Enrico Giorio 197969

José Miguel Martinho 98689

Miguel Guerreiro 198818

Group 30

OpenMP Implementation Report

SERIAL IMPLEMENTATION

The serial version of the program consists of a recursive algorithm that takes as input a set of points along with other auxiliary inputs. The base case that stops the recursion is when the set size is 1.
The algorithm, where possible, tries to work with a set of indices (long*) that refer to the original points table (long**). This approach allows reading the coordinates of each point by accessing the points[np][dim] matrix by index and avoids allocating points and computing projections through the matrix itself. Some notable implementation choices are now explained.

After the calculation of the furthest points in the set (namely a and b), we proceed by calculating the orthogonal projection on the $\overline{ab}$ line. The orthogonal projection of every point in the current set consists of an array of structures: struct ProjectedPoint* proj_table;
The following structure:

```
struct ProjectedPoint {
   long idx;
   double *projectedCoords;
};
```

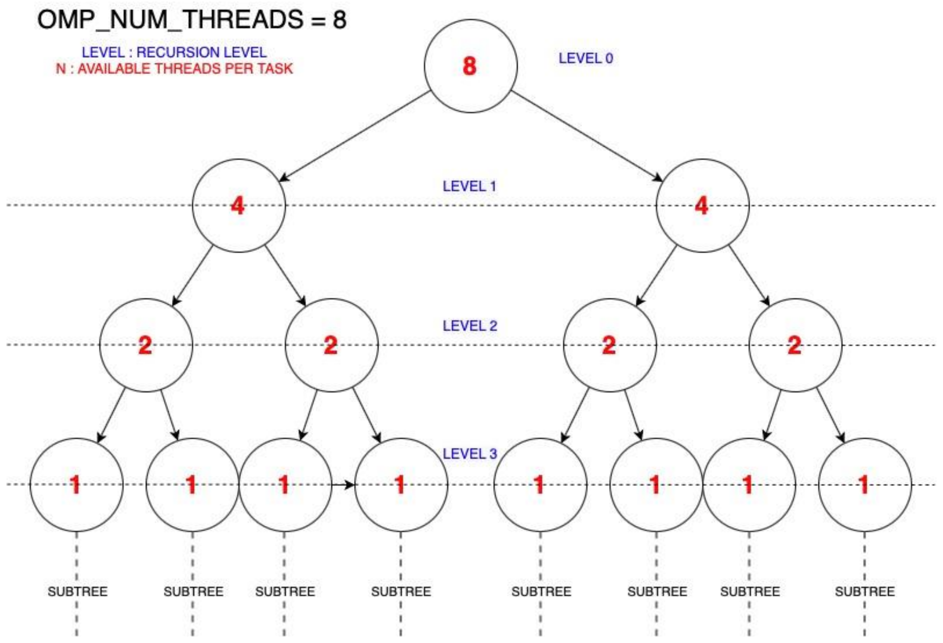
contains the index of the point to which the projection belongs to in the global table along with the coordinates of its projection on the line. Note that points a and b will have their own coordinates as they define the line. Once the structures array is created, a qsort() is performed and it will sort the structures by x coordinate (projectedCoords[0]).

Then, the indices of the structures are copied in the current_set array, preserving the order, and proj_table is freed. This way, current_set will contain the indices of the points, sorted by point position, without actually containing any information about the points' coordinates. It allows us to perform two separate recursion invocations with the two halves of current_set since the first half automatically consists of the left side w.r.t. the center, the same can be said for the right side.

```
res->left = build_tree(node_index + 1, current_set, nextLeftSize);
res->right = build_tree(node_index +nextLeftSize* 2 ,
current_set+current_set_size/2, nextRightSize);
```

PARALLEL IMPLEMENTATION

After several thought out attempts, the best speedups were achieved by using tasks when initiating the recursion. However, it is critical to stop the parallelisation of tasks at some point to avoid any unnecessary spawn of threads that would result in significant overhead and would slow down the execution. The goal is to create parallel tasks until we can effectively assign one task per thread and then stop the task creation so that every thread will execute one major task. The most effective way to follow this approach is to assign the computation of a subtree to every thread, here is a visual example with 8 threads:



First, we observe that levels must be executed in sequence (it is not feasible to start computing a level until the previous level's computation has finished). Once the computation of a level of the tree is completed, all the computing cores are free: new threads will spawn and execute the next level's computation in parallel. The relationship between each recursion level and the number of available threads per task is the following:

$$n = \frac{THREADS}{2^{LEVEL}}$$

Where THREADS is OMP_NUM_THREADS and LEVEL is the recursion level.

We can then:

- Keep track of at which recursion level a thread is computing,
- Calculate *n* at the beginning of the routine

- Constraint every `#pragma omp` directive to be active only when more than 1 thread are available for the level

Most importantly, we can create the tasks only when $n > 1$ to obtain exactly OMP_NUM_THREADS tasks and each of them will be assigned a subtree to compute (provided that the number of threads is a power of 2, otherwise the tasks distribution would be suboptimal).

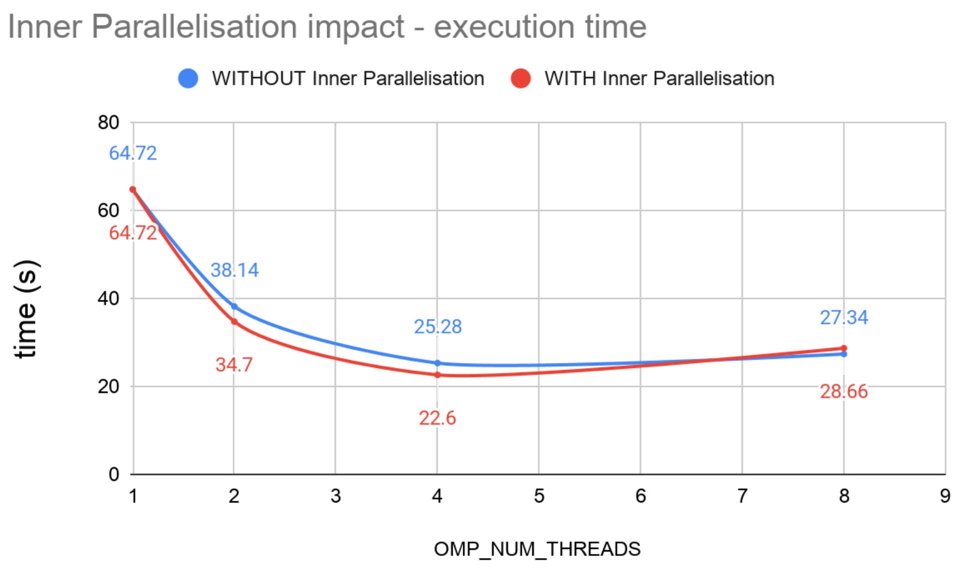
```
#pragma omp task if(n>1)
res->left = build_tree(...);
#pragma omp task if(n>1)
res->right = build_tree(...);
```

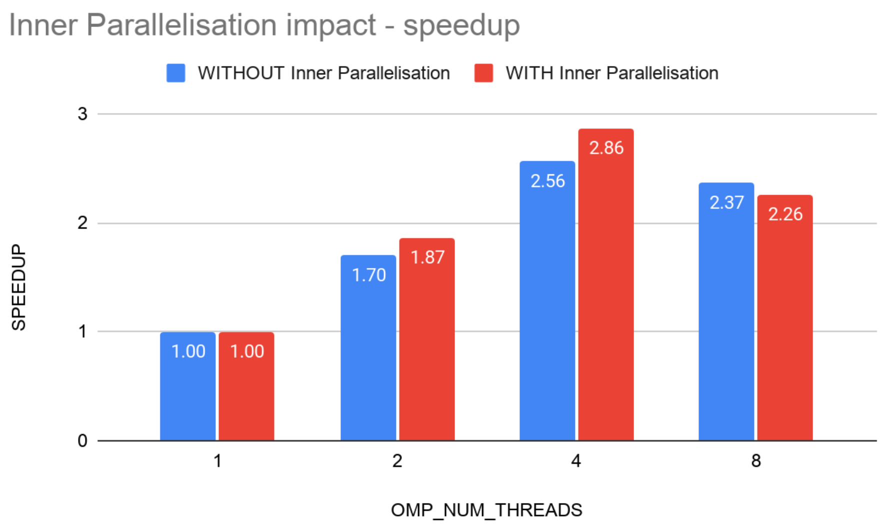
FURTHER IMPROVEMENTS: INNER PARALLELISATION

Tasks that compute at the first levels (where $n > 1$) are executed by more than one thread as we saw earlier, which means that some parallelisation of the various for loops of the algorithm at the upper levels can be done to further improve the speedup, also given the fact that the higher the level, the bigger the set size, which translates into a higher execution time.
An example of inner parallelisation is the routine shown below, used for finding the radius of a hypersphere for a given set. (Constraining the directive via "`#pragma (...) if(cond)`" does not work for some inexplicable reason and the pragma is always activated regardless of the condition, hence why the conditional statement and the code repetition).

```
if(n>1) {
  #pragma omp parallel for reduction(max:highest) num_threads(n)
   for(long i=0; i<current_set_size; i++){
      dist = distance_between_points(center,
points[current_set[i]]);
      if(dist > highest)
         highest = dist;
      }
   }
   else {
     for(long i=0; i<current_set_size; i++){
       dist =
distance_between_points(center,points[current_set[i]]);
          if(dist > highest)
             highest = dist;
     }
   }
```

Other steps of the algorithm have been parallelised in the same way but the obtained results were not as significant as we expected. In the next graphs we are going to compare solutions with and without inner parallelism.
The simulation has been performed with a set of 5 000 000 points in 5 dimensions and the next table shows the measurements.





CONCLUSIONS

The best efficiency has been obtained when using 2 threads whereas, when using more than 4 threads, the overall performance starts to worsen due to overheads in every stage of the execution.

As we can observe, the speedup gain when adding inner parallelism on top of the tasks is quite small. This happens because inner parallelisation is only affecting the first levels' computation. The time that it takes for these nodes to be executed is small compared to the whole tree (formed by millions of nodes) computation time. Taking into account all the overheads of launching/syncing threads before and after each parallel function, the overall improvement that inner parallelisation is able to provide would never be very impactful.

Finally, we observed that the parallel implementation does not scale well due to the recursive nature of the problem.