Final Project Report
Parallel Scientific Computing

Douglas Yuan (7508690)
8/21/2022

1. Introduction

The purpose of this final project is to construct a metric tree in parallel, which is used to efficiently perform k-nearest neighbors search. The metric tree is a data structure that efficiently partitions a dataset cast in n-dimensional space by randomly picking a starting point in the space and recursively splitting elements into two disjoint subtree halves: elements that are inside some radius of the start point, and elements outside of the radius. In this way metric trees are different from other trees for spatial search and partitioning that exploit the triangle inequality[1] or use rectangular shaped partitions[2] instead of circles (or spheres in higher dimensions) to prune nodes from the tree.

Within the metric tree construction algorithm, there are two main aspects that will be parallelized in this work:

- Calculating distance between starting point and all data points in parallel
- Creating the inside and outside sets (subtrees) in parallel

Therefore, this work covers the implementation of one serial program for the creation of the tree, as well as two other versions with parallelism for the above points which take advantage of multiple thread programming, shared memory, and synchronization techniques. The parallelized versions are implemented in C using the Pthreads and OpenMP paradigms. Benchmarks were performed for different inputs of dataset size as well as dimension size, to find the average execution times of each of the three versions. The result is a comparison of performance between serial and parallel versions, as well as a comparison of speedup and efficiency between different parallel versions.

Below is a description of code functionality and structure, programming methods for the three versions, and a project development timeline.

2. Project Methods

2.1 The Data Structure

The project's approach drew inspiration from the work in C++ of Steve Hanov[3] in order to create a simple metric tree for spatial partitioning. Our metric tree is defined through the following structure:

```
struct MetricTree {
    double* vp;
    double md;
    int idx;

    MetricTree* inner;
    MetricTree* outer;
};
```

Member variables declared are the starting point (which Hanov calls the vantage point) *vp*, the median distance *md* of this point from all other data points, and the index *idx* of the starting point in the original dataset. Additionally, we have the inner and outer subtrees to be created.

The tree is designed to have one main build function that is analyzed below, alongside five accessor functions for the collection of variables listed.

2.2 The Algorithm

```
MetricTree* build(double *X, int n, int d);
```

We will build the tree from the top down from an input dataset *X*. This set is represented as an *n*-by-*d* array, where *n* is the number of data points and *d* is the number of dimensions of the space. We then choose a starting point in an arbitrary manner; all versions of this algorithm choose the very last point in *X*. Next, we calculate the euclidean distance of the start point from all other points. This part is done serially, or in parallel with work-sharing as discussed in the coming section. Then, pass these distances to a selection function of choice; all versions of this

algorithm use a direct implementation of quickselect[4]. Use the result of calls to quickselect to return the median distance:

```
double quick_median(double arr[], int length){
if (length % 2 == 1)
   return quickselect(arr, length, (length+1)/2);
else
   return 0.5 * (quickselect(arr,length,length/2) +
                 quickselect(arr,length,length/2 + 1));
}
```

With the median found, we can calculate the array sizes for subtrees. For all n distance values less than or equal to the median, increase the length of the inner array; the outer length will then be `n-inner`. Distances up to and equal to the median go on the inner subtree, and those greater than the median go on the outer subtree. Now convened, we sort the inner and outer array elements by distance. Finally, recurse on inner and outer to build for the next level of the tree, until inner and outer are of size 0.

2.3 Creating Parallelism

As mentioned above, there are two main aspects that were parallelized in this work. The first aspect was parallelizing the distance calculations between the chosen starting point and all other points in the set. The differences between the three versions at completing this task are discussed below.

Serial

The serial tree building program performs its calculations using the `euclidean()` function as it stands. This returns a pointer to an array that contains all distance values from the starting point *s*:

```
double * euclidean(double *s, double *points, int n, int d) {
  double *distances = (double*)calloc(n, sizeof(double));
  double accumulator = 0;
```

```
  for (int i = 0; i < n; i++) {
      accumulator = 0;
      for (int j = 0; j < d; j++)
          accumulator += (s[j] - *(points + i*d + j)) *
                         (s[j] - *(points + i*d + j));
      distances[i] = sqrt(accumulator);
  }
  return distances;
}
```

OpenMP

For distance calculation, the OpenMP implementation performs automatic parallel conversion of the computing loop using `omp parallel` and `omp for` directives with an automatic schedule:

```
  #pragma omp parallel private(accumulator){
      #pragma omp for schedule(auto)
      for (int i = 0; i < n; i++) {
         accumulator = 0;
         for (int j = 0; j < d; j++)
            accumulator += (s[j] - *(points + i*d + j)) *
                           (s[j] - *(points + i*d + j));
         distances[i] = sqrt(accumulator);
      }
  return distances;
}
```

Pthreads

The distances array is divided into k parts, where k is the number of threads available at that time. Each thread takes its own share of the for-loop work. Program flow stops until calculations are finished, and threads are joined with `pthread_join()`.

```
disThread = calloc(threads - 1, sizeof(pthread_t));
for (int i = 0; i < threads - 1; i++)
   pthread_create(&disThread[i], NULL, euclidean, (void *)&disArg[i]);
// Work for the main thread ... monitor live thread count ... Join threads
for (int i = 0; i < threads - 1; i++)
   pthread_join(disThread[i], NULL);
```

The second aspect was parallelizing the creation of inside and outside subtrees. The differences between the two parallel versions are discussed below.

OpenMP

The OpenMP version performs automatic conversion of `build()`'s recursive step with `parallel` and `section` directives. The inner and outer subtree creation is covered in two sections, and are to be assigned to two different live threads to achieve work-sharing.

```
#pragma omp parallel shared(node) {
   #pragma omp sections {
     // Create threads
     #pragma omp section
     if(innerLength > 0)
        node->inner = build(innerPoints, innerIDs, innerLength, d);

     #pragma omp section
     if(outerLength > 0)
        node->outer = build(outerPoints, outerIDs, outerLength, d);
  }
}
```

Pthreads

The Pthreads implementation employs two available threads and performs outer subtree creation on the main thread. Then it delegates inner subtree creation to a child thread. Here, we again use `pthread_join()` for thread synchronization.

```
pthread_t subThread;
if((innerLength > 0) && (outerLength > 0)) {
   // Start inner tree creation on a thread
   pthread_create(&subThread, NULL, build_inner, (void *)subArg);
   // Run outer tree creation in the main thread
   node->outer = build_outer(outerPoints, outerIDs, outerLength, d);

   // Join thread
   pthread_join(subThread, NULL);
}
```

2.4 Timeline

This is the basic timeline of project development:

Week 1 (8/10 - 8/13)

- Conduct background research.
- Draft the initial project proposal.
- Start implementation of serial construction algorithm.

Week 2 (8/14 - 8/20)

- Finish implementation of serial program.
- Complete parallel implementations using Pthreads and OpenMP.
- Compare performance results between serial and parallel.
- Analyze speedup and efficiency of parallel versions.
- Begin drafting project report.

Week 3 (8/21 - 8/23)

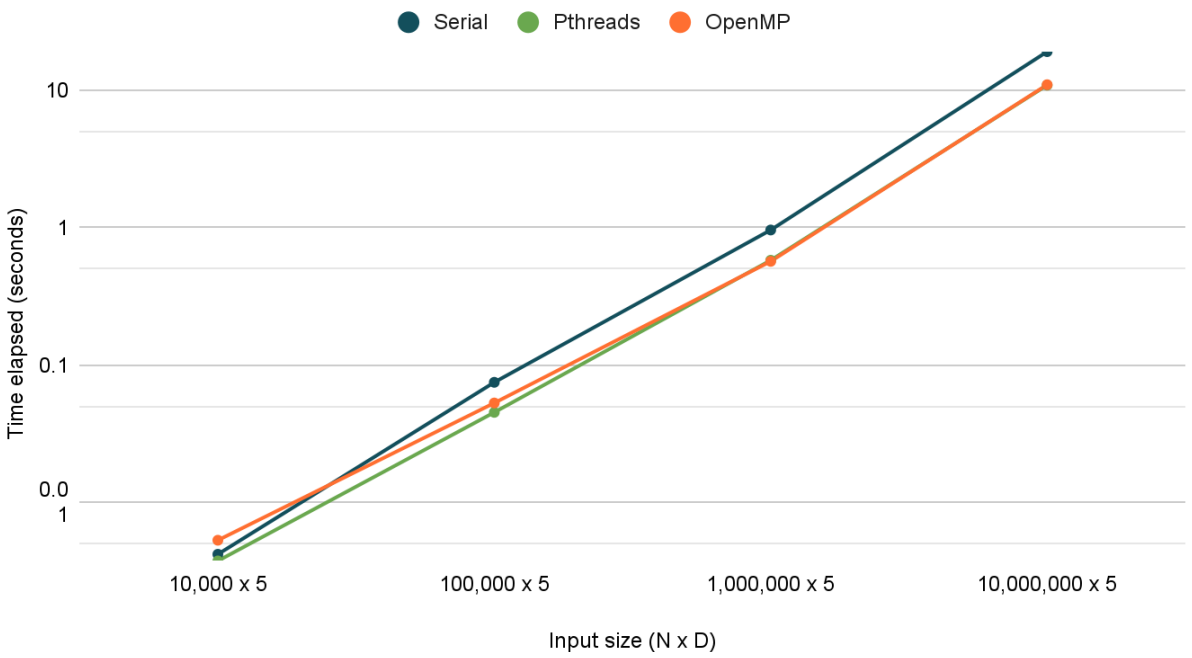
- Finish project report.

---

3. Project Results

To ensure that we solely gather data on the algorithm and not the data creation overhead, we only measure the execution time of tree construction. The benchmark function within `main.c` randomly generates a suitably large dataset prior to timing. Using increasingly complex problem sizes involving both data points and dimensions, we benchmark program performance. Tests were run on a machine with a dual-core Intel Pentium G3258 at a 4.0GHz clock and 2 hardware threads, and 8GB of DDR3 RAM.

Below are timing results in graph and table form. With regards to input dataset size, N is the number of data points, D is the number of dimensions. Times are reported in seconds.



Graph 1: Time elapsed for all construction algorithms at D = 5 problem sizes.

Input size (NxD)	**serial**	**Pthreads**	**OpenMP**
10,000 x 5	0.004176	0.003735	0.005283
10,000 x 50	0.015957	0.014899	0.013308
10,000 x 500	0.104645	0.065298	0.070047
100,000 x 5	0.074820	0.045205	0.052897
100,000 x 50	0.238934	0.139755	0.143364
100,000 x 500	1.639685	0.908958	0.893992
1,000,000 x 5	0.964336	0.580354	0.569859
1,000,000 x 50	4.173672	2.482149	2.497005
1,000,000 x 500	30.741532	16.890103	17.043323
10,000,000 x 5	19.234496	10.956070	11.045268
10,000,000 x 50	80.530315	44.643821	43.767225
10,000,000 x 500	601.968824	325.38278	320.943608

Table 1: Average time elapsed of each algorithm on an NxD input set.

The results demonstrate a significant increase in performance from both parallel implementations compared to the serial version. Using this raw data, speedup *S* and efficiency *E* of each algorithm was calculated and collected in the next table. The relative speedup for each problem size N x D provides a metric to more clearly determine the scalability of each algorithm's performance as the problem size increases.

Input size (NxD)	**serial S & E**		**Pthreads (#proc = 2) S | E**		**OpenMP (#proc = 2) S | E**	
10,000 x 5	**1.0000**		1.118072	0.559036	0.790459	0.395229
10,000 x 50	**1.0000**		1.071011	0.535505	1.199053	0.599526
10,000 x 500	**1.0000**		1.602576	0.801288	1.493926	0.746963
100,000 x 5	**1.0000**		1.644126	0.827563	1.414446	0.707223
100,000 x 50	**1.0000**		1.709664	0.854832	1.666624	0.833312
100,000 x 500	**1.0000**		1.822547	0.909113	1.831460	0.915730
1,000,000 x 5	**1.0000**		1.661634	0.830817	1.692236	0.846118
1,000,000 x 50	**1.0000**		1.681508	0.840754	1.671474	0.835737
1,000,000 x 500	**1.0000**		1.820692	0.910346	1.803728	0.901864
10,000,000 x 5	**1.0000**		1.756574	0.878287	1.742255	0.871128
10,000,000 x 50	**1.0000**		1.803840	0.901920	1.837685	0.918843
10,000,000 x 500	**1.0000**		**1.850292**	**0.925146**	**1.875849**	**0.937925**

Table 2: Speedup & efficiency of each algorithm on an NxD input set. Highs are bolded.

The data shows that with both Pthreads and OpenMP, the speedup consistently reaches close to 2 at the larger tested problem sizes; $S > 1.8$ ($E > 0.9$) values are highlighted green within Table 2. This is a significant result, given a 2x speedup is the maximum theoretical limit for the hardware that these tests were run on, which was indeed limited to two processors. The processors are well-utilized in both parallel programs at $E = 0.925$ and 0.938 while running the largest tested size, N = 10,000,000 and 500 dimensions. Of course, the proportion of effort being wasted on

communication and synchronization between threads is significantly higher at lower sizes, especially those of under 10,000 points. Up to this size, parallelization yields ineffectively small or even negative returns. For this reason the two parallel programs were designed to switch execution modes from serial to parallel for distance calculations and subtree construction only after a specific point threshold.

---

4. Conclusion

4.1 Project Summary

This project aimed to implement a metric tree construction algorithm in C, with both serial and parallel versions. In the end, this work successfully parallelized the two main parts of the construction process, and accelerated the serial version almost to the theoretical maximum ($2x$) by factors of **1.8502** and **1.8758** with Pthreads and OpenMP paradigms, respectively.

There is one particularly interesting discovery regarding the algorithm's performance results. First, the time taken to build the metric tree grows in a linear fashion with the number of data points. The trend is visible in Graph 1, using values N = $\{10^4, 10^7\}$ while fixing D at 5. However, build time grows much more slowly with the increase of dimensions. Through the increase from D = 5 to D = 50, the total build time for all tested problem sizes scaled by no more than 4.33x, with the lowest being 2.71x at N = 100,000 with OpenMP. This is one intrinsic nature of the data structure that this work did not analyze to the fullest extent, given the smaller range of dimensions tested.

4.2 Afterthoughts

A notable restriction of this work is that performance benchmarks were run on a machine with only two cores and two threads. Experimenting with increasing numbers of processors would be unfruitful given the hardware limits, and so experiments quantifying the scalability of parallelism have not been presented with this report. This marks a straightforward next step for this work,

since some techniques used to create parallelism may not maintain performance at scale. One quick example that comes to mind is the use of 'section' directives to parallelize subtree construction in the OpenMP program; this method dictates that the inner and outer arrays are to be handled by different threads. It is worth noting however that, as presented, this code defines two parallel regions and may not extract more parallelism than two threads. With a larger number of processors, there can exist a number of live threads without work to do who will likely sit idle with this implementation.

Code for this project is viewable here.

References

1) "M-Tree." Article on Wikipedia here.
2) "kD-Trees." Written by Carl Kingsford for CMSC 420 at CMU, appearing here.
3) "Data structures for finding stuff fast." Written by Steve Hanov, appearing here.
4) "Quickselect Algorithm." Article on GeeksforGeeks here.