ECE506

Architecture of Parallel Computers Machine Problem

MP1 - Stage one Serial, OpenMP

(The whole program is run on Grendel)

APPROACH

As mentioned in the project description, Radix-sort is achieved by making use of the count-sort digit by digit (starting from the ones place). Here, in my approach I have modified the count-sort in a way it takes into account the digit's place. This count-sort is repeatedly called for each digit till the maximum digits ("**numOfVertices**" found at the function loadEdgeArrayInfo).

The algorithm is:

1) Split the 'src' into digits (starting from ones place).
2) Call count sort for the corresponding digit and get it sorted.
3) Pass the output of the sorted 'src' based on the first digit to the second count-sort.
4) Repeat this process until the last digit.

Parallelizing approach

- Here, the count-sort can be parallelized in various instances. All the trivial for loops where initialization or assignment happens can be easily parallelized with ***#pragma omp parallel for*** as they are mutually exclusive iterations (i.e, each loop iteration is independent of each other).
- The main loop where the frequency of each digit is counted cannot be parallelized directly with ***#pragma omp parallel for*** as there exists a *race condition* between the threads for counting since there can only be 10 digits (0-9). So, in order to parallelize this loop, we can put the incrementing operation inside a *#pragma omp critical* section or we need to give each thread its own counter for every digit. Since the for loop runs for huge number of times, putting a critical section inside it will not give us any performance benefit. So as show below, we eliminate the race condition by changing the way the loop is structured, we declare a *vertex_cnt_t[10*n]* where n is the number of threads. (for 4 it is shown below).

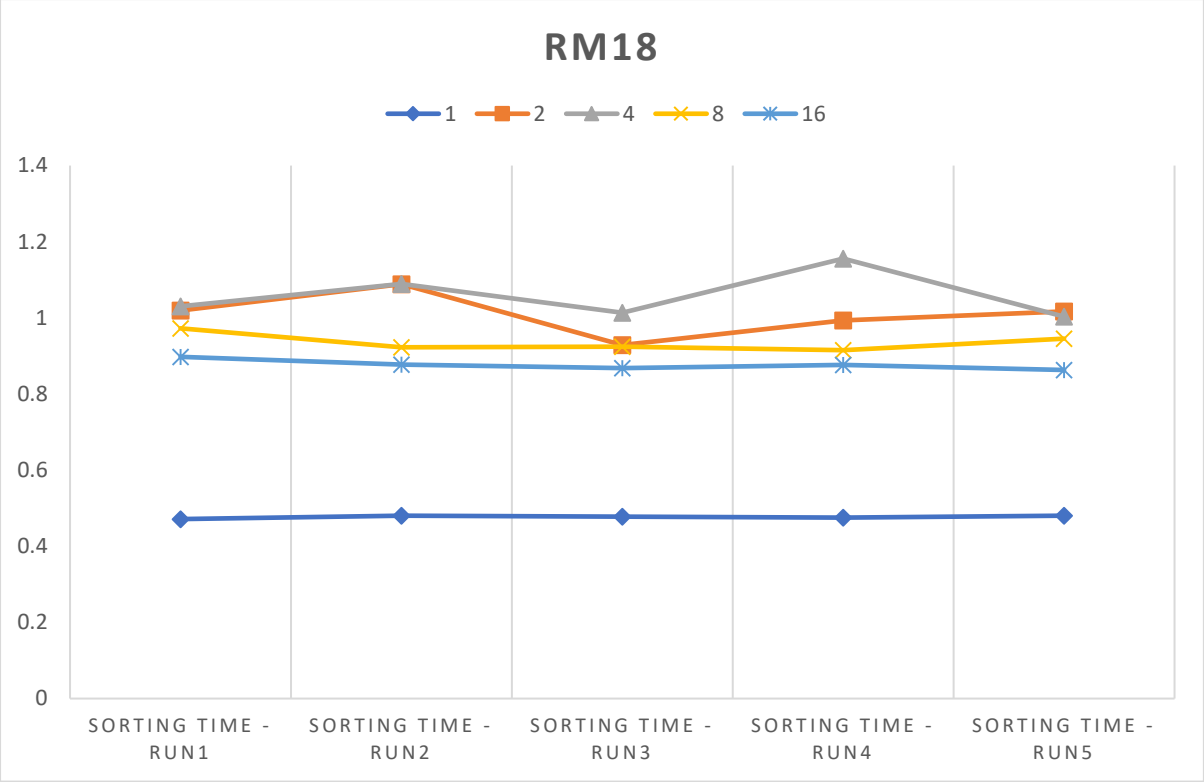
```
int vertex_cnt_t[10*4] = {0}; //this is for the partial-sum for each thread
int vertex_cnt[10] = {0};  //this is for the final sum.
int num_of_threads = 4;
//parallel
#pragma omp parallel for num_threads(num_of_threads) private(i)
for(i = 0; i < numEdges; ++i)
   {
        int num_thread = omp_get_thread_num(); //getting the ID of the thread
        dig = grab_the_digit(edges[i].src, digit);
        key = dig;
        vertex_cnt_t[(10*num_thread)+key]++; //each thread gets to write in its own space without a
//                                                                                          race condition

    }
```

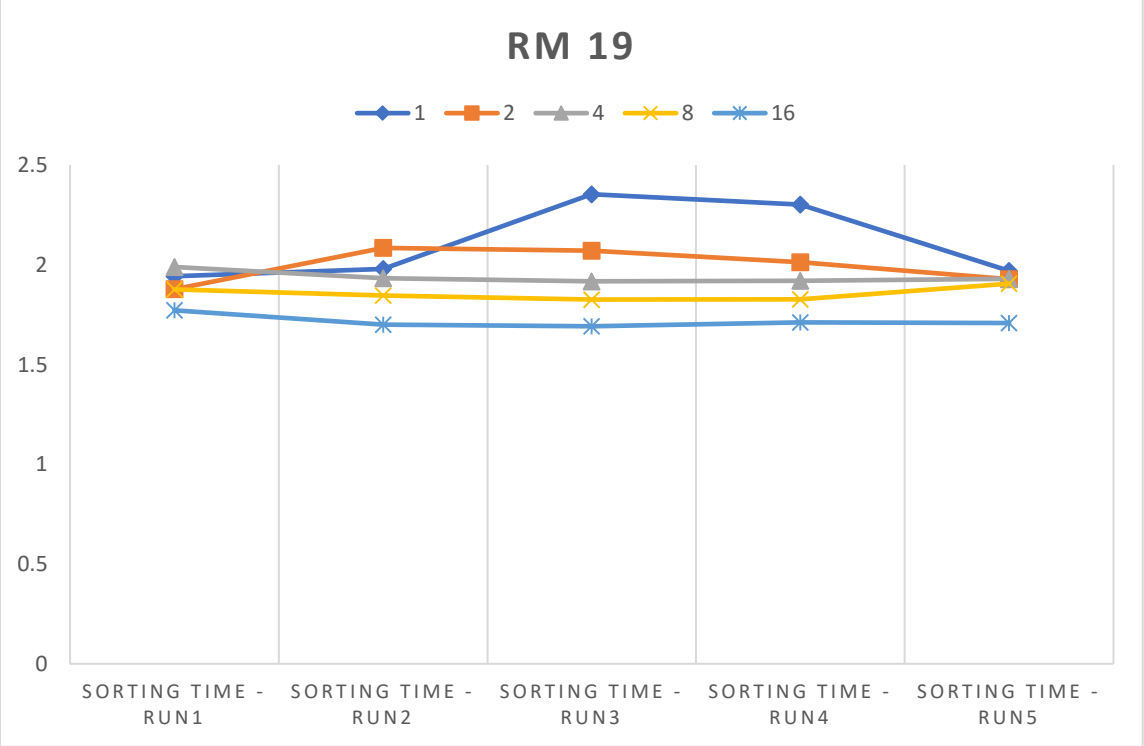
- We can run another trivial parallelized for loop in order to accumulate all the partial sums into a single array that can be used in further calculations.

OBSERVATION

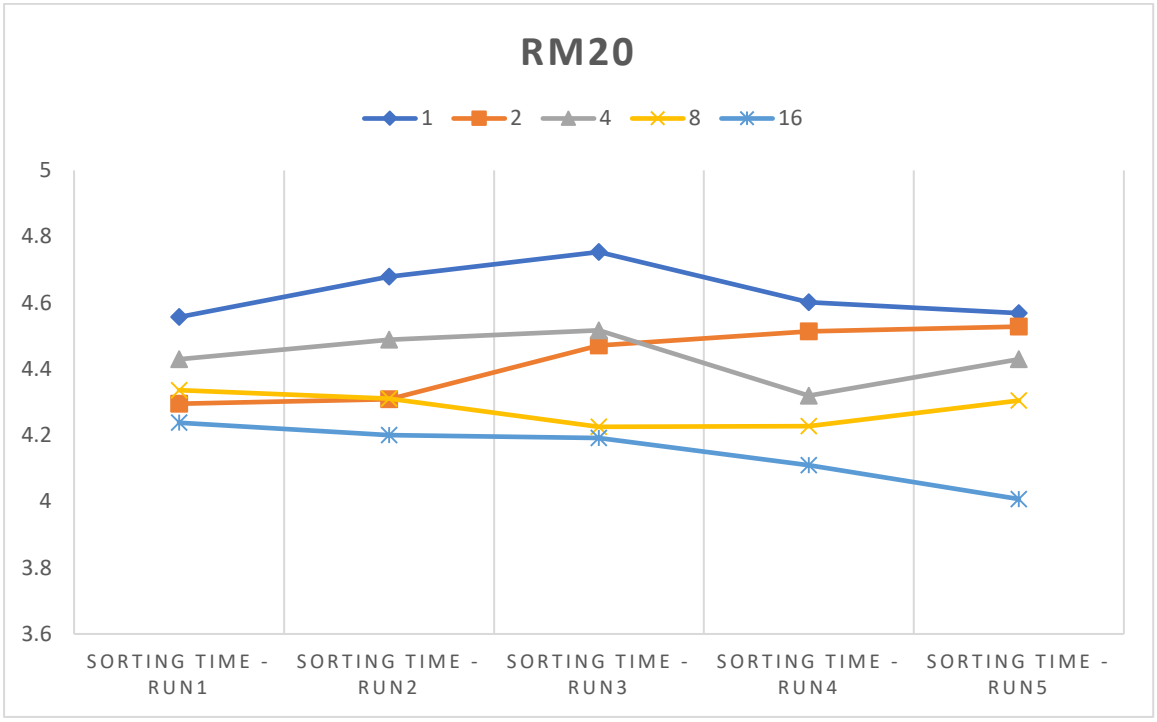
(Each dataset is run with 1, 2, 4, 8 & 16 threads. Each thread has 5 runs)



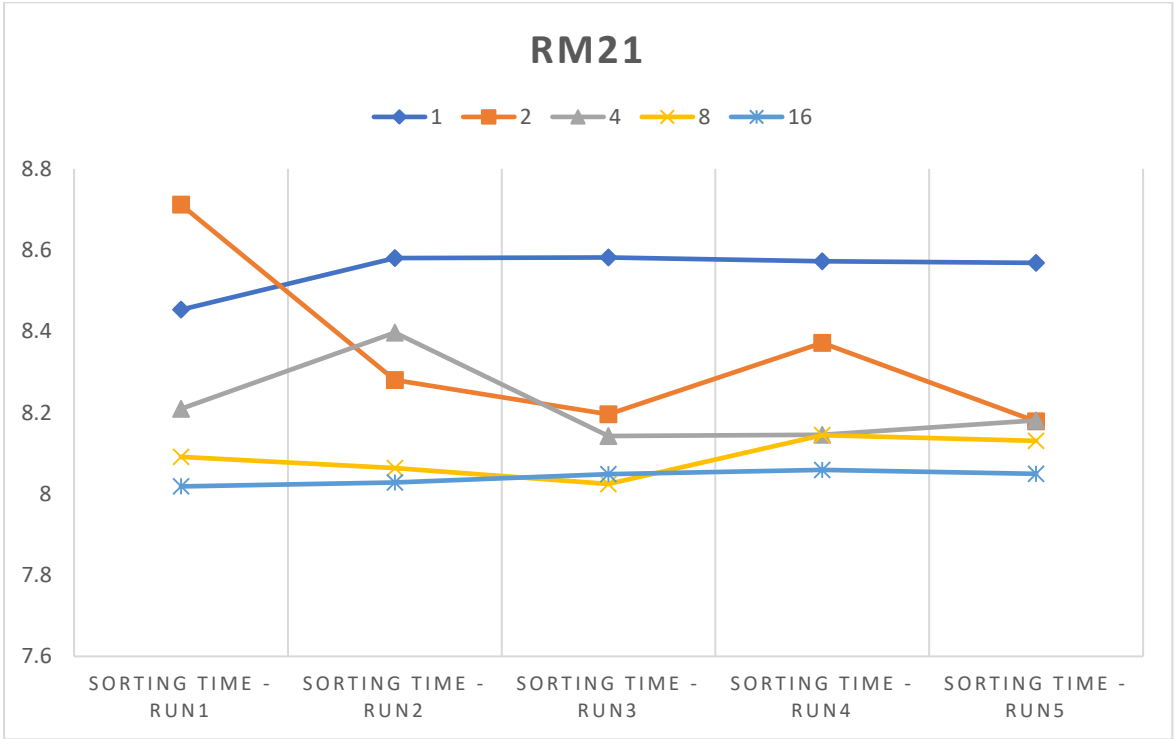
- Here, we observe that the single threaded program runs faster than the program with 2, 4, 8 & 16 threads. I believe that this dataset is too small for us to really amortize the forking and joining overheads that comes with invoking threads. As the dataset is small the overheads overshadow our sorting time.



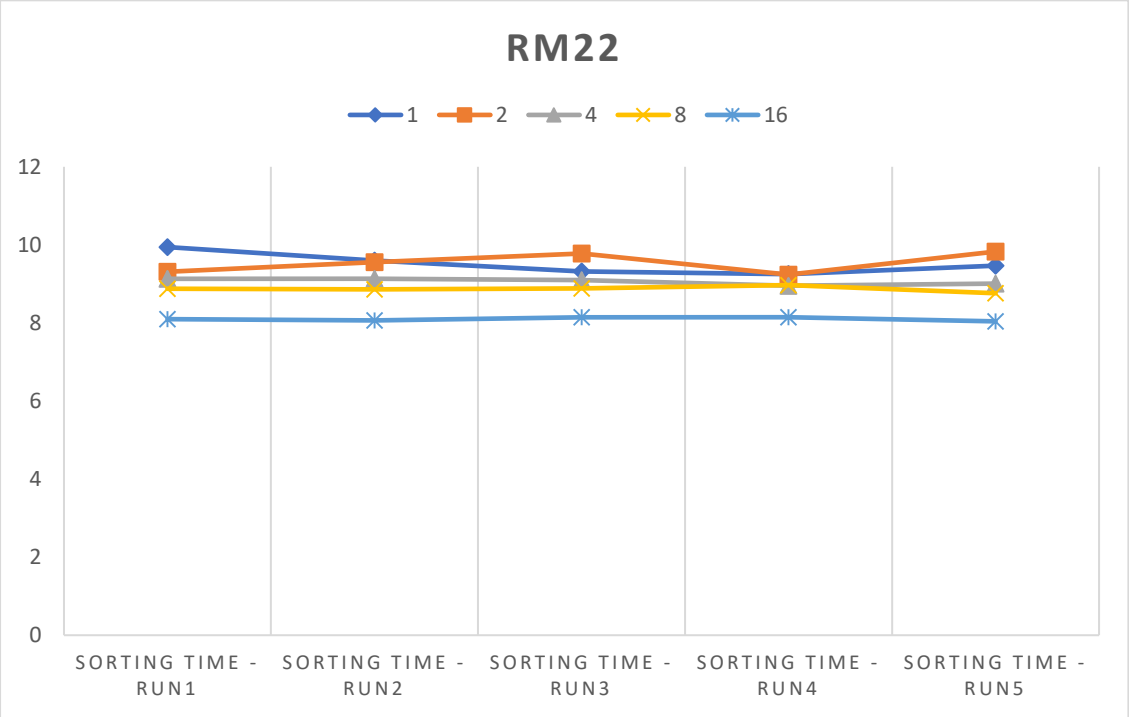
- Here, we start to see the gains of the multi-threaded approach. As you can observe that the 16 threads runs faster than every other number of threads in all 5 runs.



- Similar to RM19, we see that RM20 runs faster on multi-threaded approach. As the size of the dataset increases, we see this trend. Which is pretty intuitive as the for loops that we parallelized are doing so much more work and now since more than one thread is running them (without a race condition and a critical section), we see a steady improvement in performance.



- In RM21, we observe that the performance is still better for 16 threaded program, but there are some anomaly in some runs. This can be due to the dataset itself. This is a good indication that parallelizing a code and checking the performance multiple times for a dataset might still not mean that it will perform well in all situations.



- Finally, RM22 like we expected performs well in multithreaded code. It is clear from the plot that the consistency is something that depends on the way the OS scheduler each thread. It is to be noted and remembered that we cannot and shouldn't benchmark the performance of a code/application on a single run. We need to take multiple samples/runs to generalize the behaviour.

CONCLUSION

1) Applications/programs can be given a performance boost by parallelizing the code. But it is crucial to find out what kind of work is being done by the program. We need to find out the compute-intensive part of the code and first parallelize that to see immediate benefits. (Amdahl's law).
2) It is important to completely understand the logic of the program that you are parallelizing as we will have multiple ways to achieve a parallel code. Blindly plugging in the directives will not really give us good gains.
3) OpenMP is a fork-join model, so invoking and joining the thread will have a cost, so it is critical to amortize that by making maximum use of the threads.
4) By doing this project I observed that, we need to make sure that benchmarking a code is done in a robust manner. That is, we need to run the code for multiple times and with multiple samples. We might get a really good result on the first run; it doesn't mean that it is the general trend. Understanding the underlying hardware and the operating system's policy for scheduling might be handy in making some critical decisions. (number of threads, placement of critical section and so on.)
5) We might always have outliers for a program, that is a parallel code that performs really well for most of the data can still have its outlier. Understanding that is critical to a good parallel code as sometimes serial code runs better.

DATA DUMP:

RM18

# of threads	Sorting time - Run1	Sorting time - Run2	Sorting time - Run3	Sorting time - Run4	Sorting time - Run5	Average
1	0.471507	0.480123	0.477802	0.4753	0.480401	0.477027
2	1.019098	1.088564	0.928714	0.993009	1.016582	1.009193
4	1.030448	1.089577	1.013577	1.155669	1.003244	1.058503
8	0.972485	0.922515	0.924556	0.915123	0.945543	0.936044
16	0.897606	0.876984	0.86815	0.876376	0.863115	0.876446

RM19

# of threads	Sorting time - Run1	Sorting time - Run2	Sorting time - Run3	Sorting time - Run4	Sorting time - Run5	Average
1	1.942377	1.97887	2.353425	2.301921	1.969234	2.109165
2	1.877281	2.08422	2.071154	2.013381	1.928068	1.994821
4	1.988837	1.932636	1.916936	1.919544	1.929457	1.937482
8	1.876412	1.845214	1.825473	1.827055	1.905238	1.855878
16	1.771496	1.700072	1.691612	1.710917	1.707374	1.716294

RM20

# of threads	Sorting time - Run1	Sorting time - Run2	Sorting time - Run3	Sorting time - Run4	Sorting time - Run5	Average
1	4.556893	4.67887	4.753425	4.601921	4.569234	4.632069
2	4.295261	4.308422	4.471154	4.513381	4.528068	4.423257
4	4.430025	4.488813	4.516936	4.319544	4.429457	4.436955
8	4.336201	4.310342	4.225473	4.227055	4.305238	4.280862
16	4.237821	4.200072	4.191612	4.10917	4.007374	4.14921

RM21

# of threads	Sorting time - Run1	Sorting time - Run2	Sorting time - Run3	Sorting time - Run4	Sorting time - Run5	Average
1	8.45317	8.580141	8.581722	8.572466	8.56882	8.551264
2	8.711638	8.27993	8.19598	8.371828	8.178314	8.347538
4	8.208849	8.39685	8.142378	8.145514	8.180773	8.214873
8	8.091407	8.063646	8.024416	8.144367	8.130221	8.090811
16	8.018301	8.027715	8.048244	8.059016	8.049305	8.040516

RM22

# of threads	Sorting time - Run1	Sorting time - Run2	Sorting time - Run3	Sorting time - Run4	Sorting time - Run5	Average
1	9.942636	9.594695	9.316367	9.249497	9.463937	9.513426
2	9.310459	9.556799	9.779543	9.237862	9.827415	9.542416
4	9.128991	9.132105	9.09468	8.953914	9.011064	9.064151
8	8.873145	8.863151	8.880642	8.969764	8.760609	8.869462
16	8.09622	8.066552	8.144411	8.143558	8.04013	8.098174