Therefore, an alternative strategy was applied, which consists in dividing the nodes in groups and assigning to each thread a group of nodes to color. To perform this task in a correct way, it was necessary to consider some issues related to the fact that if two threads try to color two neighbor nodes, a deadlock can occur. As a matter of fact, when one node is checking the color of the neighbor node, another thread could be coloring it at the same time. The issue was tackled by dividing the coloring in two phases: find nodes to color, color the nodes. In the first phase, threads look for nodes that can be colored, and store their future color in a proper vector. Then, in the second phase, each node is assigned the color that was previously identified. In this way there is just a small additional overhead, but no deadlock can happen. Moreover, the function to check if a node is a local maximum and therefore can be colored (`isLocalMaximum()`) and the function to determine the minimum available color (`getMinColor()`) do not need any lock because finding nodes to color and coloring them are two separate phases (so, it is impossible that the color of a node is modified while it is being checked).

1.4.2 Threadpool

Moreover, to reduce the overhead related to thread creation and termination, a threadpool system was implemented, where the jobs consist in either finding nodes to color or coloring them among a specified subset of nodes. Threads are created at the beginning of the algorithm and they terminate at the end, so the overhead of their creation is minimized. Each of them takes jobs from a queue and executes them until a termination condition is met (i.e. all nodes are colored). The master thread creates the jobs (by generating subsets of the nodes to be checked or colored) and adds them to the queue. A flag is used to determine whether all nodes are colored or not, and consequently to check if the algorithm has to stop or to proceed to another iteration. This approach turned out to be successful and brought a visible improvement in the performances.

1.4.3 Selecting and coloring nodes

The described strategy has been implemented in two different versions. In the first case, some jobs take care of finding nodes to color and other jobs color them. In the second alternative, identified as *find and color* in the following, the same job finds which nodes can be colored, it stores them in a local queue and, after synchronizing with other running threads, it colors them.

1.4.4 Synchronization strategies

Given the unique partition of nodes assigned to each job, as described above, threads do not need to acquire any mutex when checking if a node can be colored or to color it. However, synchronization strategies are needed for the management of the threadpool and also for thread creation and termination.

- For the functions that do not use a threadpool, a condition variable is used, where the condition is given by the number of active threads (global variable protected by a mutex): each time a thread terminates, it decreases the number of active threads. The main thread waits for the number of active threads to be lower than the maximum allowed number of active threads, and when this condition is met it creates a new thread. In this way, the maximum available concurrency is exploited.
- For the management of the threadpool, instead, two condition variables are used. The first one manages the queue of jobs, allowing the main thread to insert new tasks and the working threads to execute the available jobs, whereas the other one is used to synchronize all jobs. In particular, a global variable keeps track of the jobs that have been scheduled and it is decreased by each job, before its termination. In this way, the main thread can wait until all jobs have been completed and proceed with the next phase or iteration of the algorithm.
- In conclusion, for the functions using the *find and color* approach, the threads need to synchronize after finding all nodes that can be colored in a given iteration. To achieve this, all threads wait on a condition variable that counts how many threads are still busy. When all threads have reached the synchronization point, they move on to coloring the nodes. In this case, the condition variable acts as a barrier.

2 Hyperparameter optimization

The implemented algorithms include some hyperparameters that can be tuned in order to achieve the best performances. In particular, it is possible to set the number of concurrently running threads and the size of the portion of the graph each thread works on. This second parameter is expressed through a coefficient $coef$ that is related to the number of nodes for each thread through the formula

$$nodes\_per\_thread = \frac{nodes\_in\_the\_graph}{coef * number\_of\_threads}$$

Figure 1 shows how these hyperparameters affect the performance of the algorithms on a subset of the test graphs. The performance is measured in terms of time required to color the graph and number of colors used. It can be observed that, for all algorithms, the optimum number of thread is 8 according to the time, whereas the number of colors is better with less threads. The choice of the coefficient does not bring notable differences. According to these results, the number of threads is set to 8 (which corresponds also to the number of concurrent threads supported by the machine) and the coefficient is set to 10 for the following analyses.

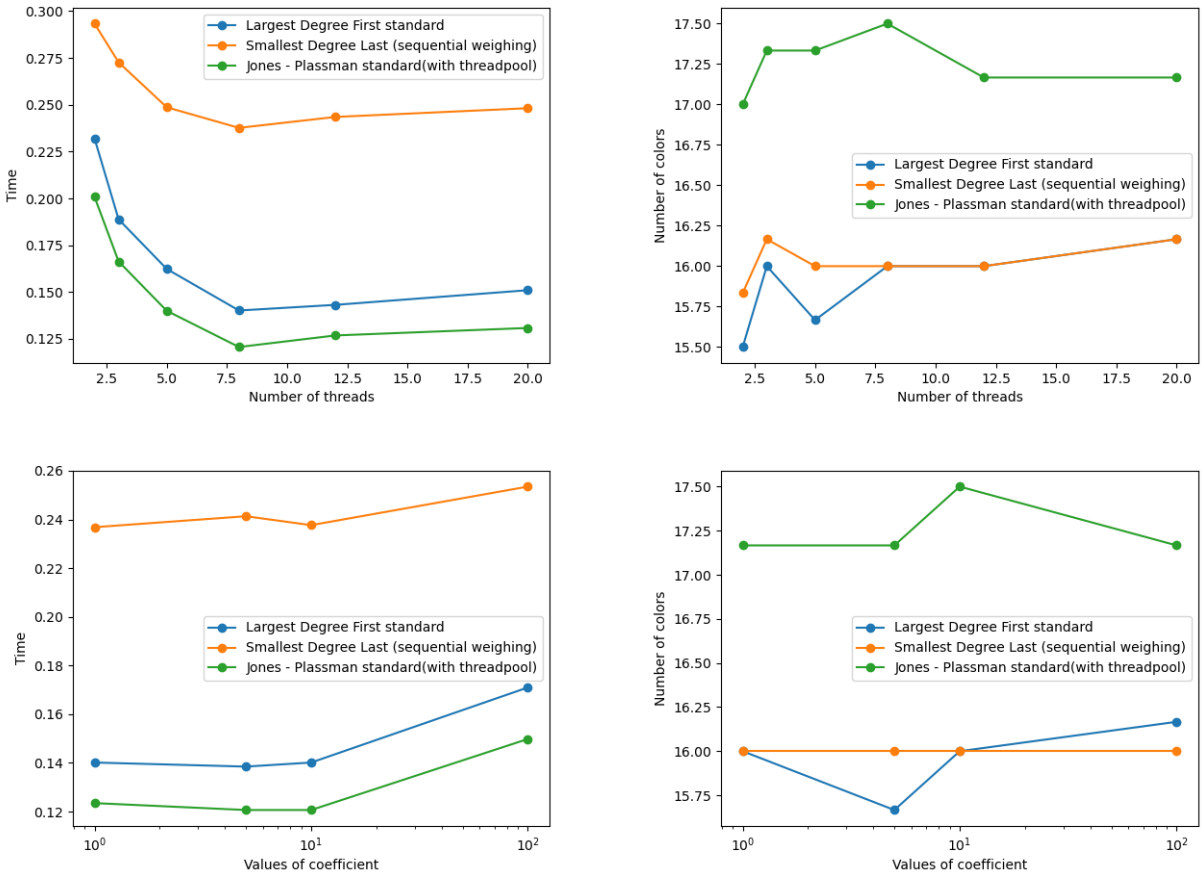


Figure 1: Values of times and colors versus number of threads and coefficient

3 Experimental results

Different algorithms are tested on many test graphs and the results are reported in this section. The tested algorithms are

- **Greedy0**: a simple greedy sequential algorithm
- **JP0**: Jones-Plassman algorithm, sequential version
- **JP1**: Jones-Plassman algorithm, parallel version without a threadpool (threads are created and destroyed when they finish their task)
- **JP2**: Jones-Plassman algorithm, parallel version with a threadpool (threads are created at the beginning and destroyed at the end of the algorithm, and they execute jobs that are scheduled by a master function and inserted in a queue)
- **JP3**: Jones-Plassman algorithm, parallel version with threadpool, *find and color*
- **LDF1**: Largest Degree First algorithm, parallel version with threadpool
- **LDF2**: Largest Degree First algorithm, parallel version with threadpool, *find and color*
- **SDL0**: Smallest Degree Last algorithm, sequential version
- **SDL1**: Smallest Degree Last algorithm, parallel version with a threadpool

3.1 Times

To begin with, the performances of the algorithms are analyzed in terms of the time required to color graphs. Table 1 shows the results.

Graphs	**Nodes**	JP0	LDF2	JP3	JP1	SDL0	SDL1	LDF1	JP2	Greedy0
citeseer.scc	693947	0.129	0.088	0.115	0.276	0.122	0.084	**0.042**	0.064	0.078
v10	10	**0.0**	0.002	0.003	0.023	**0.0**	0.004	0.002	0.003	**0.0**
go_uniprot	6967956	30.773	1.66	7.779	14.114	4.164	2.769	**1.218**	8.832	2.701
cuda	7	**0.0**	0.002	0.002	0.005	**0.0**	0.002	0.002	0.002	**0.0**
v100	100	**0.0**	0.012	0.012	1.238	0.001	0.019	0.02	0.02	**0.0**
amaze_dag_uniq	3710	0.001	0.003	0.004	0.193	0.001	0.004	0.005	0.004	0.001
rgg_n_2_18_s0	262144	0.281	0.104	0.098	0.741	0.313	0.194	0.084	**0.078**	0.12
citeseerx	6540401	16.437	6.11	6.123	7.941	4.37	2.387	5.194	5.069	**2.204**
rgg_n_2_22_s0	4194304	5.921	2.314	2.091	2.711	6.257	3.951	2.042	**1.821**	3.105
uniprotenc_22m.scc	1595444	0.318	0.186	0.243	0.266	0.297	0.153	**0.083**	0.134	0.351
vchocyc_dag_uniq	9491	0.003	0.005	0.005	0.292	0.002	0.005	0.005	0.008	0.002
ecoo_dag_uniq	12620	0.003	0.005	0.005	0.293	0.003	0.005	0.007	0.007	**0.002**
xmark_dag_uniq	6080	0.002	0.004	0.005	0.313	**0.001**	0.004	0.005	0.007	**0.001**
ba10k5d	10000	0.011	0.01	0.011	1.067	0.007	0.008	0.014	0.019	**0.003**
pubmed_sub_9000-1	9000	0.011	0.01	0.012	1.365	0.006	0.008	0.017	0.022	**0.002**
rgg_n_2_19_s0	524288	0.586	0.219	0.177	0.862	0.677	0.397	0.182	**0.162**	0.257
rgg_n_2_17_s0	131072	0.125	0.049	0.05	0.712	0.154	0.099	0.042	**0.041**	0.056
kegg_dag_uniq	3617	0.001	0.003	0.004	0.242	0.001	0.005	0.003	0.005	0.001
rgg_n_2_23_s0	8388608	12.612	5.212	4.385	5.019	13.169	8.447	4.615	**4.037**	6.574
human_dag_uniq	38811	0.008	0.007	0.008	0.271	0.008	0.008	0.007	0.008	0.004
rgg_n_2_20_s0	1048576	1.262	0.495	0.466	1.074	1.382	0.853	0.4	**0.365**	0.584
rgg_n_2_15_s0	32768	0.024	0.014	0.014	0.541	0.034	0.025	0.014	0.014	**0.012**
nasa_dag_uniq	5605	**0.001**	0.004	0.004	0.219	0.002	0.005	0.004	0.006	**0.001**
yago_sub_6642	6642	0.009	0.006	0.01	1.473	0.004	0.007	0.008	0.021	0.002
uniprotenc_100m.scc	16087295	4.24	2.765	3.303	2.288	3.386	2.039	**1.165**	1.594	4.121
rgg_n_2_21_s0	2097152	2.777	1.115	0.97	1.646	3.003	1.854	0.906	**0.826**	1.371
cit-Patents.scc	3774768	14.379	10.338	4.772	7.06	6.447	3.65	8.902	4.605	**1.768**
v1000	1000	0.032	0.1	0.101	19.499	0.086	0.276	0.247	0.252	**0.005**
rgg_n_2_16_s0	65536	0.056	0.027	0.027	0.655	0.072	0.046	0.025	**0.021**	0.023
agrocyc_dag_uniq	12684	0.002	0.005	0.006	0.27	0.003	0.005	0.005	0.006	**0.001**
arXiv_sub_6000-1	6000	0.02	0.018	0.019	2.404	0.011	0.02	0.032	0.03	**0.002**
ba10k2d	10000	0.004	0.006	0.005	0.582	0.004	0.005	0.007	0.01	**0.001**
go_sub_6793	6793	0.003	0.005	0.005	0.388	0.003	0.006	0.007	0.009	**0.001**
anthra_dag_uniq	12499	0.003	0.005	0.005	0.245	**0.002**	0.006	0.005	0.006	**0.002**
mtbrv_dag_uniq	9602	**0.002**	0.005	0.005	0.245	**0.002**	0.004	0.005	0.005	**0.002**
citeseer_sub_10720	10720	0.009	0.008	0.009	0.895	0.007	0.009	0.011	0.014	**0.003**

Table 1: Times (sec) to color different graphs using different algorithms

3.2 Colors

Then, the performance of the algorithms is analyzed in terms of the number of colors used for each graph. Table 2 shows the results.

Graphs	**Nodes**	JP0	LDF2	JP3	JP1	SDL0	SDL1	LDF1	JP2	Greedy0
citeseer.scc	693947	5	4	6	5	**2**	**2**	4	5	5
v10	10	**6**	**6**	**6**	**6**	**6**	**6**	**6**	**6**	**6**
go_uniprot	6967956	14	**7**	12	12	8	8	**7**	13	12
cuda	7	**2**	**2**	**2**	**2**	**2**	**2**	**2**	3	**2**
v100	100	**78**	**78**	**78**	**78**	**78**	**78**	**78**	**78**	**78**
amaze_dag_uniq	3710	**4**	**4**	**4**	**4**	**4**	**4**	**4**	**4**	**4**
rgg_n_2_18_s0	262144	18	**16**	18	17	17	**16**	**16**	18	17
citeseerx	6540401	20	**13**	21	21	**13**	14	**13**	20	21
rgg_n_2_22_s0	4194304	22	**20**	21	21	**20**	**20**	**20**	21	23
uniprotenc_22m.scc	1595444	4	3	4	4	**2**	**2**	**2**	4	4
vchocyc_dag_uniq	9491	5	**4**	5	5	**4**	**4**	**4**	5	5
ecoo_dag_uniq	12620	6	**4**	5	6	**4**	**4**	**4**	6	5
xmark_dag_uniq	6080	5	4	6	5	**3**	**3**	4	4	5
ba10k5d	10000	12	**7**	12	12	8	8	**7**	13	13
pubmed_sub_9000-1	9000	14	**9**	14	13	10	10	**9**	14	14
rgg_n_2_19_s0	524288	**18**	**18**	19	19	**18**	**18**	**18**	19	19
rgg_n_2_17_s0	131072	16	**15**	16	16	**15**	16	**15**	16	16
kegg_dag_uniq	3617	4	**3**	4	4	4	4	4	4	4
rgg_n_2_23_s0	8388608	23	**21**	23	22	22	23	**21**	23	**21**
human_dag_uniq	38811	5	**4**	5	5	**4**	**4**	5	5	5
rgg_n_2_20_s0	1048576	19	**18**	20	19	19	**18**	**18**	20	20
rgg_n_2_15_s0	32768	14	**13**	14	15	**13**	**13**	**13**	14	14
nasa_dag_uniq	5605	5	**4**	5	5	**4**	**4**	**4**	5	5
yago_sub_6642	6642	9	**6**	9	8	7	7	**6**	9	9
uniprotenc_100m.scc	16087295	5	3	5	5	**2**	**2**	3	5	5
rgg_n_2_21_s0	2097152	**19**	**19**	20	20	**19**	**19**	**19**	**19**	20
cit-Patents.scc	3774768	21	**14**	20	21	15	15	15	21	19
v1000	1000	752	**751**	**751**	**751**	**751**	**751**	**751**	**751**	752
rgg_n_2_16_s0	65536	17	15	16	17	16	**14**	15	16	17
agrocyc_dag_uniq	12684	5	**4**	5	6	5	**4**	**4**	5	5
arXiv_sub_6000-1	6000	30	27	29	30	**26**	27	28	30	31
ba10k2d	10000	8	5	8	8	**3**	**3**	5	9	8
go_sub_6793	6793	7	6	7	7	6	6	**5**	7	8
anthra_dag_uniq	12499	5	**4**	5	5	**4**	**4**	**4**	5	5
mtbrv_dag_uniq	9602	6	**4**	6	6	**4**	**4**	**4**	6	5
citeseer_sub_10720	10720	12	**9**	11	12	10	10	10	11	12

Table 2: Number of colors on different graphs using different algorithms

3.3 Considerations

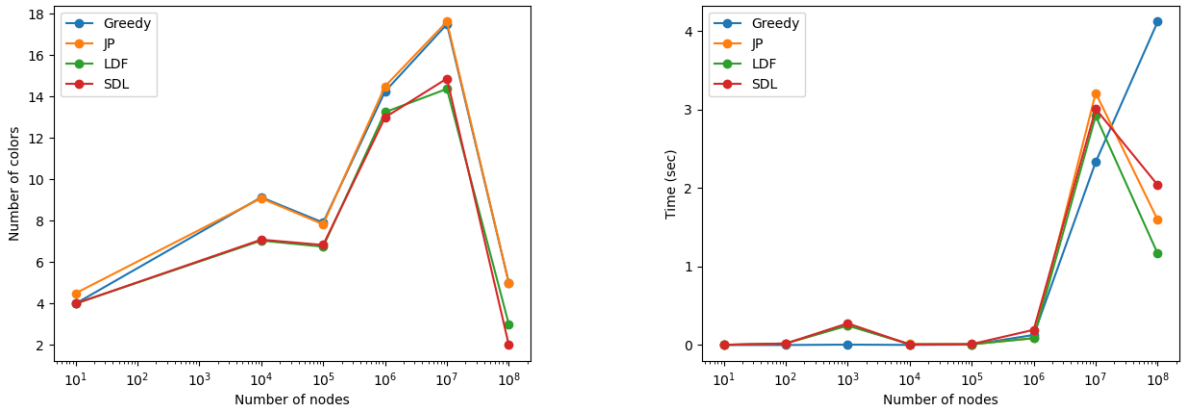
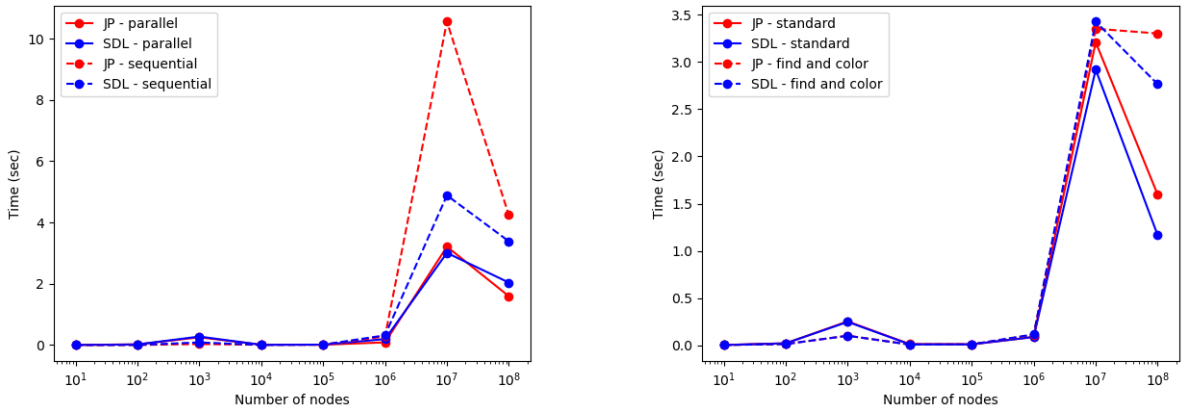


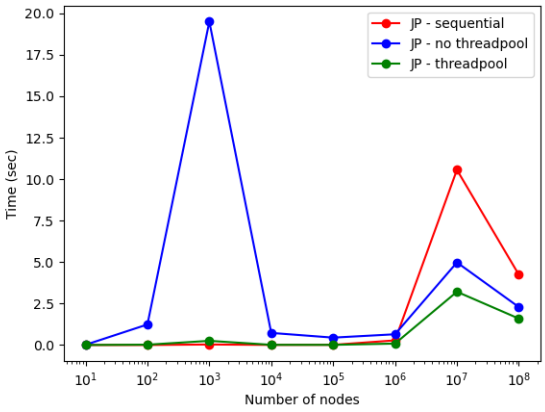
Figure 2: Times and colors versus number of nodes for different algorithms

From the above analysis, it is possible to compare the performances of different algorithms on the given test dataset. As shown in Figure 2, the greedy and the Jones-Plassman algorithms have comparable results in terms of number of colors used for each graph, and they are outperformed by the Largest Degree First and the Smallest Degree Last algorithms, which are also comparable. It is also possible to notice that there is a general exponential correlation between the number of colors and the number of nodes.



(a) Parallel and sequential algorithms

(b) Standard and *find and color* algorithms



(c) Threadpool and no threadpool

Figure 3: Further comparisons among coloring algorithms

Figure 2 shows also a comparison in terms of time needed by the different algorithms to color graphs of different size. It can be noticed that the performances of sequential and parallel algorithms are comparable for smaller graphs, whereas the advantages of parallelism become more evident with a bigger number of nodes.

Further observations can be done by considering Figure 3. In particular, it can be noticed that the parallel implementation of the algorithms improves significantly the performances in terms of time when dealing with big graphs (10^7-10^8 nodes), whereas for smaller ones the difference is less noticeable (Figure 3.a).
In addition to that, also a comparison between the standard implementation and the *find and color* version is reported in Figure 3.b. It can be noticed that the performances are similar, with the *find and color* version achieving better results on smaller graphs, while the standard implementation is better for larger ones.
Moreover, a comparison among different implementations of the Jones-Plassman algorithms (Figure 3.c) shows how the parallel version without threadpool performs worse than the sequential version for small graphs, whereas the version with a threadpool achieves significantly better results.

4 Memory

In conclusion, the efficiency of the data structure used to store the graph is analyzed in terms of the memory occupied by graphs of different size. As shown in Figure 4, small graphs tend to occupy a similar amount of memory, whereas for bigger graphs the memory occupation grows significantly.

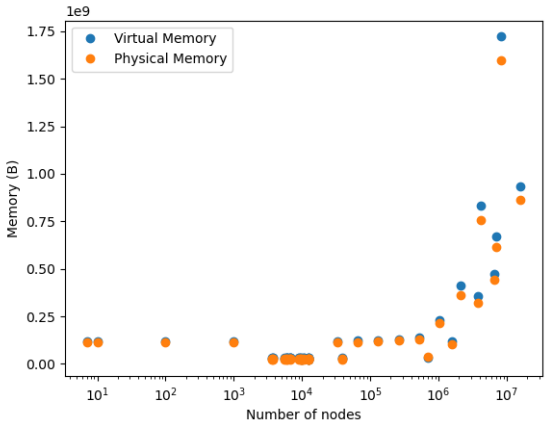


Figure 4: Memory occupied by graphs with different number of nodes