Report Assignment 2

Romina Doz

February 2022

1 Introduction

The aim of the project is to implement a program that, given a bidimensional dataset that is assumed to be immutable and homogeneous in both the dimensions, can build a kd-tree data structure (with k=2) exploiting parallelism.

To create this structure, starting from the root, at each level a different dimension is considered and the point to be inserted is chosen to be the median of the points with respect to that dimension. In this way, considering that data are homogeneously distributed, the tree will be balanced.

For the parallelization, both the OpenMP and MPI approaches are used.

2 Algorithm

The data structure of the kd-tree is a list of kd-nodes, where each kd-node has the following attributes:

- An array of size k that contains the values of the node (in our case an array of size 2)
- An integer that represents the dimension associated to the node (which dimension was considered when the node was inserted). In this case, it can be 0 or 1.
- Two pointers to the children (left and right kd-nodes)

At the beginning, a list of kd-nodes is created and the values are randomly generated. Then the kd-tree is built by recursive steps:

1. At each recursion, choose the dimension, that must be different from the previous level (Round-Robin scheduling)
2. Sort the list of kd-nodes with respect to the dimension chosen in step 1. For the sake of simplicity, bubble sort algorithm is used, but performance can increase replacing this algorithm with a more efficient one.

3. Take the median as the new kd-node of the tree. The first half of the list will be the left subtree of the node; the second half will correspond to the right subtree.

3 Implementation

The data structure of the kd-tree is implemented by the following struct:

```
struct kdnode{
    double x[MAX_DIM];
    int d;
    struct kdnode *left, *right;
};
```

Some methods have been developed to print the kd-nodes and the kd-tree:

- **void print_kdnode(struct kdnode *n, int size, int dim)**:
  - **Input:** a list of kd-nodes, the size of the list, the dimensions
  - **Output:** prints a list of the values of the kd-nodes, one under the other.
- **void print2D(struct kdnode *root)**:
  - **Input:** the kd-node representing the root of the kd-tree
  - **Output:** prints the kd-tree from the root to the leaves, level by level. The output can be viewed in horizontal from left to right.

In the main, an empty list of kd-nodes is allocated and populated by the following function that generates random numbers between 0 and 1000:

- **void getInput(struct kdnode wp[], int size, int dim)**:
  - **Input:** an empty list of kd-nodes, the size of the list, the dimensions
  - **Output:** populate the values of the empty list with random double precision integers between 0 and 1000

For sorting a simple and naïve solution has been implemented with the knowledge that it can be improved at a later time. The algorithm is the following:

- **void bubbleSort(struct kdnode *t, int n, int dim)**:
  - **Input:** a list of kd-nodes, the number of kd-nodes to be sorted, the dimension to be considered for sorting
  - **Output:** the input array is sorted accordingly to the specified dimension

The recursive function that creates the tree, can be summarized as:

- **struct kdnode * make_tree(struct kdnode *t, int len, int i, int dim, int p, int id)**:
  - **Input:** a list of kd-nodes, the number of kd-nodes to be considered, the dimension that was used for sorting in the previous iteration (for the first call is 1), the total number of dimensions. For the MPI version, also the total number of processors and the rank of the current processor are passed in input.
  - **Output:** the kd-node that has to be added to the kd-tree. For the first call, it matches the root.

3.1 OpenMP

The OpenMP version is characterized by the presence of a parallel region for managing the recursion with multiple tasks inside the **make_tree** method:

```
/* create the kd-tree structure by recursion */
struct kdnode* make_tree(struct kdnode *t, int len, int i, int dim)
{
    struct kdnode *n;
    //choose the dimension using round-robin
    i = (i + 1) % dim;
    //sort the list of kd-nodes according to the dimension i
    bubbleSort(t, len, i);
    if (!len) return 0;

    //find the node to be added to the kd-tree (the median of the sorted list)
    //and start the recursion on the left and right sub-arrays
    //use the parallel tasks
    if ((n = find_median2(t, t + len))) {
        #pragma omp task shared(n, t) firstprivate(i) if(len > 10)
        {
        n->left  = make_tree(t, n - t, i, dim);
        }
        #pragma omp task shared(n, t) firstprivate(i) if(len > 10)
        {
        n->right = make_tree(n + 1, t + len - (n + 1), i, dim);
        }
        #pragma omp taskwait
    }
    n->d = i;
    return n;
}
```

There is a conditional creation of the parallel region depending on the size of the list of kd-nodes. The aim is to avoid the overhead of parallelization when dealing with a small amount of work.

3.2 MPI

For the MPI parallelization, the input dataset is declared in all the processors, but it's populated only on processor 0. In this way, when the function **make_tree** is called, all the processors access it but only one of them owns the input and can generate the tree, with the help of the others.

In fact, at each recursion a domain decomposition is performed in order to split the work between the processors. All processors receive the same number of kd-nodes (load balancing), by the **MPI_Scatter** function. If the number of kd-nodes is not divisible by the number of processors, some processors will have one more kd-node. All the chunks are then sorted by the bubble sort function.

```
// compute the chunk size for all the processors (approximated upwards)
c = (len %p == 0) ?(len/p) :(len /(p - 1));

// compute the chunk size specific for this processor
s = (len >= c * (id+1)) ? c : len - c * id;

//scatter the chunks to the processors
struct kdnode *chunk = (struct kdnode*)malloc(c*sizeof(struct kdnode));
MPI_Scatter(t, c*sizeof(struct kdnode), MPI_BYTE, chunk, c*sizeof(struct kdnode), MPI_BYTE, 0, MPI_COMM_WORLD);

// sort the chunks
bubbleSort(chunk, s, i);
MPI_Barrier(MPI_COMM_WORLD);
```

Once all the chunks are sorted, they are merged by a logarithmic merge, following these steps:

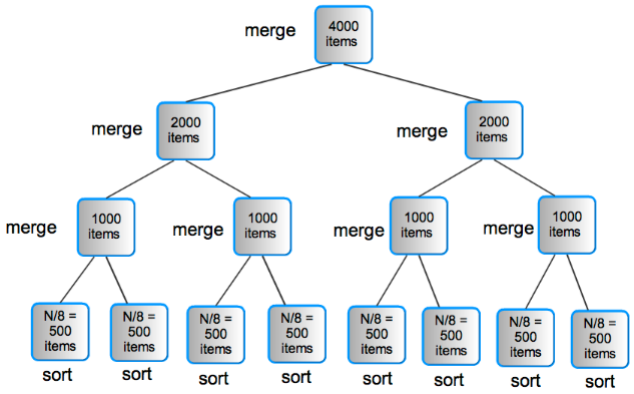
1. Half of the processors send their sorted chunk to the processor with the previous rank and go out from the merge phase

```
// up to log_2 p merge steps
for (step = 1; step < p; step = 2*step) {
    if (id % (2*step)!=0) {
       // id is no multiple of 2*step: send chunk to id-step and exit loop
       if(s>0){
            MPI_Send(chunk, s*sizeof(struct kdnode), MPI_BYTE, (id-step), 0, MPI_COMM_WORLD);
       }
       break;
     }
     // id is multiple of 2*step: merge in chunk from id+step (if it exists)
     if (id+step < p) {
        // compute size of chunk to be received
        o = (len >= c * (id+2*step)) ? c * step : len - c * (id+step);
        // receive other chunk
        if(o>0){
            struct kdnode* other = (struct kdnode*)malloc(o*sizeof(struct kdnode));
            MPI_Recv(other, o*sizeof(struct kdnode), MPI_BYTE, (id+step), 0, MPI_COMM_WORLD, &status);

            // merge the chunk of the processor with the chunk received
            struct kdnode *t = (struct kdnode*)malloc((s+o)*sizeof(struct kdnode));
            merge(chunk, s, other, o, i, t, id);

            free(other);
            chunk = t;
            s = s + o;
            printf("I am processor %d and I MERGED. Now my array has size %d \n", id, s);
         }
      }
   }
```

2. The remaining processors merge their own chunk with the chunk that they received. The merge computation is the following:

```c
/* merge two sorted list of kd-nodes v1, v2 of lengths n1, n2, respectively */
void  merge(struct kdnode * v1, int n1, struct kdnode * v2, int n2, int dim, struct kdnode * t, int id)
{
  int i = 0;
  int j = 0;
  int k, h;

  for (k = 0; k < n1 + n2; k++) {
    struct kdnode *temp;
    if (i >= n1) {
      temp = v2+j;
      j++;
    }
    else if (j >= n2) {
      temp = v1+i;
      i++;
    }
    else if ((v1+i)->x[dim] < (v2+j)->x[dim]) {
      temp = v1+i;
      i++;
    }
    else { // v2[j] <= v1[i]
      temp = v2+j;
      j++;
    }
    for(h=0;h<2;h++) {
         t->x[h] = temp->x[h];
      }
   t++;
   }
}
```

3. The previous two steps are repeated until the whole list is merged on processor 0. The following graph is an example of what happens with a list of 4000 elements and 8 processors.



At this point, processor 0 has the whole array of kd-nodes sorted depending on the chosen dimension. So it can return the median kd-node as the next element of the kd-tree and use recursion on the left and on the right sub-tree.

Before computing the left recursion, a copy (by value) of the second part of the array of kd-nodes is performed by **getRightTree** function, in order to keep an "untouched" version of the array after the left recursions.

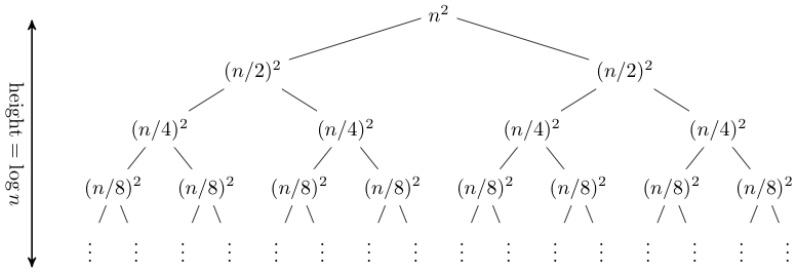
Unfortunately, also with this workaround, the right recursion doesn't start for a segmentation fault error. Because of this bug, the program at the moment is able only to build the left-most branch of the tree.

4 Performance model and scaling

To calculate the complexity of the algorithm, it is convenient to build the recursion tree, considering that at each recursion the input list (of size n) is sorted with an algorithm of complexity $\Theta(n^2)$. Moreover, at each recursion the input list is split in two balanced parts and each part is the input of two recursive calls. So the cost function is:

$$T(n) = 2 * T(\frac{n}{2}) + \Theta(n^2)$$

and can be represented by the following recursive tree:



To solve the cost function, the Master Theorem for Divide and Conquer can be used:

$$T(n) = a * T(\frac{n}{b}) + \Theta(n^k log^p n)$$

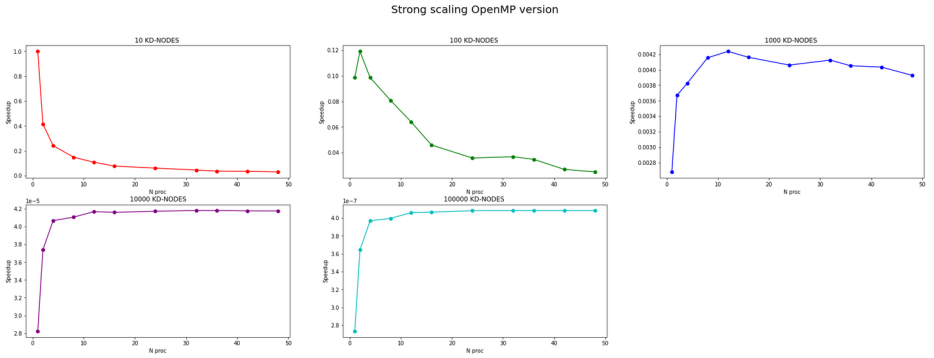
with, in this case, $a = 2, b = 2, k = 2, p = 0$. So, the case $a < b^k$ holds and the result is:

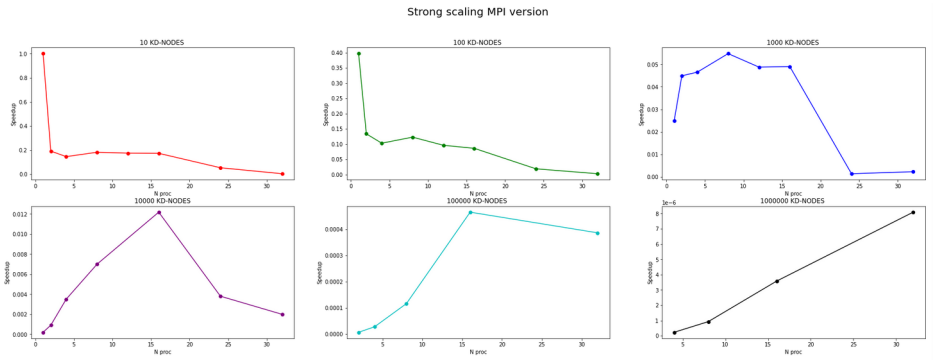
$$T(n) = \Theta(n^2)$$

When the code is parallelized, it is possible to measure the scalability with respect to the number of processes and the size of the problem.
For the OpenMP part, the number of threads was set controlling the environmental variable OMP_NUM_THREADS, while for the MPI program the number of processors is specified by the parameter -np of the mpirun command. For both the versions the input size can be passed at runtime when requested by

the program.

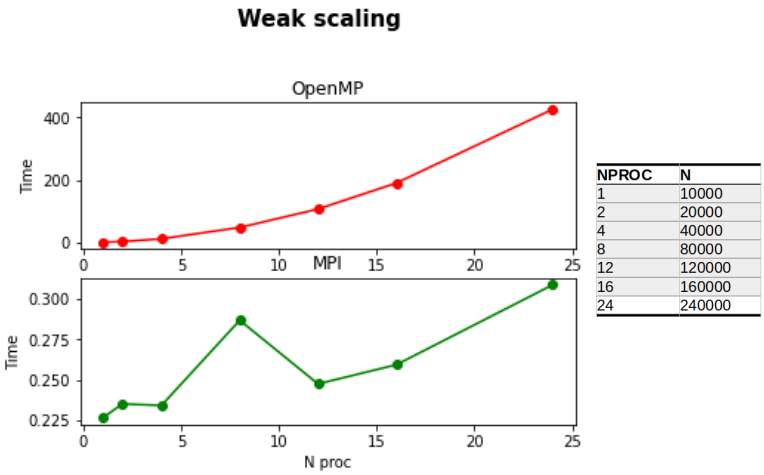
The measurement of strong scaling is done by testing how the overall computational time of the job scales with the number of processing elements. This is done for different sizes, both with OpenMP and MPI. Of course, due to the bug mentioned above, the MPI version is tested only on the computation of the left-most branch of the tree. The results are shown in the following plots:





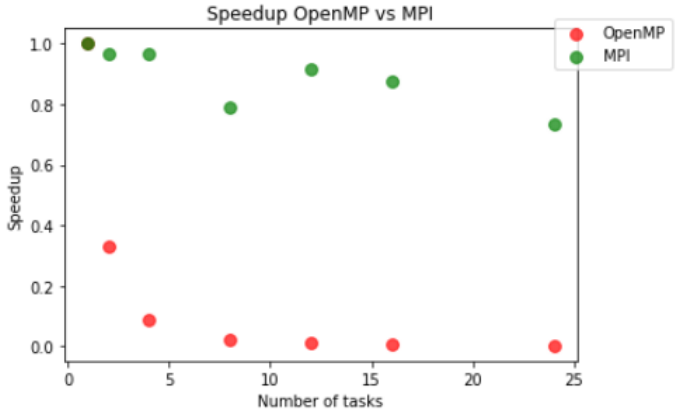
It is possible to notice that in both the OpenMP and MPI implementations, for a small input size (10 or 100 kd-nodes) the time increases when more processing elements are added, because the overhead of parallelization is dominant. When the size is higher, the benefit of parallelization improves the performances following the Amdahl's law trend.

Instead, the test for weak scaling is done by increasing both the job size and the number of processing elements:

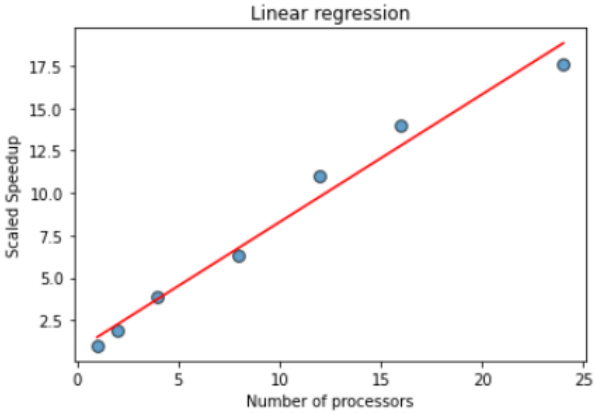


NPROC	N
1	10000
2	20000
4	40000
8	80000
12	120000
16	160000
24	240000

The comparison between OpenMP and MPI speedup is the following:



And it shows that the MPI version has an higher weak scalability. It is possible to fit the scaled speedup of the MPI version in order to retrieve the parallel % of the code, using the Gustafson's law $scaledspeedup = s + p * n$ where s is the proportion of execution time spent on the serial part, p is the proportion of execution time spent on the part that can be parallelized, and n is the number of processors. Fitting the data, the result is a 75% of parallelizable code (the slope of the red line):



5 Discussion

Looking at the plots of the strong scaling, it is evident that parallelization does its job, decreasing the code execution time as the number of processors increases, when the size of the problem is large. But the room for improvement is wide, starting with the sorting algorithm that could be replaced by an algorithm of complexity $O(nlog(n))$, as quicksort.

The idea of sorting at each recursion could be modified, starting from two separate ordered lists, each representing a dimension. In this case, a more complex data structure should be handled, that contains a link between the two arrays. So maybe this solution could be more performant for time complexity, but less for space complexity. Another possibility could be finding at each recursion an "almost median" element as $\frac{min-max}{2}$ and use it to partition and to build an "almost balanced" tree, giving up a perfectly balanced tree but increasing performance.

The most tricky part of the project was the recursion in the MPI version, because all the processors should access the recursive function during all the recursions, but only one of them (0 for simplicity) has to collect the data. This is probably the reason why it was not possible to fix the segmentation fault bug that is thrown as soon as the right recursion is called.

The solution should be analyzed more carefully to properly manage the recursion with MPI.

Finally, memory handling could be improved, allocating less structures and making better use of pointers. To achieve this goal, probably it is required a greater knowledge and experience of the C language.