Parallel K Nearest Neighbours using OpenMP

Michael Beukman

July 28, 2021

Contents

Introduction

K Nearest neighbours (KNN) is an intuitive algorithm that can be used to perform classification [1], or regression [2].

The idea behind KNN is that we are given a list of reference points in some space where each point either has a corresponding label (classification) or a corresponding real valued property (regression). Using this, we can predict the label or property value of any query point by using information from its closest k neighbours. [1]

This could either be to assign the query point the label that most of its neighbours have in the case of classification or to assign the query point's property value to an average (could be weighted by distance) of its nearest neighbours when doing regression.

The way nearest is defined can be problem specific, but in general any distance metric (for example the Manhattan or Euclidean distances) can be used.

In the following discussion, we will refer to M (m) as the number of reference (training) points and N (n) as the number of query points. D (d) will denote the number of dimensions each point has.

There are many ways to perform KNN, but we will be using the brute force method. The general structure of this algorithm is as follows:

For each query point q_i, calculate the distance between q_i and each reference point p_j, $0 <= j < M$. Then sort these distances and choose the points corresponding to the k smallest distances.

Finally perform prediction using those points.

Since we are only focused on the performance and benchmarking of the algorithm, the prediction step is largely ignored.

The main computationally intensive parts are the distance calculation, and the sorting time. They have complexities of $O(nmd)$ and $(nm\log m)$, respectively (the sorting complexity depends on the exact method one uses)

Uses

KNN can be used for many different applications, for example:

- Classification in Machine learning [1].
- Regression in Machine learning [2].
- To determine the 'novelty' of an indivdual when using novelty search in the case of genetic algorithms [3].

Methodology

In the following sections we describe the general structure of the code as well as the main parallelisation techniques used.

Code

Here we show the main compute intensive parts of the algorithm, the distance computation and the subsequent sorting. We used C++ for all of the algorithms and methods.

Note: In the actual code, when sorting, we create a structure consisting of a double (the distance) and an integer (the index). This is used to get the actual points that had the minimum distance at the end. We could just use the distances, but the two main problems with that is:

- This would require another linear search through all the points to determine which ones had which distances. That would make a relatively significant part of the total time be taken up by this linear search instead of the actual distance calculation/sorting.
- Since the distances aren't necessarily unique, one would not always be sure in exactly which order the points came, which will make validation more complex than it needs to be.

For the above reasons, we decided to sort the <distance, index> pair instead of just the distance. This does add some overhead to the sorting algorithm, but we decided it was the best compromise.

```
    // distance calculation
    for (int i=0; i<n; ++i){
        for (int j=0; j<m; ++j){
            dist_array[i][j] = dist(points[j], query_points[i]);
        }
    }
    // sorting
    for (int i=0; i<n; ++i){
        sort(dist_array[i]);
    }
```

Here we show a snippet of how we parallelise the code.

Parallel Code

```
    // distance calculation parallel
    #pragma omp parallel for
    for (int i=0; i<n; ++i){
        for (int j=0; j<m; ++j){
            dist_array[i][j] = dist(points[j], query_points[i]);
        }
    }
    // sorting
    for (int i=0; i<n; ++i){
        parallel_sort(dist_array[i]);
    }
```

Parallelisation

As seen in the above code, the distance calculation is performed using 2 for loops. The outer one iterates over all of the query points and the inner one iterates over all of the reference points. We chose to only parallelise the outer for loop for two main reasons:

- Emperically, when testing on smaller cases this performed better than parallelising the inner loop.
- If we parallelise the inner for loop, we could fall victim to false sharing because it is likely that different threads will share the same cache line when parallelising the inner for loop. When instead using the outer for loop, the array elements of different threads are more spread out, so this doesn't fall victim to the same issue.

Thus, for the distance calculation, we simply use the omp command *pragma omp parallel for* for the outer for loop. We used the default scheduling, which is implementation defined, but prior testing showed that the type of scheduling used didn't affect the performance significantly. This could be because we didn't use a lot of threads.

For the sorting, we implemented 3 different types of sorting algorithms (merge sort, bitonic sort and quick sort). For each of these three, we implemented a serial version, a parallel version that uses the *omp task* construct and a final parallel version that uses the *omp sections* construct. We use OpenMP for all of the parallelisations.

We implemented all of the above sorting functions recursively, so the way we parallelise them is quite similar. It consisted of assigning one task or section to a single recursive call. Shown here is the implementation for merge sort. The sections version is similar to the task version, except that we create a parallel region (with two threads) first. The parallel region for the tasks is created by the function that calls *par_merge_sort_tasks_inner* the first time.

For the parallel sort, we had to determine the *low limit*, which determines the smallest array size to actually perform more parallelism rather than deferring to the sequential function (i.e. for an array smaller than the low limit, we just call the sequential version of the sorting algorithm). We empirically tested many different values and found that for tasks a good value was 1000 and for sections, 10000. We opted to go for these constant values for all the sorting algorithms instead of something dependent on the input size to keep everything consistent and easily comparable. It is very likely that there exist better values/formulas than these, but we opted to go with what we did for simplicity. The reason for the sections' larger value is potentially that at every recursive step, we double the number of threads used (because we create two new threads at each step). So the number of threads used is very large when using a small low limit. Tasks don't have this issue, because tasks are generated on one thread, and then multiple threads can perform the tasks until there are no more remaining.

```
1  void seq_merge_sort_inner(int l, int r, T* data, T* temp_data) {
2      // sorts in the range [l, r) (i.e. not including r)
3      // if (r <= l)
4
5      if (r - l <= 1)  // if this is the case, our array has a size of 1 or
           less.
6          return;
7      int mid = (l + r) / 2;  // middle
8
9      seq_merge_sort_inner(l, mid, temp_data, data);
10     seq_merge_sort_inner(mid, r, temp_data, data);
11     // and using the sorted sub arrays of temp data, we merge them back
           into data.
```

```
    merge(l, mid, r, temp_data, data);  // output into data.
}

void par_merge_sort_tasks_inner(int l, int r, T* data, T* temp_data, int
    low_limit) {
    if (r - l <= 1)
        return;
    if (r - l < low_limit) {
        // use sequential version if the task size is small enough
        return seq_merge_sort_inner(l, r, data, temp_data);
    }
    int mid = (l + r) / 2;  // middle
#pragma omp task
    par_merge_sort_tasks_inner(l, mid, temp_data, data, low_limit);
#pragma omp task
    par_merge_sort_tasks_inner(mid, r, temp_data, data, low_limit);
#pragma omp taskwait
    merge(l, mid, r, temp_data, data);  // output into data.
}
```

Experiments

We ran experiments by varying different aspects, and measuring the time for the distance calculation, the time for the sorting and the total time.

Specifically, we had the following different methods/functions:

- For the distance calculation, we used both the Manhattan distance as well as the Euclidean distance (L1 and L2 norms respectively).
- For the sorting, we tested sequential versions of quick sort, merge sort and bitonic sort. For each of these, we also implemented two parallel versions. One used the *omp task* construct and the other used the *omp sections* construct.
- We used both parallel and sequential versions of the distance calculation.
- We used many different values for M, N and D.

We then compared the results of each of the above with our sequential reference implementation to make sure that our results were correct with all methods. We measured 3 different times to judge the different methods:

- The time to do the distance calculation for all the query points.
- The time to sort the distances for all query points.
- The total time taken for the whole algorithm. This was mainly to see if any extra part (e.g. memory allocation) had a big effect on the total time taken.

All our experiments were run 10 times and averaged over those runs. This was done so that the results are reliable and any randomnesss reduced. We used a constant $k = 10$ for all our experiments, but this doesn't impact the performance.

Every run was verified by comparing the results to those of a simple, serial reference implementation. Other verification was also done (e.g. ensuring the distance array is actually sorted).

We ran the experiments on a *Intel(R) Core(TM) i7-7700 CPU @ 3.60GHz* running Ubuntu 18.04 with 4 cores. We used 2, 4, 8 and 16 threads for to see how the results compare when scaling up the number of threads used. We used *g++ 7.5.0* as our compiler with flags *-lm -O3 -fopenmp*.

Very few other proccesses were running on the machine during our experiments.

When comparing the speedup of the parallel versions to their sequential counterparts, we usually compare the same underlying algorithm (e.g. when measuring the speedup of parallel bitonic sort, it's relative to sequential bitonic sort.). This is just to make a fair comparison.

The data used was randomly generated since we used such a large range of different parameters. We seeded the initial random generator to get roughly reproducible results. The time taken to generate this data was not measured. We generated the data 10 times (once for each iteration) and evaluated each method on the same data to be able to make fair comparisons.

Results

We represent the result in a few distinct sections for clarity and simplicity. We performed the data analysis using Python

We will first examine the distance computation, then the sorting and finally the total time taken. For each metric, we will plot 3 graphs. These will show the time taken by different techniques as a function of M, or N or D, with the other 2 being kept constant. For the next 3 sections, the results were achieved when using 4 threads.

Note: we denote the time taken by the distance calculation as *Dist (ms)*, and similarly for the other cases. The *ms* denotes the units of time used, in this case milliseconds.

Distance Calculation

Here we plot the time taken to perform the distance calculation for a constant sorting function.

As we can see, which distance function (Manhattan vs Euclidean) we used didn't impact the results much. For clarity, we will thus only show the results obtained using Euclidean distance in the subsequent sections. In general, the distance calculation is linear in N, M and D, which makes sense since in effect we iterate over all of the query points, then all reference points and finally over all dimensions. The parallel distance calculation consistently performed much better than the serial version and achieved a speedup of between 3 and 3.5 times. The speedup is mostly constant with respect to M, N and D.

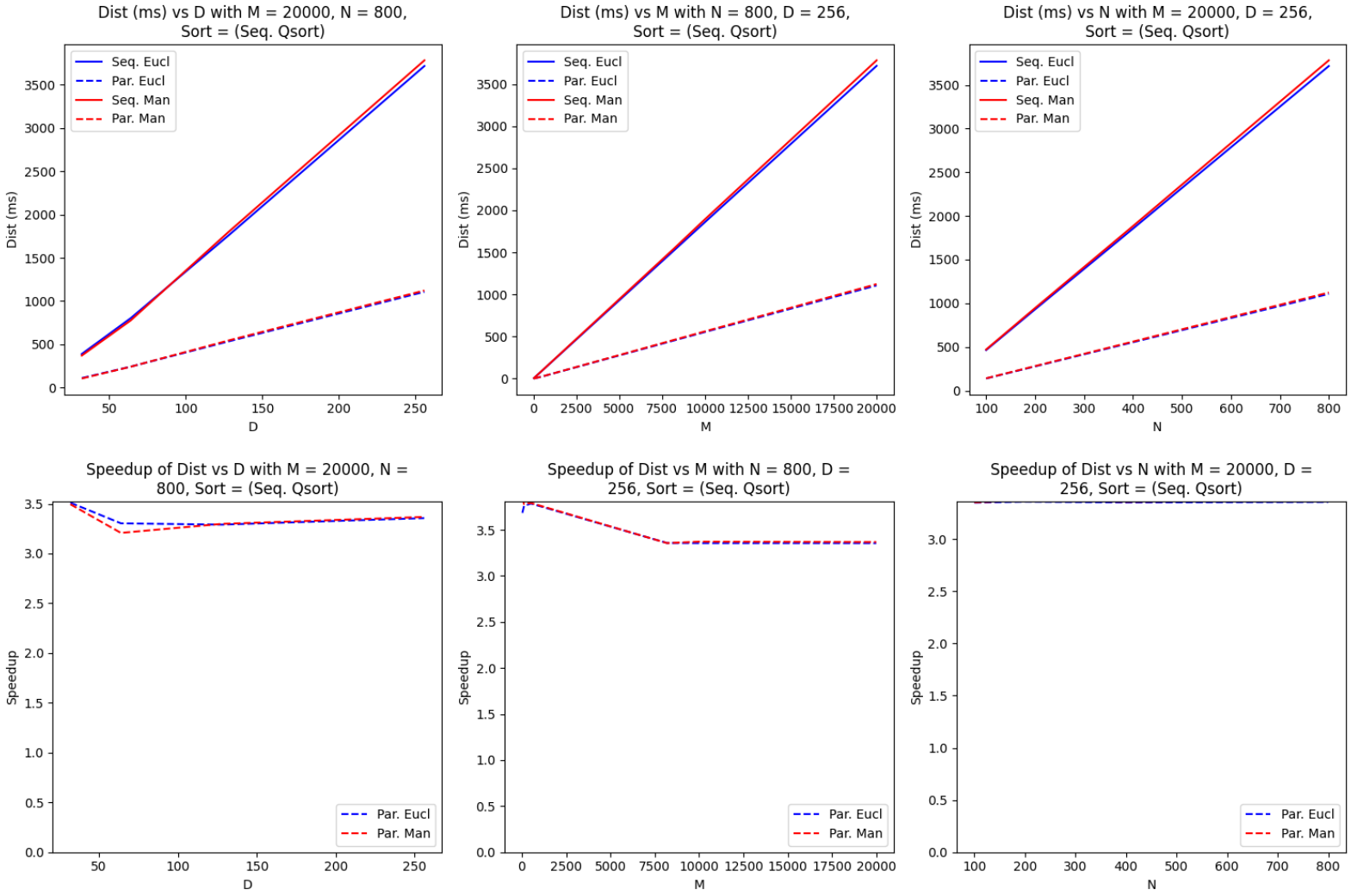


Figure 1: Time taken to compute distances as a function of N, M and D.

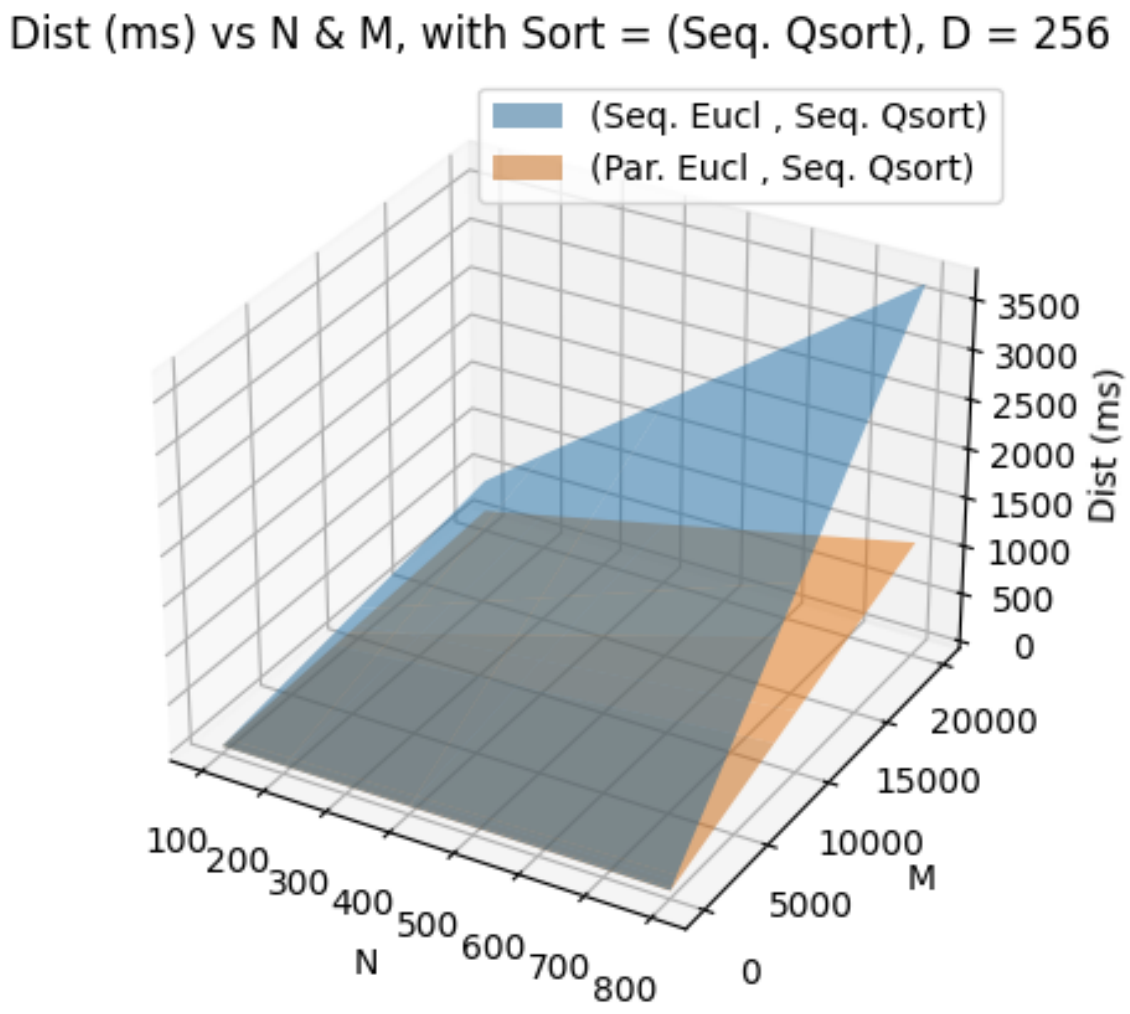


Figure 2: A 3d plot of M and N vs distance calculation time. We see that the distance time is proportional to both M and N

Sorting

Here we show the performance of different sorting algorithms and their parallel versions. We colour coded the lines in the following graphs. The colours are as follows:

- Quick Sort = Blue

- Merge Sort = Red
- Bitonic Sort = Green

And we used different line styles for the sequential and different parallel versions:

- Sequential ——
- Tasks - - -
- Sections ·····

In the below graphs, we see a few interesting things: Sorting is constant with respect to D, which makes sense because the sorting only iterates over the distances, which are single scalar numbers. We can see that the speedup is generally proportional to our M value, and constant with respect to N. As a function of N and M, the complexity of our sorting algorithms look linear, but they are in fact $O(m\log(m))$ (for merge sort [4]), $O(m^2)$ or $O(m\log m)$ (for quick sort, worst case and average case respectively [5]) and $O(m\log^2(m))$ for bitonic sort [6]. The reason for the seeming linearity is possibly that our values for M were small, and the sorting algorithms only sorted arrays of size M.

In general, we can see that quicksort is substantially faster than the other cases, with sequential quick sort even beating bitonic sort's parallel performance. Using tasks performed better than sections in general. The speedup achieved ranged from 0.5 to 3 with quick sort using tasks again proving to be best. Merge sort took second place as far as speedup and time taken is concerned. Bitonic sort achieved a respectable speedup for both sections and tasks, but it was still the slowest algorithm.

Quick sort using sections performed did not perform very well and this can be attributed to using an ineffective low limit.

Merge sort using sections had similar performance as when using tasks, but slightly worse. The speedups achieved from merge sort and bitonic sort are quite similar, but since sequential merge sort had a lower absolute time, mergesort still vastly outperforms bitonic sort.

The sharp corner in some of those lines can be attributed to using a section low limit of 10000, as it only parallelised for arrays larger than that.

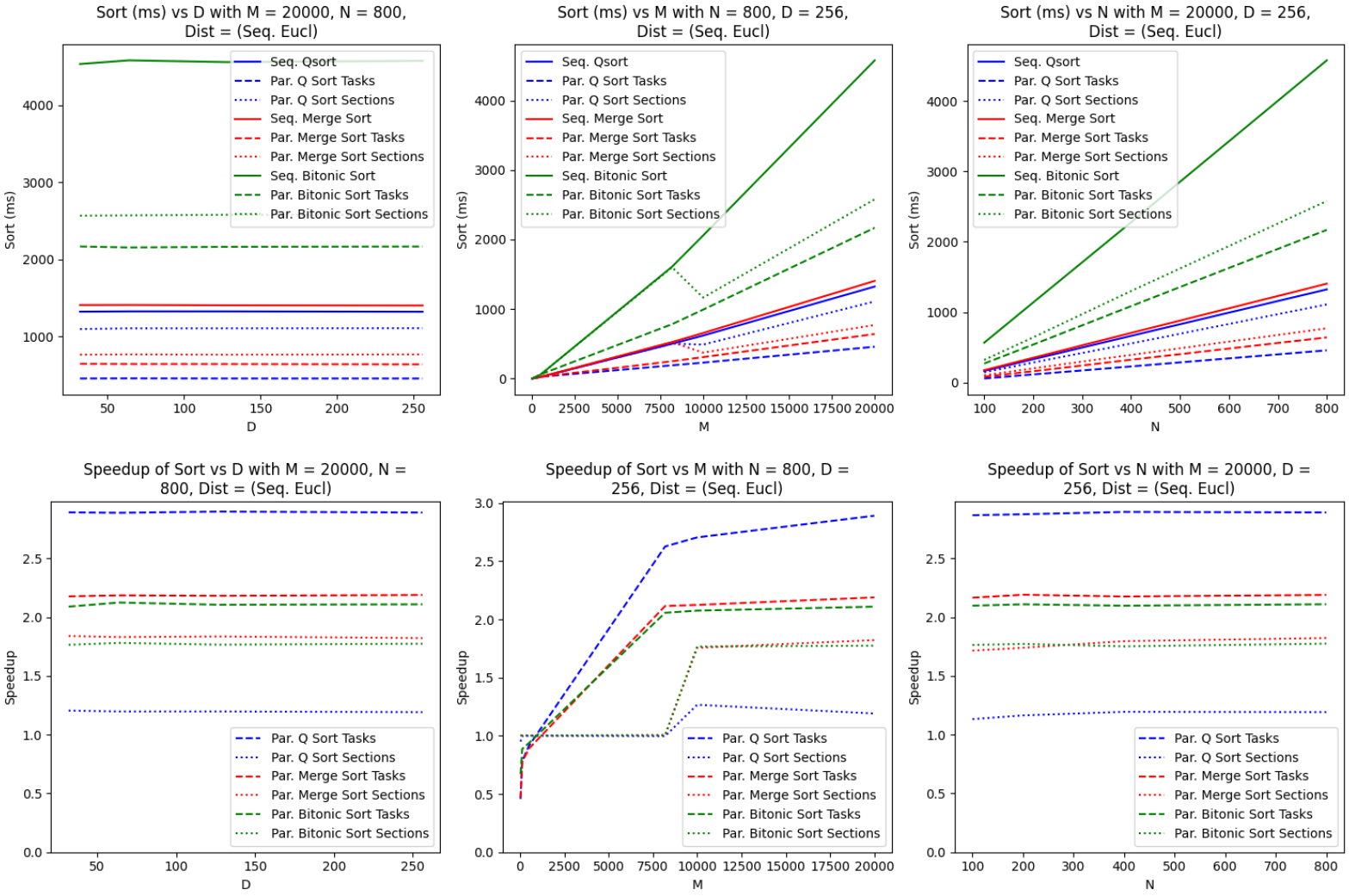


Figure 3: Time taken to sort the computed distances as a function of N, M and D. Speedups are relative to the sequential version of the same algorithm

Total Time

Here we plot only the runs where both parts were parallelised (i.e. both parallel distance and parallel sorting), because there are many different combinations and displaying all of them at once makes seeing anything meaningful difficult. We can see that the total time is roughly linear in M, N and D. Using the parallel distance computation in conjunction with parallel qsort (tasks), we see that we achieve a speedup of around 3 times for the largest cases (which is pretty good since we only used 4 threads). Even the worst parallel algorithm achieves a speedup over its sequential counterpart for some values of the parameters.

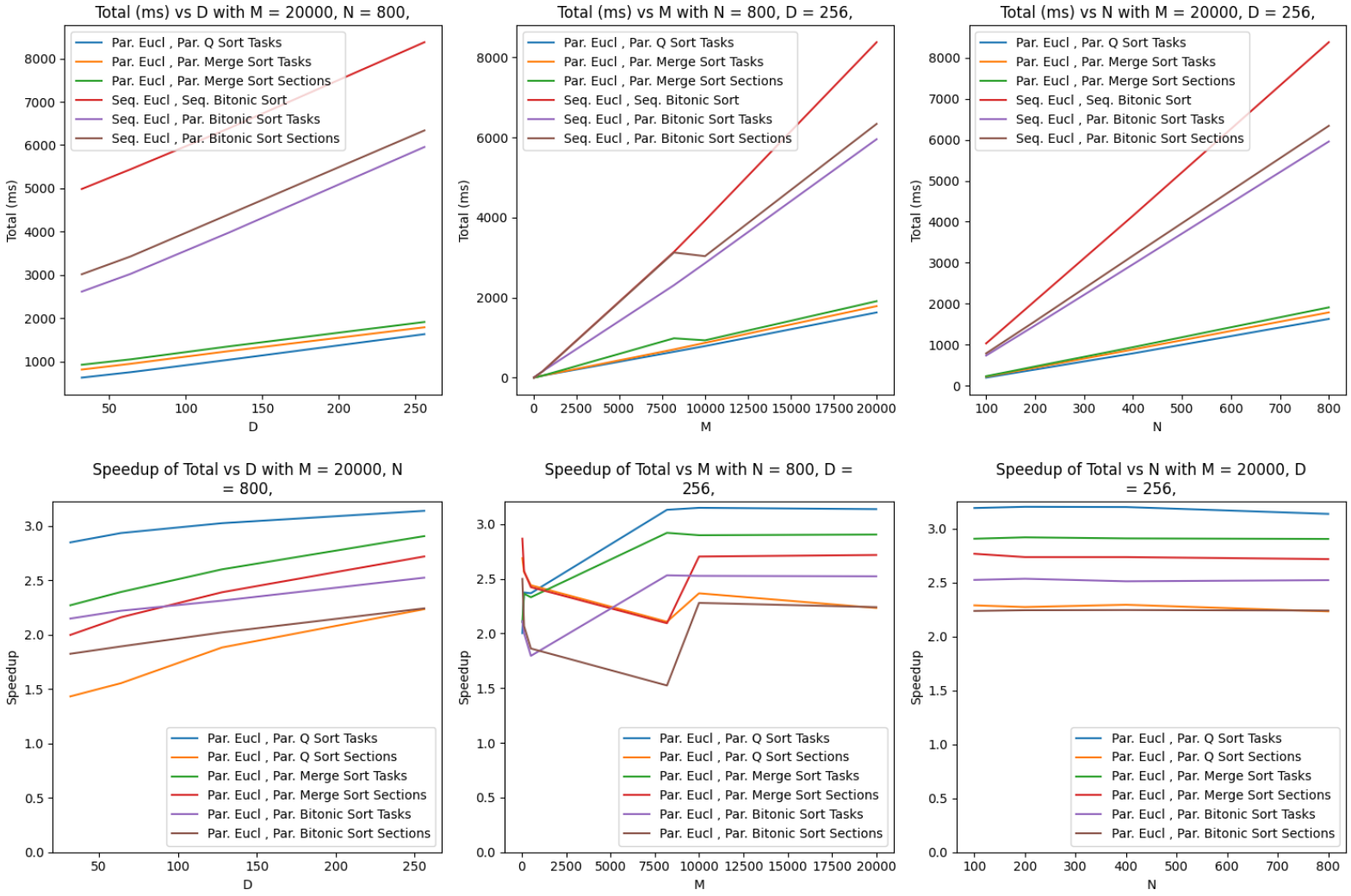
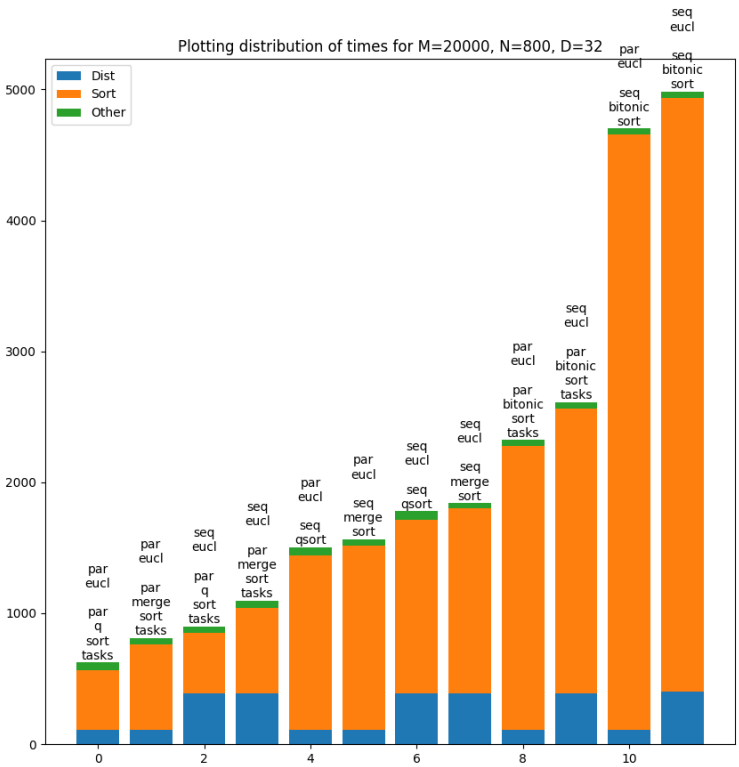
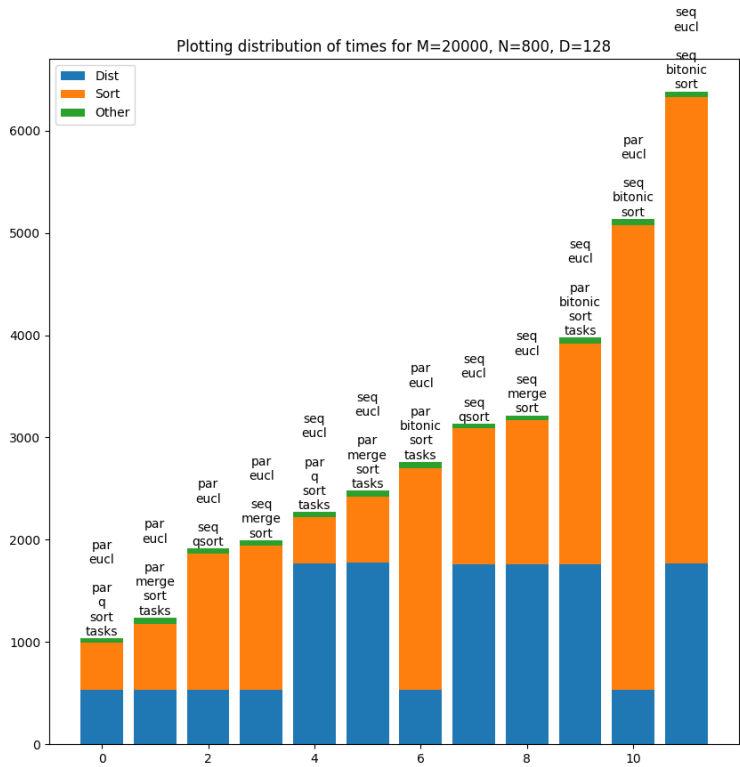
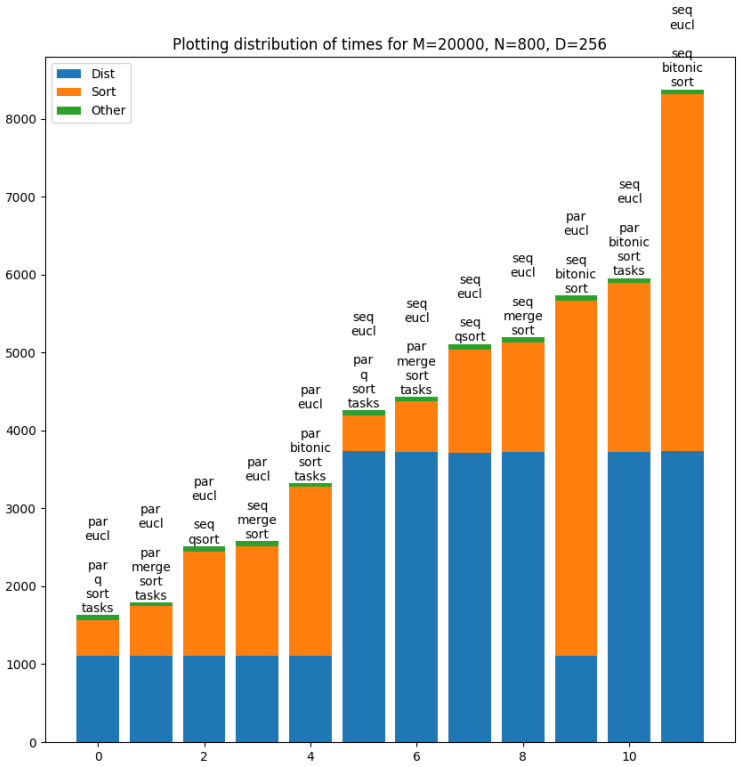


Figure 4: Total time taken for the whole algorithm as a function of N, M and D. The speedups are relative to the same methods used, but just sequentially (e.g. the speeup of bitonic sort with tasks is relative to the sequential bitonic sort run)

In the bar charts shown below, we see how much of the time each part took up. In general, with a small D value, sorting took up most of the time. When D was large, however, this is reversed and the distance computation takes the most time. When averaging the times taken over all the combinations of N, M and D the time taken is roughly balanced and is more dependent on which part was parallelised. As expected, only a very small fraction of the time is spent outside the two main computationally intensive portions.







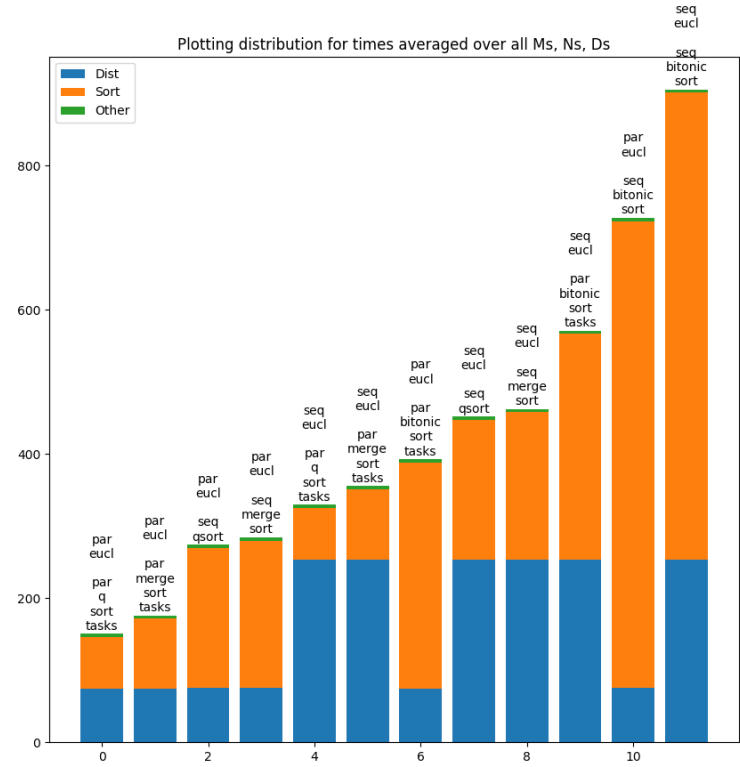


Figure 5: The distribution of time across different parameters

Name	Speedup Dist	Speedup Sort	Speedup Total	Dist (ms)	Sort (ms)	Total (ms)	Dist %	Sort %
(Par. Eucl , Par. Q Sort Tasks)	3.35	2.90	3.14	1109	456	1630	68.0	28.0
(Par. Eucl , Par. Merge Sort Tasks)	3.37	2.20	2.90	1108	640	1788	62.0	36.0
(Par. Eucl , Par. Merge Sort Sections)	3.37	1.86	2.72	1108	756	1911	58.0	40.0
(Par. Eucl , Par. Q Sort Sections)	3.35	1.20	2.23	1110	1107	2289	48.0	48.0
(Par. Eucl , Seq. Qsort)	3.36	0.99	2.03	1107	1337	2516	44.0	53.0
(Par. Eucl , Seq. Merge Sort)	3.37	1.00	2.02	1108	1410	2576	43.0	55.0
(Par. Eucl , Par. Bitonic Sort Tasks)	3.37	2.11	2.52	1108	2166	3321	33.0	65.0
(Par. Eucl , Par. Bitonic Sort Sections)	3.37	1.78	2.24	1108	2571	3739	30.0	69.0
(Seq. Eucl , Par. Q Sort Tasks)	0.99	2.89	1.20	3735	458	4259	88.0	11.0
(Seq. Eucl , Par. Merge Sort Tasks)	1.00	2.19	1.17	3729	641	4430	84.0	14.0
(Seq. Eucl , Par. Merge Sort Sections)	1.00	1.82	1.14	3734	771	4558	82.0	17.0
(Seq. Eucl , Par. Q Sort Sections)	1.00	1.19	1.04	3728	1110	4904	76.0	23.0
(Seq. Eucl , Seq. Qsort)	1.00	1.00	1.00	3716	1323	5112	73.0	26.0
(Seq. Eucl , Seq. Merge Sort)	1.00	1.00	1.00	3729	1405	5193	72.0	27.0
(Par. Eucl , Seq. Bitonic Sort)	3.37	1.00	1.46	1108	4558	5731	19.0	80.0
(Seq. Eucl , Par. Bitonic Sort Tasks)	1.00	2.11	1.41	3727	2169	5955	63.0	36.0
(Seq. Eucl , Par. Bitonic Sort Sections)	1.00	1.77	1.32	3724	2579	6338	59.0	41.0
(Seq. Eucl , Seq. Bitonic Sort)	1.00	1.00	1.00	3734	4578	8377	45.0	55.0

Table 1: Table comparing different methods and their speedups. These results are with M = 20000, N = 800, D = 256 and 4 threads.

This table gives us an overview of our results. We can see that the parallel distance methods had the most success, and parallel quick sort (using tasks) was by far the best sorting algorithm. For the first two places, we can see that the distance calculation took more than 60% of the time. In general, the distance calculation took up the most time.

Using a different number of threads

Here we compare results when running the same workload and the same algorithm (quick sort using tasks and parallel euclidean distance) across a different number of threads. We can see that the number of threads does not massively impact the sorting speedup (from 2 threads to 8 it's about a 1.5 increase). The reason for this could be that M was too small to sufficiently make use of the increased number of threads. It is also possible that our values for the *low limit* was not optimal. We can see that the speedup generally increases up to the number of hyperthreads the machine has (8 in this case). Using any more threads causes a significant performance loss. When using $d = 256$, the time required to do the distance calculation was much larger than the sorting time. When $d = 128$, however, their times are much closer together. We can see the main speedup of the algorithm comes from the distance parallelisation.

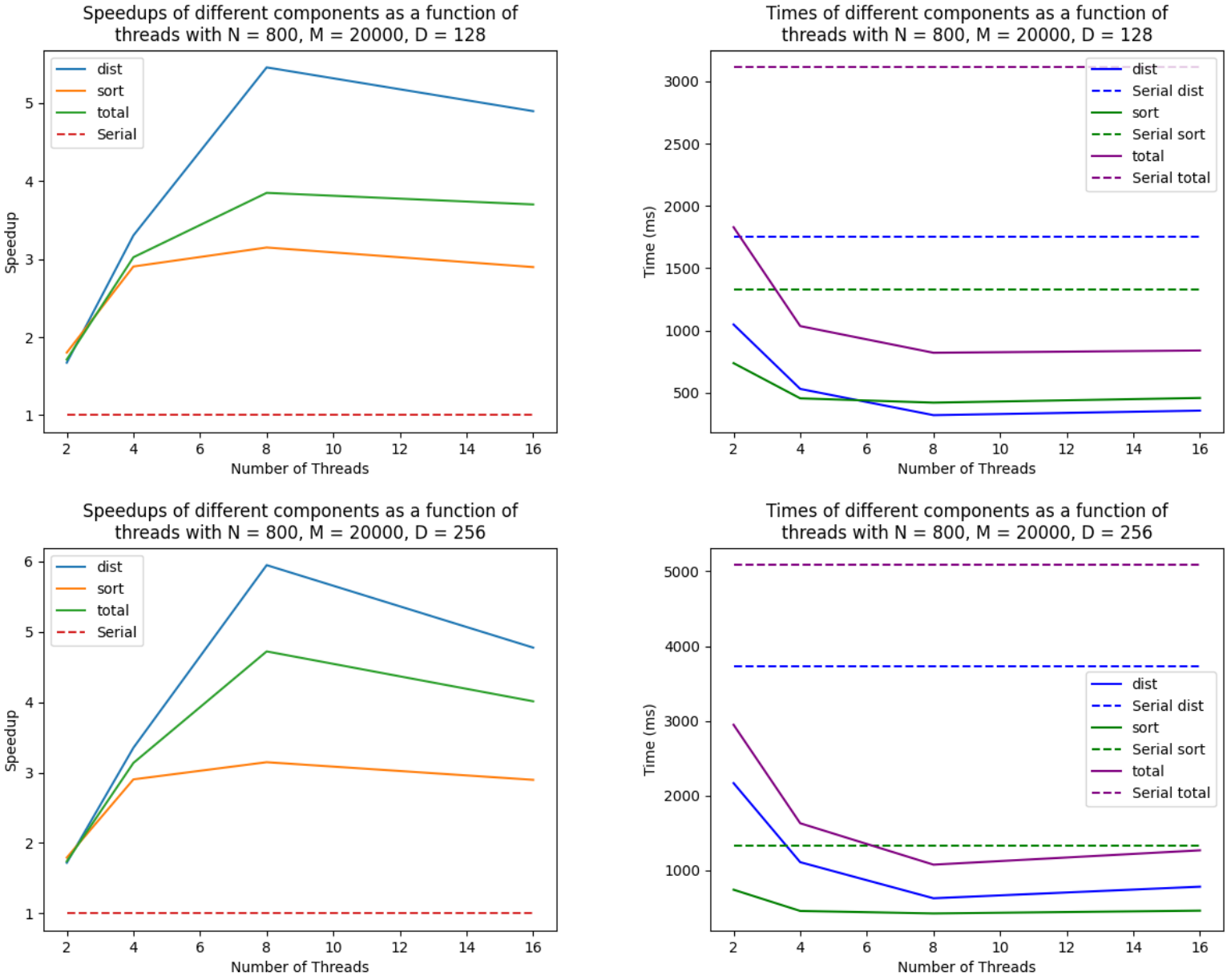


Figure 6: Plotting the performance as a function of the number of threads for 2 values of D

Challenges

We faced a few challenges and problems during this project. The main ones were:

- Finding good values for the *low limit* for sections and tasks. We chose a constant value that performed decently

(especially with tasks), but this can definitely be improved.

- Sorting the combination of a double and an integer, to be able to retrieve the actual top k points was a compromise that impacted our sorting performance, but allowed us to conveniently retrieve the top k points.
- Finding a good range of values for M, N and D to iterate over. We had to find a balance between large enough values to give valid results, and small enough values that they will run in a reasonable time.

Conclusion

In this report, we implemented the KNN algorithm and parallelised the most compute intensive parts. We compared various different methods, sorting algorithms as well as problem sizes. We found that the exact relationship between the parameters M, N and D can affect the distribution of times significantly.

We achieved a good speedup when using multiple cores, but speeding up the sorting component would help the total speedup as well. A way to do this is to potentially parallelise the sorting loop in addition to the parallel sorting already implemented. Preliminary results showed that this method has promise. We found that the exact distance method (euclidean vs manhattan) didn't really impact the performance, because they are in effect performing the same amount of work. The sorting algorithm definitely had a large impact.

Quick sort was by far the best sorting algorithm, even though bitonic sort is explicitly designed to be parallelised. Our lack of great performance from bitonic sort might be caused by the fact that our implementation was recursive and didn't assume that the number of elements to sort is a power of 2. Using a iterative, looping version or assuming the latter condition might improve the parallel performance significantly.

Parallelising using tasks seemed to perform much better than when using sections and for quick sort, sections resulted in bad performance. There are a few (possible) reasons for this:

- Sections make use of nested parallelism, and create a parallel region at every recursive iteration. This adds overhead.
- When experimenting to find the *low limit* for sections and tasks, we found that the best one for sections (10000) was 10 times that of tasks (1000). When using a low limit below that for sections, the results were worse (especially for quicksort). Our datasets barely managed to use a large enough M to make the sections actually take advantage of the parallelism.
- This low limit might just be flawed, and using another one might increase the performance of sections.

Our code is generally scalable, and achieves higher performance when the number of threads is increased, up to the hardware limit. When working on a larger problem size, we still achieve a consistent (or better) speedup.
Even though the brute force algorithm is not very efficient, parallelising it has the potential to increase performance significantly.

Acknowledgements

Computations were performed using High Performance Computing infrastructure provided by the Mathematical Sciences Support unit at the University of the Witwatersrand.

References

[1] Padraig Cunningham and Sarah Jane Delany. k-nearest neighbour classifiers: 2nd edition (with python examples), 2020.

[2] N.S. Altman. An introduction to kernel and nearest neighbor nonparametric regression. 1991.

[3] Joel Lehman and Kenneth Stanley. Exploiting open-endedness to solve problems through the search for novelty. *Artificial Life - ALIFE*, 01 2008.

[4] Jyrki Katajainen and Jesper Larsson Träff. A meticulous analysis of mergesort programs. In Giancarlo Bongiovanni, Daniel Pierre Bovet, and Giuseppe Di Battista, editors, *Algorithms and Complexity*, pages 217–228, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[5] Steven S. Skiena. *The Algorithm Design Manual*, pages 126, 129. Springer, London, 2008.

[6] Vipin Kumar. *Introduction to Parallel Computing*, page 389. Addison-Wesley Longman Publishing Co., Inc., USA, 2nd edition, 2002.