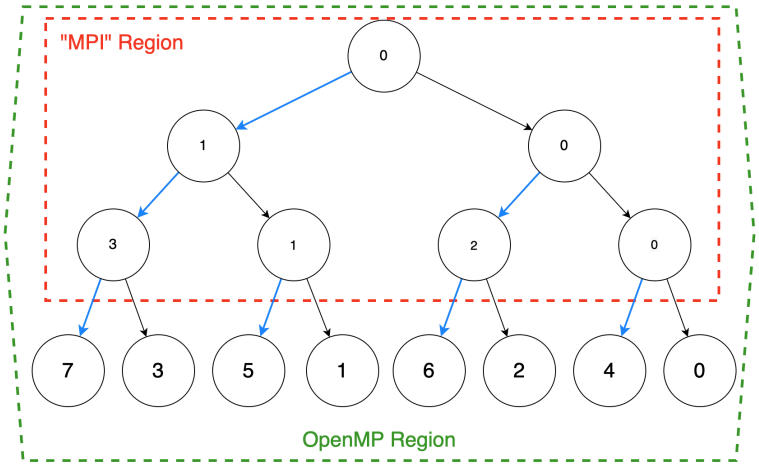
José Martinho
Enrico Giorio 197969

# Group 31 - BallAlg MPI implementation

Regarding the MPI distributed implementation, we decided to keep the OMP implementation and build an improved version starting from there.

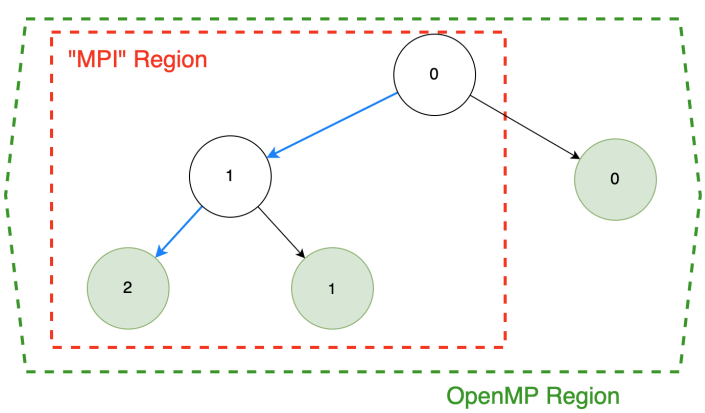
## PARALLELISATION APPROACH

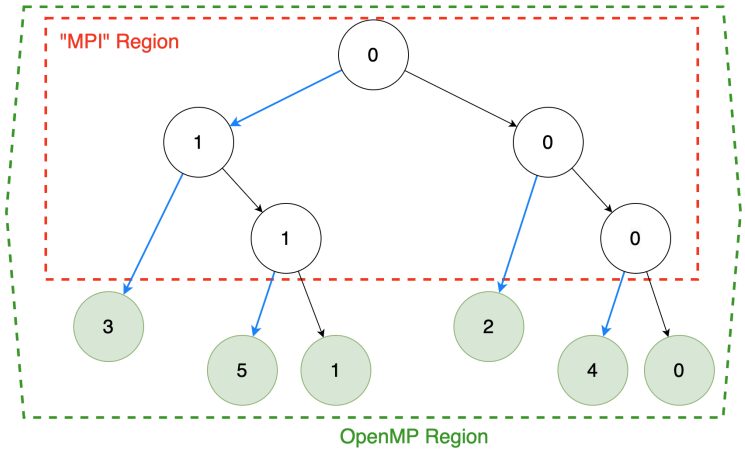
Our first attempt is a safe scaling that relies on the fact that we can implement a similar approach to the one used in the OMP implementation.




An execution model with 8 processing units [0...7] is presented in the figure above.
Every process will generate and store all the points. The root node is computed by only one process [0], the amount of processes involved in the computation then doubles every time a new level of recursion is performed, until all the processes are involved. At this point, every process is assigned a subtree to compute.
The computation of a node of the tree is performed with threads whenever possible (only with the available number of threads for any given processor).
The communication phases are shown by the blue arrows and the processes on the receiving end of these arrows, after the program starts, remain idle, waiting in the MPI_Recv() routine, until the process above finishes computation and is able to send the set of indices of the assigned subtree.
We are aware of most of the limitations of this version:

1) There is a non optimal usage of processes in the red box: many processes are idle and waiting when they could actually contribute to the computation of the higher nodes of the tree.
2) The algorithm is only working with powers-of-two amounts of processors. A solution that allows using an arbitrary number of processors would be in-between the basic solution we are delivering and the advanced solution we are proposing later. The approach would be to assign the computation of subtrees unevenly among processes and would eventually incur in load balancing issues. Following, a couple of illustrations of our hypothetical solution for this issue:

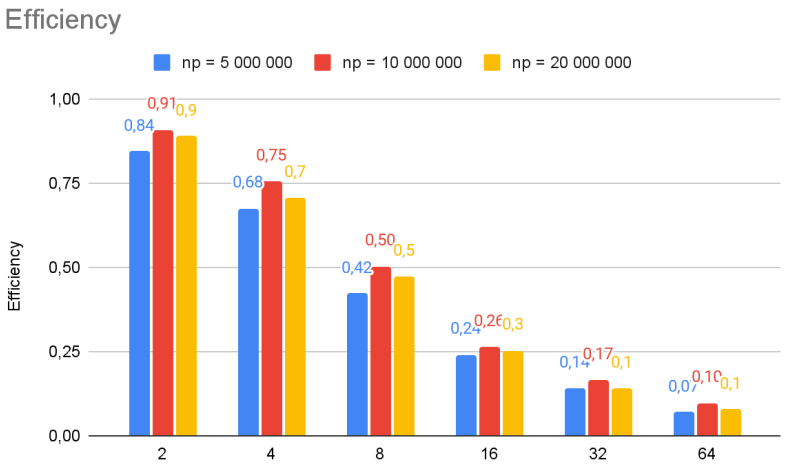


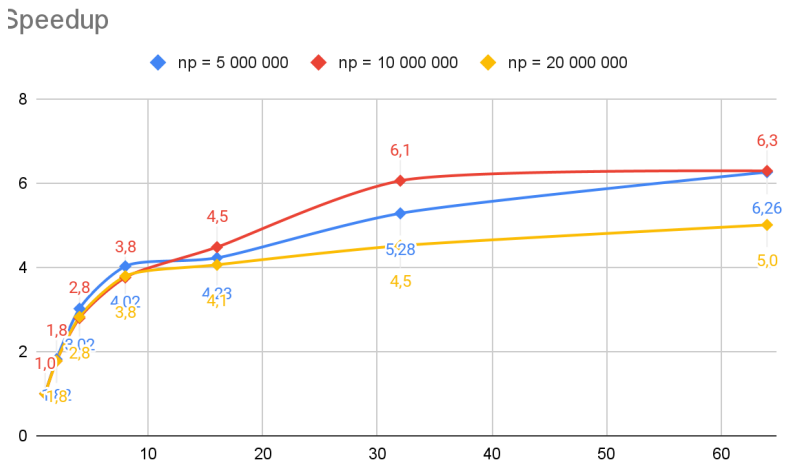




3) The maximum number of points that the algorithm can process is constrained by the maximum amount of doubles that can be stored in p0. Overcoming this issue is particularly complex and requires implementing the algorithm in a distributed recursive way. More on this in the dedicated chapter.

*Performance Analysis.*

We are going to show the performance measurements of the implemented version, in terms of speedup and efficiency:




Once again, without considering the recursive nature of the problem, the results might seem poor. However, we believe that, considering how sequentially-oriented the algorithm is, our implementation significantly improves the performance of the serial program.
Lastly, the speedup decreases as the number of points increases since the lower part of the tree increases exponentially and cannot be parallelised.

*Implementation Note:*

During this last phase of the project, we realised that we could make an improvement on the algorithm itself that could apply to both the sequential version and the OMP version. In particular, we realised that the Projection Table on the AB line was not necessary for every point. Instead, we could calculate only the projection of the first dimension and sort based on that value. We only need to calculate the full projection for the central point.

The program benefits a lot from this improvement and we wish we could apply this “trick” earlier in the project. The sequential version delivered is also improved.

*Advanced MPI implementation*

As mentioned earlier, we tried to improve the maximum amount of points that can be stored. The solution we are going to present is particularly complex and due to time constraint and group situations we were not able to fully test it nor debug it and, as a consequence, extract performance analysis from it. We would like to show our attempt anyway because we think it is a smart and valid solution, at least in principle.

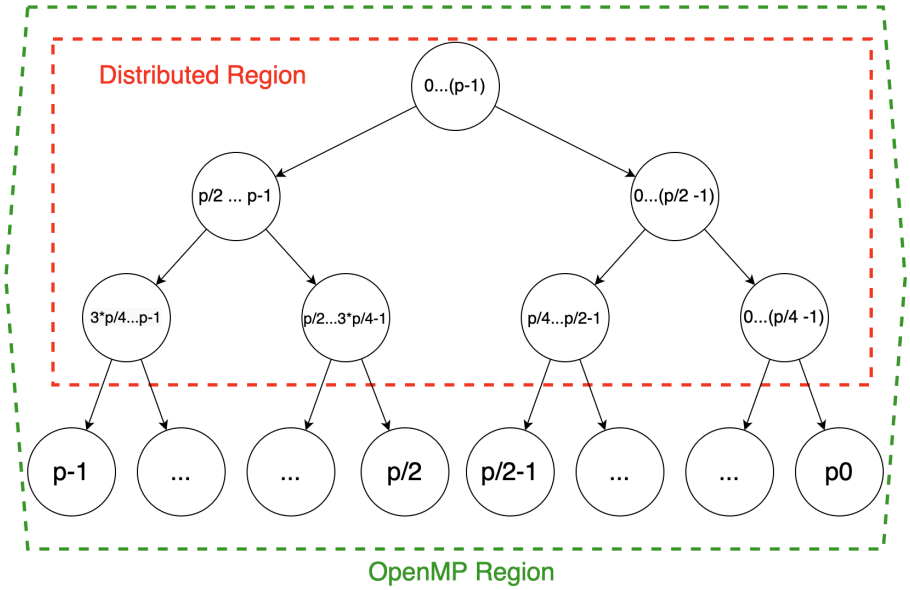
The first step of the algorithm is the generation of points: every process will generate the same amount of points in the same way but it will store only a subset of them.

Then, the computation of the root node starts and is performed distributedly among all the processes: here is an example of one step of the routine (calculating point ‘a’), executed in a distributed way:

```
double * calc_A_dist(){
   double * first = (double *) malloc(dim*sizeof(double));
   if(!me){
       first = points[0];
   }
   //broadcast point 0
   MPI_Bcast(first,dim,MPI_DOUBLE,0,MPI_COMM_WORLD);
   //now everyone has 'first' point
   //calculate local a, furthest from 'first'
   double * a = local_furthest_point_from_point(points, np, first);
   double ** as;
   if(!me){
   //allocate gathering vector of points (to store ‘a’s)
       as = (double **) malloc(nprocs * sizeof(double *));
       for(int i=0;i<nprocs;i++){
           *(as+i) = (double *) malloc(dim * sizeof(double));
       }
   }
   //gather everyone's 'a' at p0
 MPI_Gather(a,dim,MPI_DOUBLE,as,dim,MPI_DOUBLE,0,MPI_COMM_WORLD);
   //calculate real A = furthest among 'a's
   double * A;
   if(!me)
       A = local_furthest_point_from_point(as, nprocs, first);
   //Bcast A
   MPI_Bcast(A, dim, MPI_DOUBLE, 0, MPI_COMM_WORLD);
   //now everyone has the real A of the set
   return A;
}
```

As we can see, every step of the algorithm is much more complex than before and even without accurate performance analysis, we can conclude that there is a lot more communication overhead. The sort is also performed in a distributed way using odd-even transposition sort since it’s important for us to maintain the same size of each process’ set of points.

Locally, we are using our own implementation of Quicksort. The computation of nodes then is divided between additional subgroups of processes that change at every recursion level, as illustrated in the next figure. After the illustration, we are showing the code for the formation of the subgroups.




```
MPI_Group group_world;
MPI_Group subgroup;
MPI_Comm subgroupCommunicator;
int* process_ranks;
int n_comms = nprocs-1; //number of communicators needed
MPI_Comm comm[n_comms]; // array of communicators
comm[0] = MPI_COMM_WORLD;
int* group;
int w;
for(int i = 1 ; i < n_comms ; i++){ //0 is always MPI_COMM_WORLD
  int commsThisLevel = pow(2,i);//there are 2^i communicators (groups) in level i
  //for every communicator there are nprocs/2^i processes
  process_ranks = (int*) malloc(nprocs/commsThisLevel*sizeof(int));
  for(int j = 0 ; j<commsThisLevel; j++){
    //if I belong to group j
    if(me>=j*nprocs/commsThisLevel &&
me <( j+1)*nprocs/commsThisLevel){
        w=0;
        for(int k=j*nprocs/commsThisLevel; k<(j+1)*nprocs/commsThisLevel; k++) {
//loop through all the processes in this group
           process_ranks[w++] = k;
        }
        //get the group under MPI_COMM_WORLD
        MPI_Comm_group(MPI_COMM_WORLD, &group_world);
        // create the new group
        MPI_Group_incl(group_world, nprocs/commsThisLevel, process_ranks, &subgroup);
        // create the new communicator
        MPI_Comm_create(MPI_COMM_WORLD, subgroup, &subgroupCommunicator);
        comm[i] = subgroupCommunicator;
      }
   }
}
```

When we reach a level with the same number of parallel subtrees as available processes, we stop the distributed phase and we proceed with the standard OpenMP implementation we used in all of the previous examples.

To conclude, we would like to highlight the fact that the report is signed by only two members since the third one never showed up at any labs or meetings and has not participated neither in the development phase of the project nor in the testing/writing of the report.