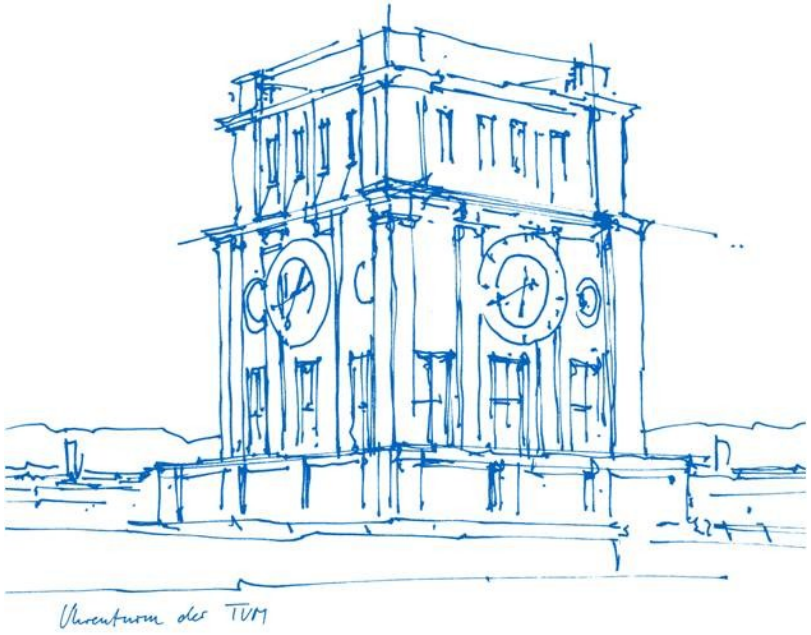
Parallel Programming SS21 Final Project

Project 3: KDTree

Group - 301

9th July 2021

Virág Vörös



Outline

1. Sequential code analysis - Profile
2. Sequential code analysis - Amdahl‘s law
3. OpenMP
   i. Parallelized implementation and approach
   ii. Intermediate Speed-up results and profiling
   iii. Final Implementation improvements and new speed-up (or Theoretical Analysis of Improvement limits with OMP)
4. Message Passing Interface (MPI)
   i. Parallelized implementation and approach
   ii. Intermediate Speed-up results and profiling
   iii. Final Implementation improvements and new speed-up (or Theoretical Analysis of Improvement limits with MPI)
5. Hybrid (OpenMP + MPI)
   i. Parallelized implementation and approach
   ii. Final performance results
6. Conclusion

Sequential code analysis - Profiling

- Profiling is the process of analysing all parts of the code and determine, which parts are consuming the most time and processing powers.
- Profiling can be used to find bottlenecks in our code, which would give a better idea which part of the code would need to be rewritten or parallelized to get better performance.
- There are many profiling tools such as: perf, gprof, likwid, vtune, vi-hps. Each of these tools are customised to work with particular applications or hardware.
- In our analysis we have used gprof and perf.
- From gprof, we have summarized the results for the functions in the kdtree_sequential.cpp and used it for the calculation of Amdahl’s Law.

Sequential code analysis - Profiling with gprof

Function name	Time [s]	Time [%]
main(int argc, char** argv)	5.846	99.1
➢ build_tree(Point**, int)	4.242	71.9
➢ build_tree_rec(Point**, int, int)	4.242	71.9
➢ nearest_neighbour(Node*, Point*)	1.150	19.5
➢ nearest(Node*, Point*, int, Node*, float&)	1.150	19.5
➢ Point::distance_squared(Point &)	1.097	18.6

Table 1: Profiling with Gprof

- As it can be seen in Table 1, the most time is being used to create the kd-tree which is the build_tree() and build_tree_rec() [taking 71.9%].
- Hence, first we will parallelize the build_tree() function to obtain a more significant speed up.
- Besides this, the search of the tree for the query points is taking the second most time [taking 19.5%], therefore our second focus is going to be the optimization of the corresponding for loop in main().

Sequential code analysis - Amdahl‘s law

- We know that when trying to improve the efficiency of the code using parallel implementation, there is always a theoretical upper limit for parallelization.

$$SU(p) = \frac{T}{T(p)} = \frac{T}{(1-f)*T + \frac{f*T}{p}} = \frac{1}{1-f+\frac{f}{p}}$$

- The maximum fraction of parallel execution = 0.719 + 0.195 = 0.914, as build_tree_rec() takes 71.9% of the total runtime and the nearest_neighbour() function takes 19.5% of the total runtime.
- Therefore in theory if both these functions could be completely parallelized, we should get maximum speedup. The maximum speedup in case of infinite processing units would be x11.6.
- The estimation of potential parallel code is very complicated and an error can be expected during the calculations.
- For the different parts of the project, we were using different number of maximum parallel tasks, therefore using Amdahl‘s law we get different theoretical upper limit for speed-up as well.
  p(OMP) = 32, p(MPI) = 16, p(Hybrid) = 4.

OpenMP - Parallelized implementation and approach

- As we saw from the sequential code profiling, the function that occupies the most time is the build_tree_rec() function, which is used to create the KD-tree.
- The implementation can be seen in Figure 1 and 2.
- The criteria of task creation was defined as final(depth > 3) as we have 32 threads available.
- First we did not parallelize the for loop in main() to search for query points, as there were only 10 points.
- This gave a speed up of approximately 1.37

```
Node* build_tree(Point** point_list, int num_nodes){
    Node* tree;
    #pragma omp parallel
    #pragma omp single
    tree = build_tree_rec(point_list, num_nodes, 0);

    return tree;
}
```

Figure 1: One single thread creating OMP tasks in build_tree_rec()

```
// left subtree
#pragma omp task shared(left_node) final(depth > 3)
left_node = build_tree_rec(left_points, num_points_left, depth + 1);

// right subtree
#pragma omp task shared(right_node) final(depth > 3)
right_node = build_tree_rec(right_points, num_points_right, depth + 1);

// return median node
#pragma omp taskwait
return new Node(*median, left_node, right_node);
```

Figure 2: Creating separate tasks for the left and right subtree

OpenMP - Intermediate Speed-up results, profiling

- After implementing our first idea in OpenMP to parallelize the build_tree_rec() function, we see that there is only a speedup of 1.37.
- From the profiling we discovered a significant overhead when it comes search for query points (e.g.: distance_squared function). The results from Perf for the sequential code can be seen on Figure 3.
- This can be solved with giving separate query points to separate threads in the main function.
- We calculated the theoretical maximum speedup based on Table 1 and corresponding to Amdahl‘s law with p(OMP) = 32 and f = 0.914 the theoretical maximum speedup of OMP was SU(p) = 8.72.
- The results of the improved approach can be seen on Figure 4.

Symbol	Time [s]	Time [%]
Point::distance_squared	2.71	45.88
Point::compare	1.94	32.88
Utility::generate_problem	0.42	7.08
Others	0.84	14.16

Figure 3: Analyzing sequential code with Perf

Symbol	Time [s]	Time [%]
Point::distance_squared	0.66	36.74
Point::compare	0.70	38.71
Utility::generate_problem	0.09	4.91
Others	0.35	19.64

Figure 4: Analyzing improved OMP code with Perf

OpenMP - Final Implementation improvements and new speed-up

- For finalizing our OpenMP code, we parallelize the build_tree_rec() function by using tasks.
- We also use *#pragma omp parallel for* in the 2 for-loops in main() to improve the search for query points and definition of tree points.
- To ensure correct order of printing, the results from the different threads were collected into an array with the index of the current loop (or thread).
- We use schedule(dynamic, 1) so that each iteration of the loop is given to a separate thread.
- The implementation can be seen in Figure 5 and 6.
- With this implementation we got a speedup of 3.5
- Our speedup is far from the theoretical x8.72: it can be seen in Figure 6, for the search we are utilizing only 10 threads (for loop num_queries) instead of 32, and due to other overheads (e.g.: task barrier)

```
#pragma omp parallel for
for(int n = 0; n < num_points; ++n){
    points[n] = new Point(dim, n + 1, x + n * dim);
}
```

Figure 5: parallelizing definition of tree points

```
#pragma omp parallel for schedule(dynamic, 1)
for(int q = 0; q < num_queries; ++q){
    float* x_query = x + (num_points + q) * dim;
    Point query(dim, num_points + q, x_query);

    Node* res = nearest_neighbor(tree, &query);

    // array to print data in correct order
    distances[q] = query.distance(*res->point);
```

Figure 6: parallelizing search for query points

MPI - Parallelized implementation and approach

- Our first idea for the MPI implementation was that each process builds up the whole KD-tree from all of the points, but later searches only for one query point, this way the processes can be executed in parallel and then the results can be collected back to the root.
- We used MPI_Bcast() to send the seed, dim and num_points to each of the processes and calculate one mapped query point.
- MPI_Gather() function was used to collect the distances in order, which is then displayed when all the messages are returned to the root.
- The implementation can be seen in Figure 7 and 8.

```
if(rank == 0) {
    Utility::specify_problem(&seed, &dim, &num_points);
}

MPI_Bcast(&seed, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&dim, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&num_points, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

Figure 7: distribute parameters of full tree

```
// create array of distances in order to print data
float distances[size] = {0.0};

// for each query, find nearest neighbor
float* x_query = x + (num_points + rank) * dim;
Point query(dim, num_points + rank, x_query);

Node* res = nearest_neighbor(tree, &query);

// output min-distance (i.e. to query point)
distances[0] = query.distance(*res->point);      You, seconds ago • Uncommitted c

// gather data into process with 0 rank
MPI_Gather(distances, 1, MPI_FLOAT, distances, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);

// print results with rank 0
if(rank == 0) {
    for(int print = 0; print < num_queries; ++print){
        Utility::print_result_line((num_points + print), distances[print]);
    }

    std::cout << "DONE" << std::endl;
}
```

Figure 8: calculating query points and gathering into root

MPI - Intermediate Speed-up results, profiling

- By using this approach we were able to obtain a speedup of 1.5.
- Using Amdahl‘s law with p(MPI) = 16 and f = 0.914, from the formula we get the theoretical maximum speedup of SU(p) = 7.0. The percentage of parallelized code is roughly calculated based on Table 1.
- As it can be seen from Table 1, we were building up the tree many times and therefore not optimizing a very time-consuming function.
- The results of the recursive function cannot be collected by MPI functions, as the return type is not supported. But our first implementation was not giving sufficient speedup, therefore we came up with a different approach: dividing the tree into small trees and searching for all the query points only on the small tree. The results from Perf can be seen on Figure 9 and 10.
- T[illegible]o build_tree() and the search of queries

Symbol	Time [s]	Time [%]
Point::distance_squared	2.71	45.88
Point::compare	1.94	32.88
Utility::generate_problem	0.42	7.08
Others	0.84	14.16

Figure 9: Analyzing sequential code with Perf

Symbol	Time [s]	Time [%]
Point::distance_squared	0.17	18.46
Utility::generate_problem	0.44	49.17
Others	0.29	32.37

Figure 10: Analyzing improved MPI code with Perf

MPI - Final Implementation improvements and new speed-up

Figure 11: distributing tree points among all processes

```
// distribute points among threads
for(int n = 0; n < num_points; ++n){
    int d = floor(n / num_points_loc);
    if(rank == d){
        points[div_counter] = new Point(dim, n + 1, x + n * dim);
        div_counter++;
    }
}
```

Figure 12: calculating and reducing distances

```
// create array of distances in order to print data
float distances[num_queries] = {0.0};
float min_distances[num_queries] = {0.0};

// for each query, find nearest neighbor
for(int q = 0; q < num_queries; ++q){
    float* x_query = x + (num_points + q) * dim;
    Point query(dim, num_points_loc + q, x_query);

    Node* res = nearest_neighbor(tree, &query);

    // output min-distance (i.e. to query point)
    distances[q] = query.distance(*res->point);
}

// reduce minimum distance into process with 0 rank
MPI_Reduce(distances, min_distances, num_queries, MPI_FLOAT, MPI_MIN, 0, MPI_COMM_WORLD);

// print results with rank 0
if(rank == 0) {
    for(int print = 0; print < num_queries; ++print){
        Utility::print_result_line((num_points + print), min_distances[print]);
    }

    std::cout << "DONE" << std::endl;
}
```

- In our final implementation we send individual small parts of the tree to each process to build up, and execute the search for all query points
- Later the smallest distance for the calculated query points is collected back at the root and printed.
- MPI_Reduce() function is used to collect the minimum distance of each of the query points.
- The implementation can be seen on Figure 11 and 12.
- With this approach we got a speedup of 7.0.

Hybrid - Parallelized implementation and approach

- As a Hybrid approach, we combined both the OpenMP and the MPI implementations.
- The MPI approach successfully divided the problem into smaller sub-trees and calculated the minimum distances, however the build_tree() recursive function was not parallelized in each of the processes.
- Therefore each process was divided inside of the build_tree() into OpenMP tasks.
- Based on the instructions, we used 4 threads for the MPI. The criteria of task creation was defined as final(depth > 1).
- The implementation can be seen in Figure 13, 14 and 15.
- Using this approach, a speedup of 3.2 was obtained.

Hybrid - Parallelized implementation and approach

```
MPI_Bcast(&seed, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&dim, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&num_points, 1, MPI_INT, 0, MPI_COMM_WORLD);

// local array size on individual processes
int num_points_loc = num_points/size;

// last points are query
float* x = Utility::generate_problem(seed, dim, num_points + num_queries);
Point** points = (Point**)calloc(num_points_loc, sizeof(Point*));

// counter for local entries of "points" array
int div_counter = 0;

// distribute points among threads
for(int n = 0; n < num_points; ++n){
    int d = floor(n / num_points_loc);
    if(rank == d){
        points[div_counter] = new Point(dim, n + 1, x + n * dim);
        div_counter++;
    }
}
```

Figure 13: distributing tree points among all processes (MPI)

```
// for each query, find nearest neighbor
for(int q = 0; q < num_queries; ++q){
    float* x_query = x + (num_points + q) * dim;
    Point query(dim, num_points_loc + q, x_query);

    Node* res = nearest_neighbor(tree, &query);

    // output min-distance (i.e. to query point)
    distances[q] = query.distance(*res->point);
}       You, 2 weeks ago • Hybrid solution

// reduce minimum distance into process with 0 rank
MPI_Reduce(distances, min_distances, num_queries, MPI_FLOAT, MPI_MIN, 0, MPI_COMM_WORLD);

// print results with rank 0
if(rank == 0) {
    for(int print = 0; print < num_queries; ++print){
        Utility::print_result_line((num_points + print), min_distances[print]);
    }

    std::cout << "DONE" << std::endl;
}
```

Figure 14: calculating and reducing distances (MPI)

Hybrid - Parallelized implementation and approach

```
    // left subtree task
    #pragma omp task shared(left_node) final(depth > 1)
    left_node = build_tree_rec(left_points, num_points_left, depth + 1);

    // right subtree task
    #pragma omp task shared(right_node) final(depth > 1)
    right_node = build_tree_rec(right_points, num_points_right, depth + 1);

    // return median node after all tasks finished
    #pragma omp taskwait
    return new Node(*median, left_node, right_node);
}

Node* build_tree(Point** point_list, int num_nodes){
    Node* tree;

    // starting the parallel code
    #pragma omp parallel
    #pragma omp single
    tree = build_tree_rec(point_list, num_nodes, 0);

    return tree;
}
```

Figure 15: Creating separate tasks for the left and right subtree with criteria of final(depth > 1)

Hybrid - Final Performance Results

- According to Amdahl's law with p(Hybrid) = 4 and f = 0.914, from the formula we get the theoretical maximum speedup of SU(p) = 3.2.
- We obtained a speedup of the theoretical maximum, but the percentage of parallelizable code is roughly calculated based on Table 1.
- The results from Perf can be seen on Figure 16 for the sequential code and on Figure 17 for the hybrid approach.

Symbol	Time [s]	Time [%]
Point::distance_squared	2.71	45.88
Point::compare	1.94	32.88
Utility::generate_problem	0.42	7.08
Others	0.84	14.16

Figure 16: Analyzing sequential code with Perf

Symbol	Time [s]	Time [%]
Point::distance_squared	0.46	30.92
Point::compare	0.33	22.25
Utility::generate_problem	0.31	20.44
Others	0.40	26.39

Figure 17: Analyzing Hybrid code with Perf

Conclusion

- OMP: leveraging on access to shared memory, we used the advantage of fast loop-level parallelization, using *#pragma omp parallel for*, but not maximizing the 32 threads (x3.5 speedup).
- MPI: without the benefit of shared memory access and using only one core per compute node, we gained minor speedup, however with divide-and-conquer strategy, broadcasting, gathering and reducing to get the minimum distance x7.0 speedup was obtained.
- Hybrid: assuming not to have shared memory access, but having multiple compute nodes with several cores per node with local shared memory access, we can combine the beneficial aspects of OMP and MPI implementations. We kept the problem division from MPI, and divided the recursive function into OpenMP tasks (x3.2 speedup).

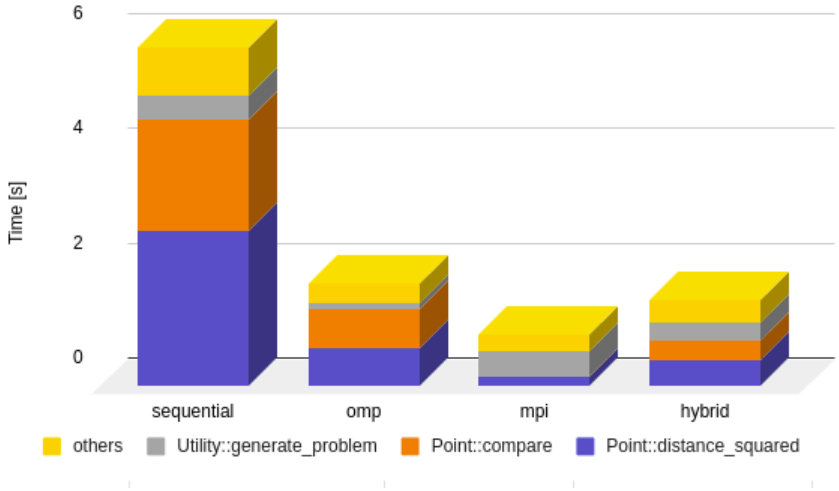


Figure 18: Summary of performance analysis