REPORT for Project 1

Approach:

In the radix sort we apply count sort to a dataset iteratively based on the maximum number of decimal places per entry in the dataset.

The approach to parallelize the radix sort was to create a 2D “vertex_cnt” array instead of a 1D one which allowed us to allocate a chunk of data to each thread.

For eg., if our data was of size 10 indexed from 0-9 and we had 2 threads t0 and t1 then each would be assigned five data points(in this case a pair of vertices with an edge between them) so t0 would compute vertex count for 0-4 points and t1 would cover 5-9.

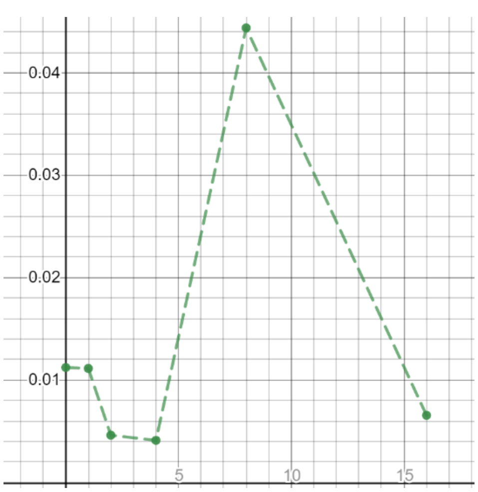
This helps us from preventing any race conditions and allows for correct synchronization of threads.

However, the cumulative sum step could not be parallelized and is hence done serially.

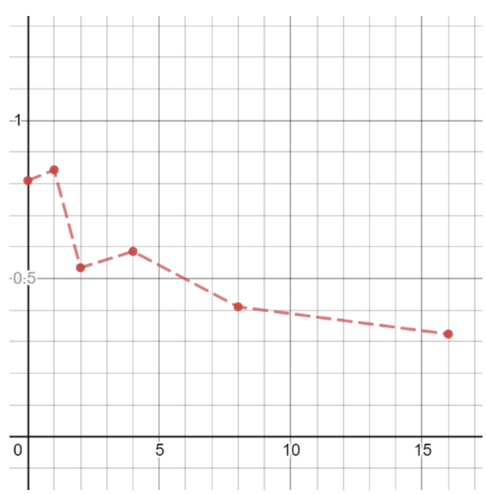
We then find the position of each element in the sorted array and this too is done parallelly using the vertex_cnt array.

OUTPUT

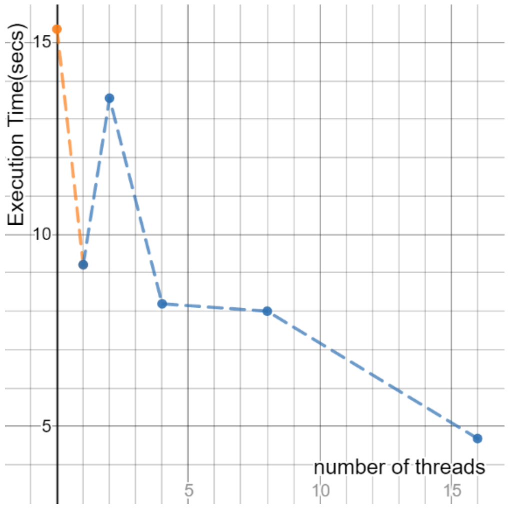
	Serial	**1**	**2**	**4**	**8**	**16**	**BFS**
Test	0.000007	0.000029	0.000107	0.000182	0.001874	0.000572	0.000001-0.000003
Wiki-vote	0.012247	0.013132	0.011598	0.004960	0.004656	0.005724	0.000184
Facebook	0.011220	0.011130	0.004607	0.004102	0.044384	0.006550	0.000014
RMAT 18	0.809575	0.843679	0.533916	0.585508	0.410135	0.323870	0.000496
Rmat 19	1.628316	1.706500	1.48785	0.933045	0.739003	0.551390	0.039710
RMAT 20	2.014778	3.955505	3.122015-3.356549	2.590064	1.943042	1.513439	0.001406
RMAT 21	-	-	-	-	0.045957		0.000206
RMAT 22	15.343040	9.212273	13.545454	8.194581	8.001994	4.687720	0.003156



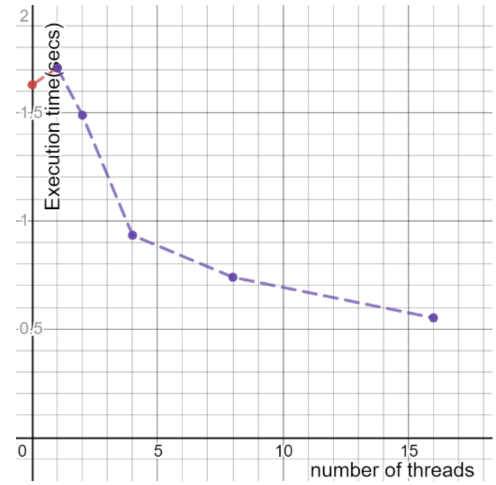
Facebook



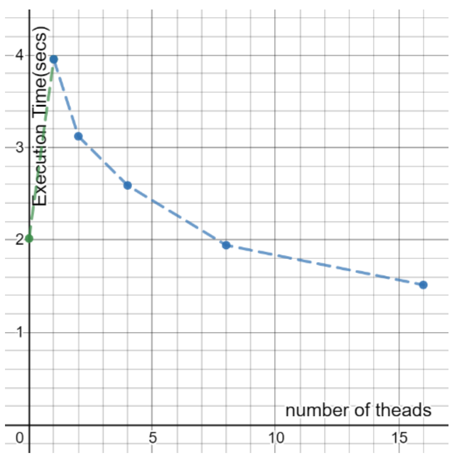
RMAT 18



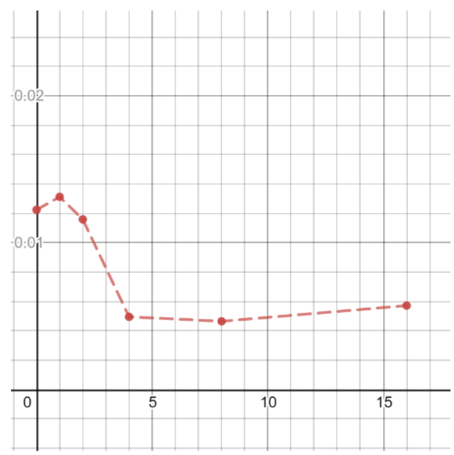
RMAT 22



RMAT 19



RMAT 20



WIKI-vote

Performance:

As we can see from the graphs, the performance improves as the number of threads increases provided that the dataset is large enough because if the dataset isn't large enough the overheads for multithreading outweigh the benefits of multithreading like in the case of the "test/ RMAT test" dataset.

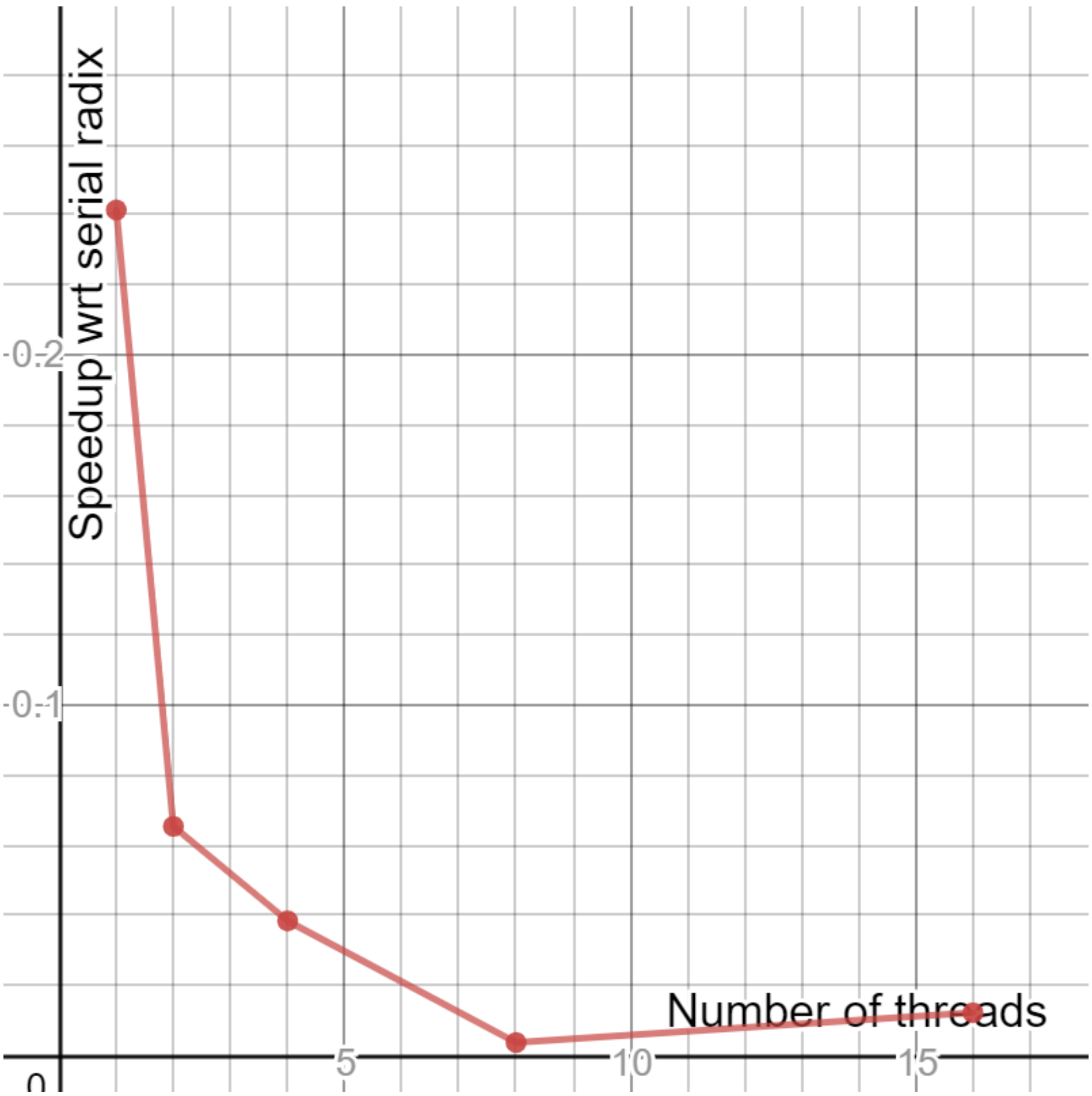
Also, we observe that sometimes the increase in number of threads can slowly plateau the performance like in the case of the facebook dataset. When we increase the number of threads from 1 to 2 we see a significant improvement in execution time but as we further increase the number of threads to 4 we see a plateau in the execution time.

One more thing I noticed was that every time I ran the code for same number of threads on the same dataset the execution time differed each time slightly. This is because of the memory access times changing because of various factors.

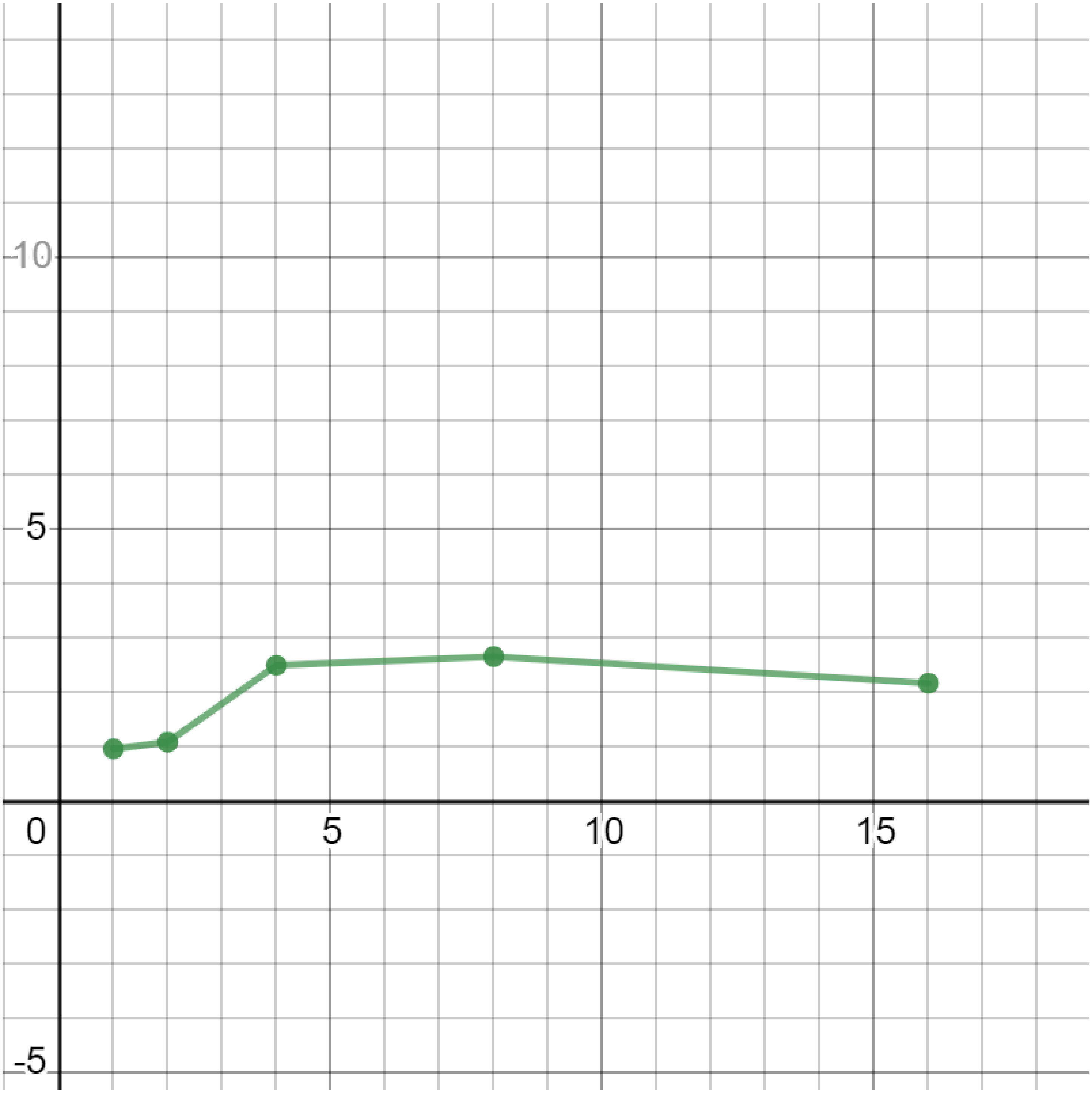
Speedup = $\frac{Serial\ Runtime}{Parallel\ Runtime}$

	1	2	4	8	16
Test	0.24137	0.06542056075	0.03846153846	0.003735325507	0.01223776224
Wiki-vote	0.9326073713	1.055957924	2.469153226	2.630369416	2.139587701
Facebook	1.008086253	2.435424354	2.735251097	0.2527937996	1.712977099
RMAT18	0.9595770429	1.516296571	1.382688196	1.97392322	2.499691234
RMAT19	0.9541845883	1.094408711	1.745163417	2.203395656	2.953111228
RMAT20	0.5093604988	0.6453453939	0.7778873418	1.036919428	1.331258148
RMAT22	1.665499926	1.132707697	1.872339782	1.917402088	3.273028253

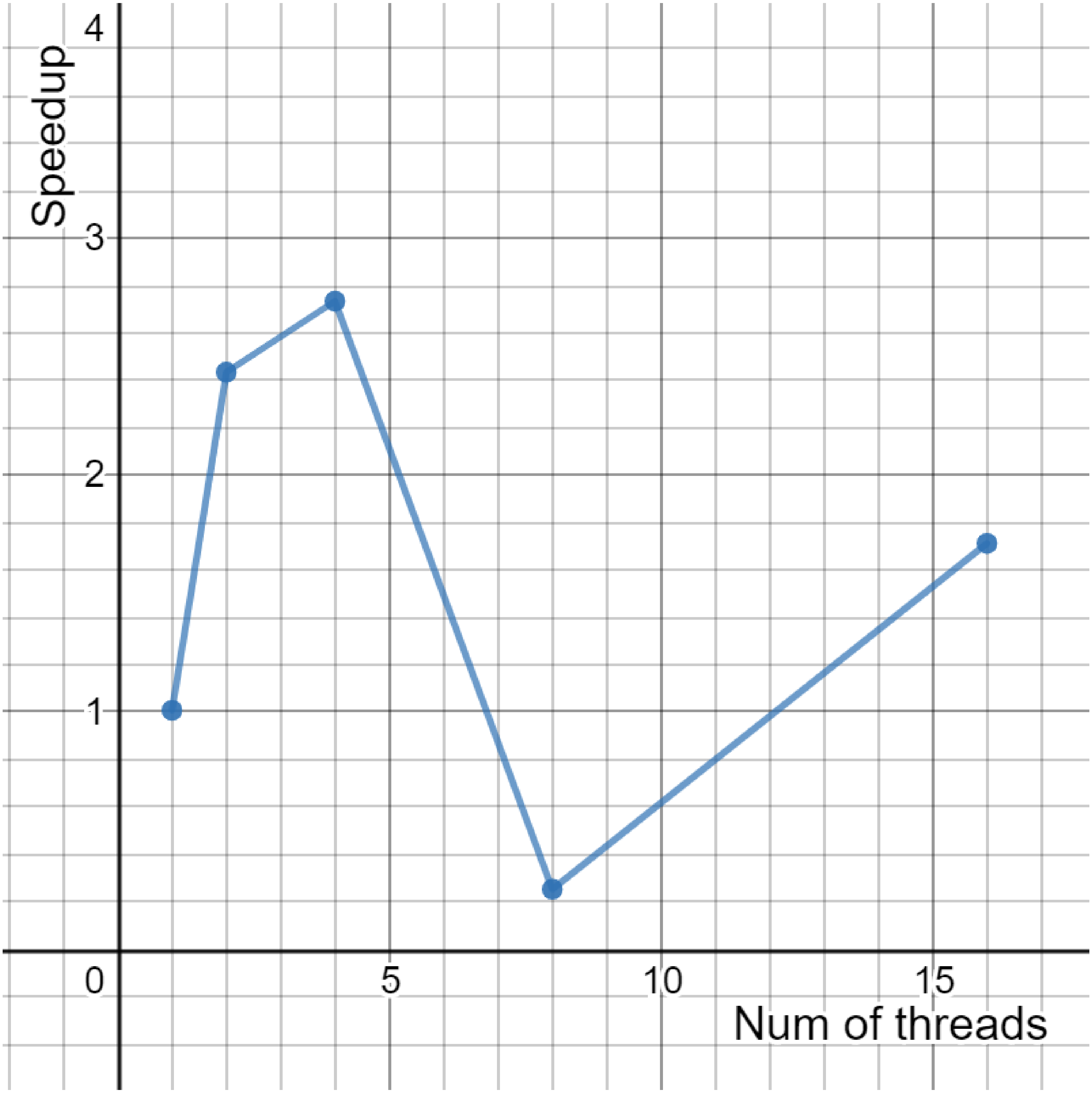
Speedup



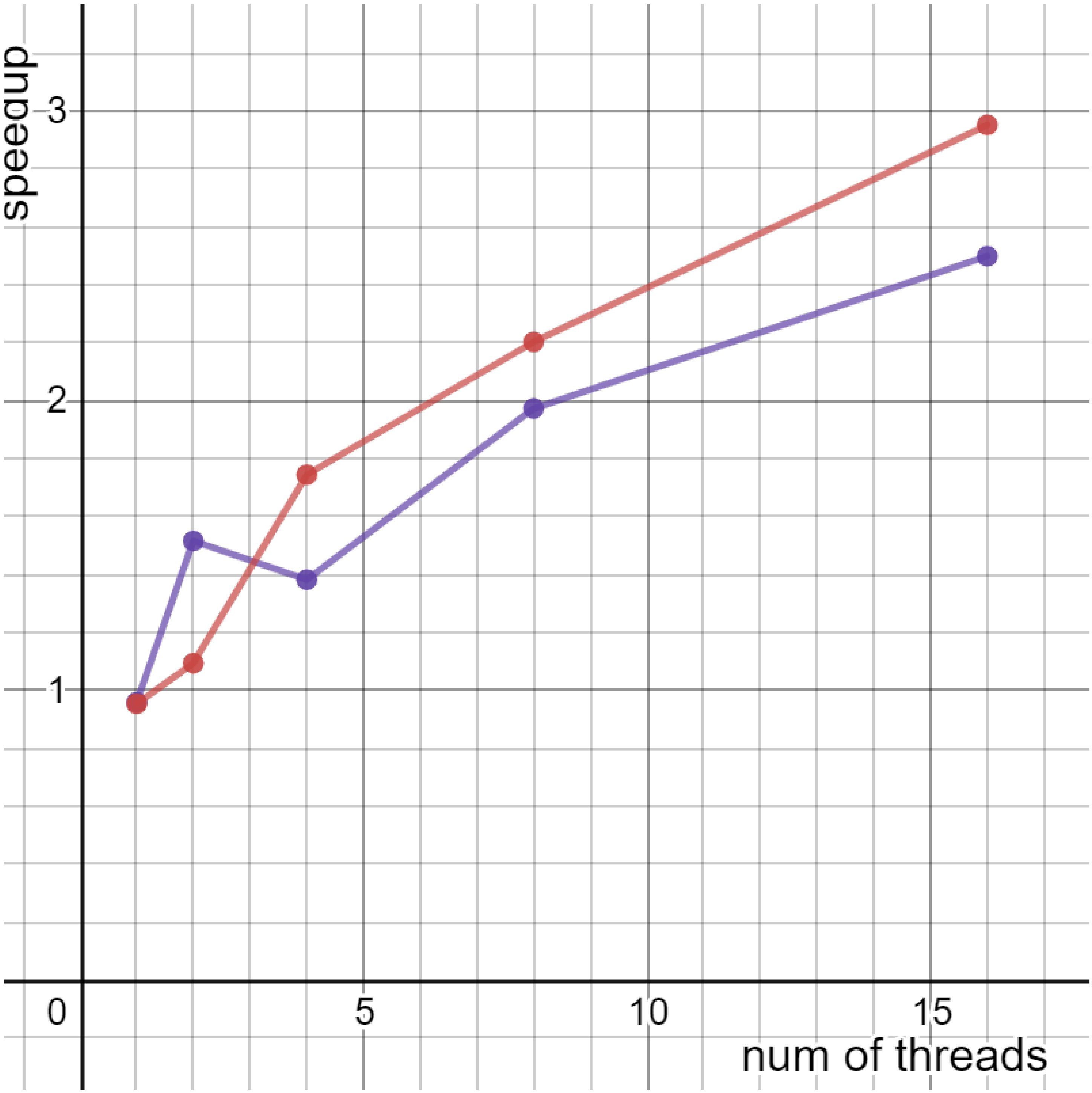
Speedup for test dataset



Speedup WIKi-vote dataset



Speedup for facebook dataset

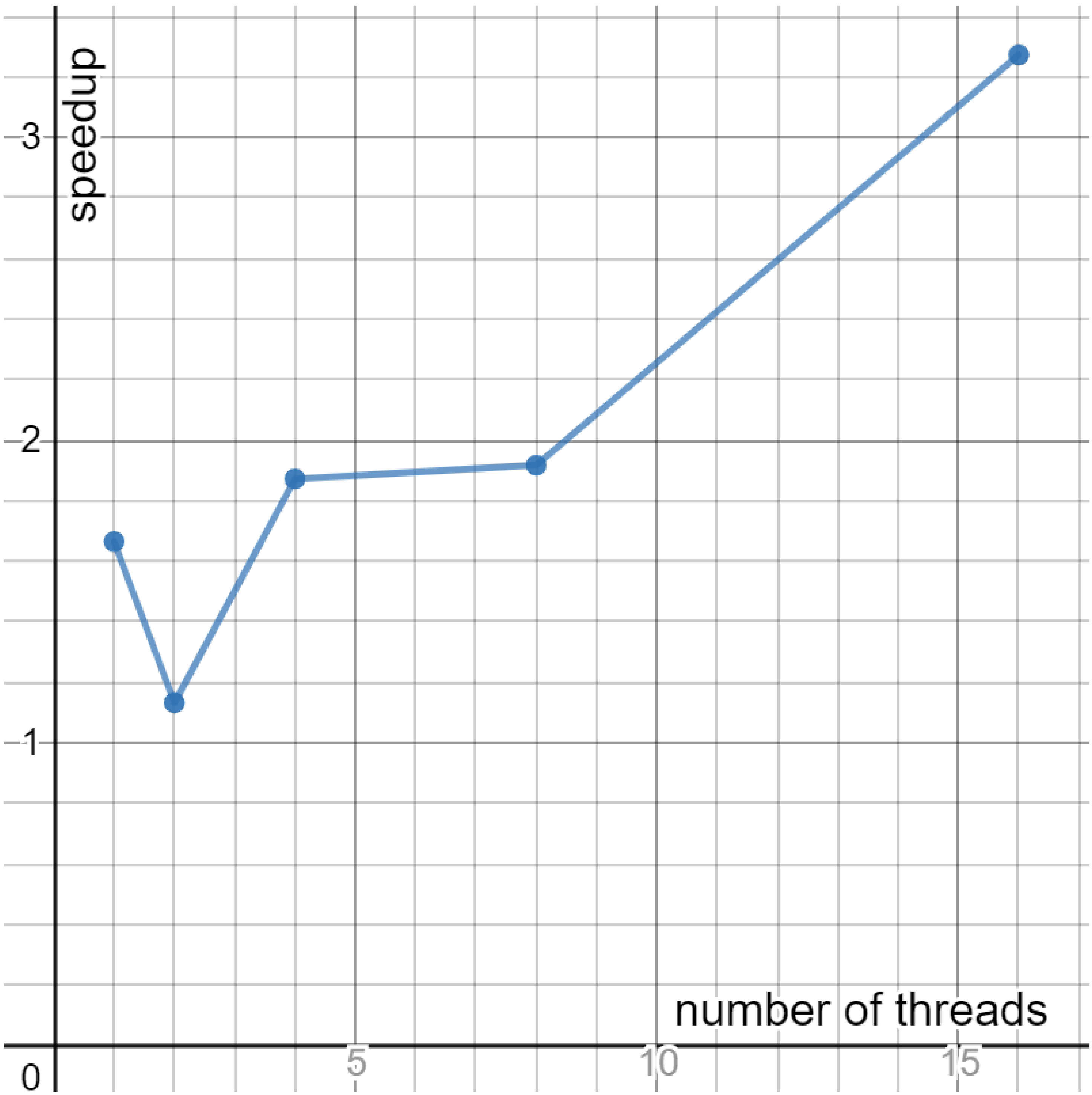


Speedup of RMAT 18 and RMAT 19

RMAT 18-purple

RMAT 19- red

RMAT 20 Speedup as you increase num of threads



SPEEDUP FOR RMAT 22

Conclusions:

As the number of threads increases for a large enough dataset speedup increases.

Notes:

I used Grendel to run the code.

Rmat 21 wasn't running despite trying to make it run numerous times. The terminal would just keep computing and show connection errors from time to time.