Assignment 2 on MPI and openmp

Foundation of High Performance Computing

Baurice Nafack

Lecturers: **Prof. Stefano Cozzini and Prof. Luca Tornatore**

Date Last Edited: March 20, 2022

1 Introduction : KD-tree Data Structure

Jon Louis Bentley introduced the Kd-tree data structure in 1975 to represent a set of k-dimensional data efficiently for efficient nearest neighbor searches. A kd-tree is a binary tree whose nodes represent points $\{x_i\}_{i=1}^n \in \mathbb{R}^d$. The cutting dimension *cutdim* is assigned to each node, so that all points in the left subtree have a smaller coordinate value than the parent point, and vice versa for the right subtree, see figrure 1. This causes all nodes in a subtree to be bound to a specific region in $\mathbb{R}^d$ within the tree. This can speed up searches for k nearest neighbors because regions further away from the k-th nearest neighbor need not be searched.

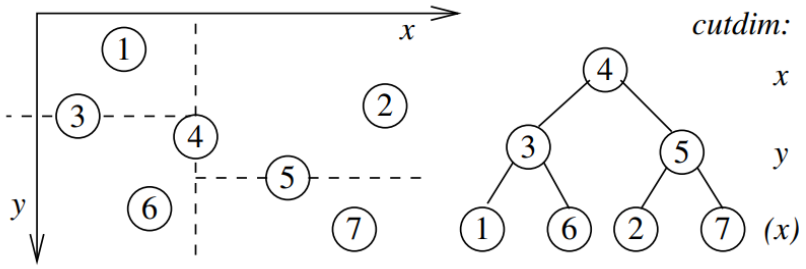

Figure 1: Kdtree formed from seven points in $\mathbb{R}^2$. *Cutdim* is marked by dashed lines, which indicate where the plane is cut by the node.

An k-d tree constructed from n points has the following worst-case complexity:

- When using a heapsort or mergesort for finding the median(spliting point) at each level of the nascent tree, $O(n \log^2 n)$ is needed;
- A median of medians algorithm used to select the median at each level of the nascent tree would take $O(n \log n)$;
- The time required would be $O(kn \log n)$ if n points were presorted according to k dimensions using a sort such as heapsort or mergesort before building the tree.

The assigned task is to create a KD-tree for k=2 by writing a parallel program using points randomly generated. There are two versions to consider:Hybrid MPI+OpenMP and OpenMP.
Notation: n value in the following part refers to the size of the problem in this work, which is 10^n.

2 Kd-Tree Algorithm Description

This section presents and describes the serial algorithm that we will use.

Algorithm 1: An algorithm with non presorting data

```
Data: Choose n ≥ 8
 myaxis = False ;                                  /* False is for x axis and true for y axis */
Data: Generate an array X of 2D data points with length 10^n.
struct {
    ;                                          /* Simple representation of a node inside the tree */
    x,y;
    Node* : Left=NULL, Right=NULL ;            /* Left and Right for left and right subtree
    respectively */
    ,
} Node;
procedure BuildkdTreeBuildkdTree(X, myaxis)
     Initiate X_left and X_right empty array
if  X.length== 1 then
    return  new leafNode (X[1].x, X[1].y);          /* This check if we are in the last node
    return the last leaf if true.  */
else
    if myaxis==0 then
        X ← Sort(X, axis = x) ;                     /* Sorting the data according to x axis */
        l ← X.lenght/2 ;                                    /* Get the index of the median */
        for i ← 1 l − 1 do
            append X[i] to X_left;
        end

        for i ← l + 1 to X.length do
            append X[i] to X_right; ;           /* This data will be use to construct the right
            subtree */
        end

    end
    if myaxis==1 then
        X ← Sort(X, axis = y) ;                     /* Sorting the data according to y axis */
        l ← X.lenght/2 ;                                    /* Get the index of the median */
        for i ← 1 to l − 1 do
            append X[i] to X_left;
        end

        for i ← l + 1 to X.length do
            append X[i] to X_right; ;           /* This data will be use to construct the right
            subtree */
        end

    end
    root = newNode(X[l].x, X[l].y);
    root.left = BuildKdtree(X_L,not myaxis),;           /* The recursive algorithm is applied */
    root.right = BuildKdtree(X_R,not myaxis);                  /* to construct the subtree */
        return return node
end
```

Algorithm 2: An algorithm with presorting data

Data: Choose $n \geq 8$
myaxis = False ; /* False for x axis and true for y axis */
Data: Generate X $2D$ data points with length 10^n.
$X \leftarrow Sort(X, axis = x)$; /* X will be presorted in x axis */
$Y \leftarrow Sort(X, axis = y)$; /* Y will be presorted in y axis */
struct {
  x,y;
  Node* : Left, Right; ,
} *Node*;
procedure BUILDKDTREEBUILDKDTREE($X, Y, myaxis$)
if $X.length == 1$ **then**
  ; /* This check if we are in the leaf node or not */
    return *newleafNode* $(X[1].x, X[1].y)$
end
if *myaxis==False* **then**
  $l \leftarrow \frac{X.lenght}{2}$; /* Get the index of the median */
    allocate X_L, X_R, Y_L, Y_R : point arrays of size l
    Copy $X[0 : L-1]$ to X_L
    Copy $X[L+1 : X.length]$ to X_R
  for $i \leftarrow 1$ $X.length$ **do**
    if $Y[i].x \leq X[l].x$ **then**
      Append $Y[i]$ to Y_L;
    else
      Append $Y[i]$ to Y_R;
    end
  end
    root = new Node(X[l].x, X[l].y);
end
if *myaxis==True* **then**
  ; /* This check if we are in the leaf node or not */
  $l \leftarrow \frac{Y.lenght}{2}$; /* Get the index of the median */
    allocate X_L, X_R, Y_L, Y_R : point arrays of size l
    Copy $y[0 : L-1]$ to Y_L
    Copy $Y[L+1 : Y.length]$ to Y_R
  for $i \leftarrow 1$ $Y.length$ **do**
    if $X[i].y \leq Y[l].y$ **then**
      Append $X[i]$ to X_L;
    else
      Append $X[i]$ to X_R;
    end
  end
    root = new Node(Y[l].x, Y[l].y);
end
root.left = BuildKdtree(X_L, Y_L,not myaxis),
root.right = BuildKdtree(X_R,Y_L,not myaxis)
  return *return node*

3 Kd-Tree Implementation with OpenMP

We parallelized the following part of our algorithm using OpenMP.

Algorithm 3: Part of algorithm 1 parallelized using openMP

if *myaxis*==0 **then**
  $X \leftarrow Sort(X, axis = x)$; /* Sorting the data according to x axis, this was done in parallel */
  .
  .
  .
end
.
.
.
The parallel region is created here using the pragma open parallel directive to assign the following instruction to a different task.
We then use pragma open omp single nowait to creat the task;
root.left = BuildKdtree(X_L,not myaxis),; /* This was assign to one task */
root.right = BuildKdtree(X_R,not myaxis) ; /* This was assign to another task */

3.1 Sort Algorithm for y Axis in Parallel Region

To sort our data on the x axis, we use the *sort()* function built into c++. On the y axis, first, we swap the data points, this was done in a parallel region(see the code below). We then sort the data and swap it again in a parallel region. We swap the axis because the build-in function *sort()* only works with the x-axis. (see the code bellow).

```
if(myaxis==true){ // we sort according to y axis
#pragma omp parallel shared(vect,l,m)
{
        // 1. Let's swap the vector
                #pragma omp for ordered
                        for(int i=0; i<m; i++){
                                #pragma omp odered
                                swap(vect[i][0],vect[i][1]);}
        // 2. sort the swap vector

        sort(vect.begin(),vect.end());
        // 3. swap again the vector
                #pragma omp for ordered

                for(int i=0; i<m; i++){
                        #pragma omp ordered
                        swap(vect[i][0],vect[i][1]);}

} // close pragma
```

3.2 Task creation

Each node of the tree had a separate task assigned to build the right and left subtrees at the same time. Pragma omp single nowait directive was used because the nowait clause allowed other threads to skip the implicit barrier at the end of a region and wait here to be assigned to a task (see the code bellow).

```
#pragma omp parallel
{
        #pragma omp single nowait // Due to nowait claude, all the threads skip the
        //  implied barrier at the end of single region and wait here for being assigned a task
        {
        #pragma  omp task
                newnode->left = kd_tree(left,!myaxis, compt);
#pragma omp task
{
if(right.size()>0) // this condition is use to avoid dumped core because, for 2 data, right=empty
        newnode->right= kd_tree(right,!myaxis, compt);
}
```

4 Kd-Tree Implementation with Hybrid MPI-OpenMP

MPI uses distributive memory devices, which enable each parallel process to work with its own memory space in isolation from others. A recursive algorithm is used to construct the right (left respectively) subtree of a giving node. During our hybrid implementation, instead of directly assign the right and left subtree construction to the task , we firstly use the message passing interface to send the data amount process after sorting the data and selecting the median. Process 0 send the X right portion of the data (see algorithm 1) to process 1 to construct the right subtree and remain with X left for left subtree construction; process 1 send the X right of the data received to process 4 and remain with X left to construct the left child of the subtree it represents; and process 0 send the X right of the data. The process schema can be found on firgure 2. Once this process is completed, we use OpenMP (section 3.2) to create tasks on each process for subtree construction.

Before beginning to build the tree, processes $1, 2, 3$, and 4 are in receiving mode. When process 0 finishes generating the dataset, it sends a right data from the median to process 1. Once finished, it sorts the left portion of the data, then sends the right from the median to process 3. Process 1 did the same with process 4. After a while, all of the processes began to construct their own subtree.

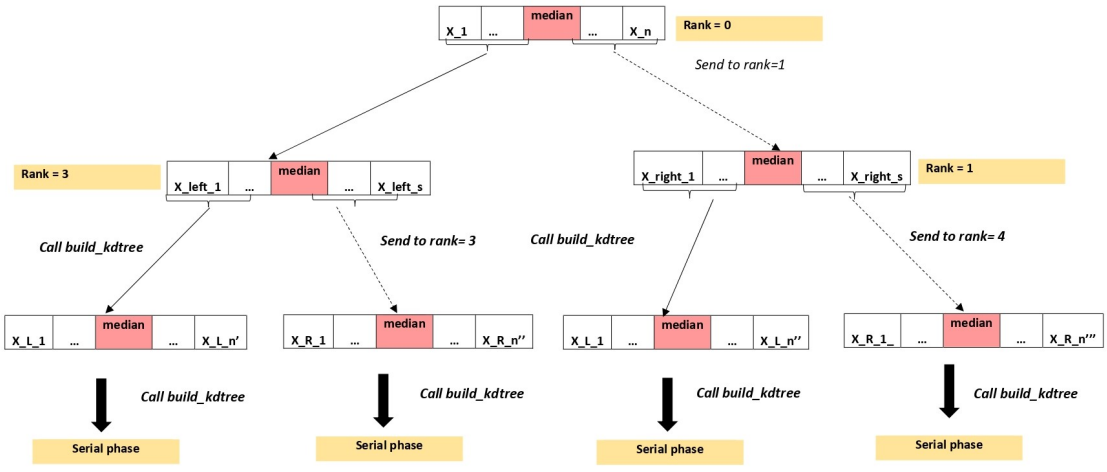

Figure 2: Data distribution process across the processes for Hybrid MPI-OpenMP

5 Performance Model and Scaling

The execution time of a parallel program can help determine which constraints (size of the problem or number of processors) are impacting higher performance and how much improvement can be achieved by adding more processors. We will examine three metrics for performance: speedup, which measures the ratio between sequential and parallel execution times; efficiency, which measures the speedup multiplied by the fraction number of processors used; and scalability, which measures the ability of a parallel system to perform better as more processors are added.

5.1 Scalability

Scalability concerns are strong when working with a large number of threads on a fixed problem size. It is tested by running the code with different numbers of threads, while keeping the value of n constant. The graph (figure 3) shows that as the number of threads increases, the overall program running time for our problem decreases exponentially for a given size n. The plot also shows that the serial code (thread number= 1) with $n = 8$ takes 150 minutes to build the Kd-tree, while the parallel code with 10 threads takes about 14 minutes. For $n = 9$, it took 96 minutes for 10 threads to build the Kd-tree, while 48 threads took 17 minutes. The weak scaling is the speedup for a scale problem size with the respected number of threads (figure 4). In figure 4, we can see that for $n = 4$, the running time exponentially decreases as more threads are added. However, when the thread number exceeds 34, the running time increases, and the overhead of balancing threads increases, so the performance suffers. When we increase the size of the problem and the number of threads, we observe that the running time grows exponentially (weak scaling).

Hybrid programs run at the same speed when using $4, 8$ or 16 threads, suggesting that the model does not improve 5 by increasing the number of thread. Moreover, the hybrid model is the worst.

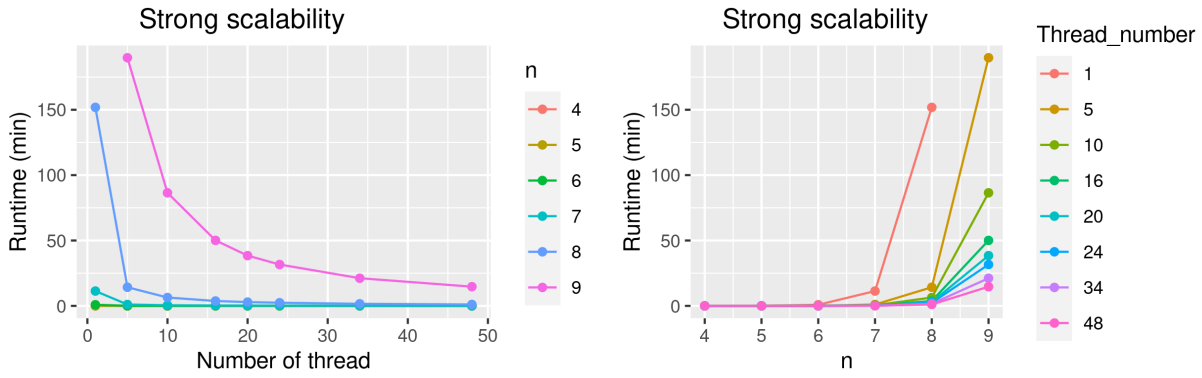

Figure 3: Strong scaling using openmp program

5.2 Performance

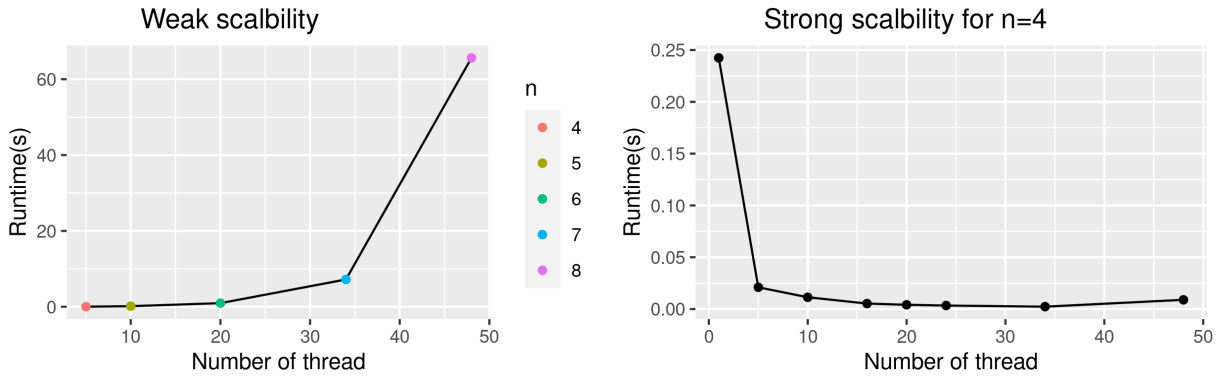

Figure 4: The right curve is the strong scalability plot for $n = 4$ and the left plot is for weak scaling using openmp.

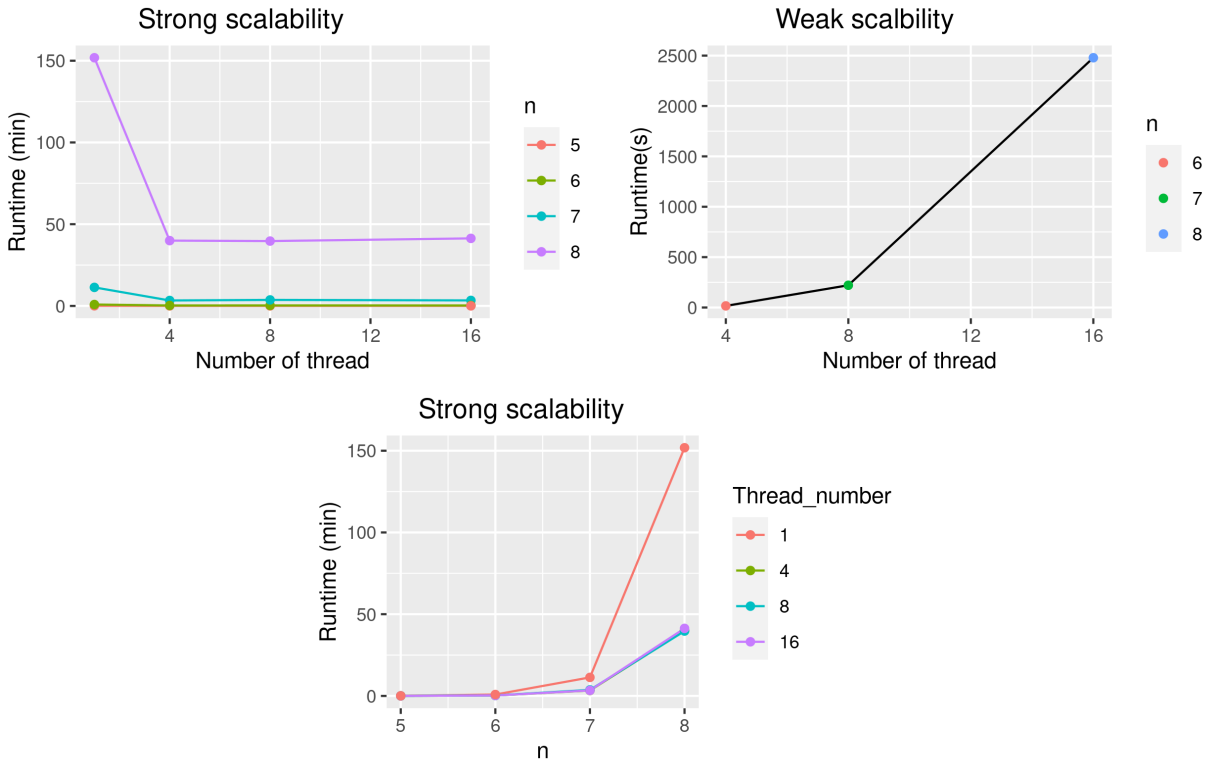

Figure 5: Strong and weak scalability plot for hybrid program using 4 processes.

6 Discussion

6.1 OpenMP implementation

The weak scaling measure is done by fixing the problem size per thread, then increasing more and more the problem size, checking whether the running time remains unchanged. For our experiment, it appears that the weak scaling running time increases exponentially with the problem size(thread and n), the weak and strong speed-up linearly increases, and the weak and strong efficiency increase quadratically and tries to reach the plateau with a larger number of threads.

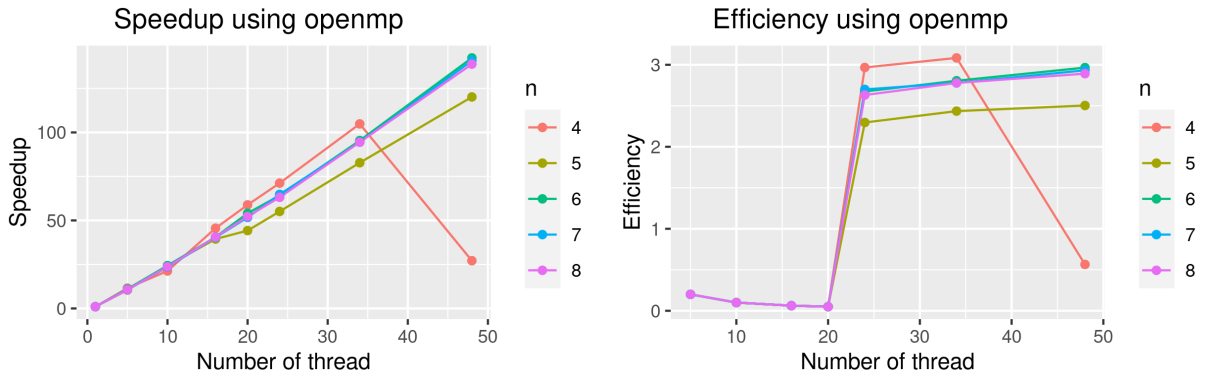

Figure 6: Strong speedup and efficiency using openmp

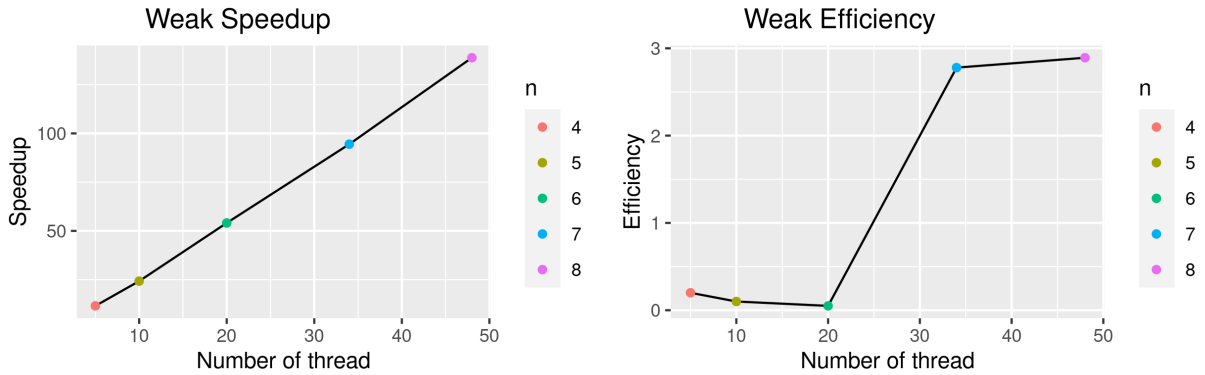

Figure 7: Speedup and efficiency using openmp

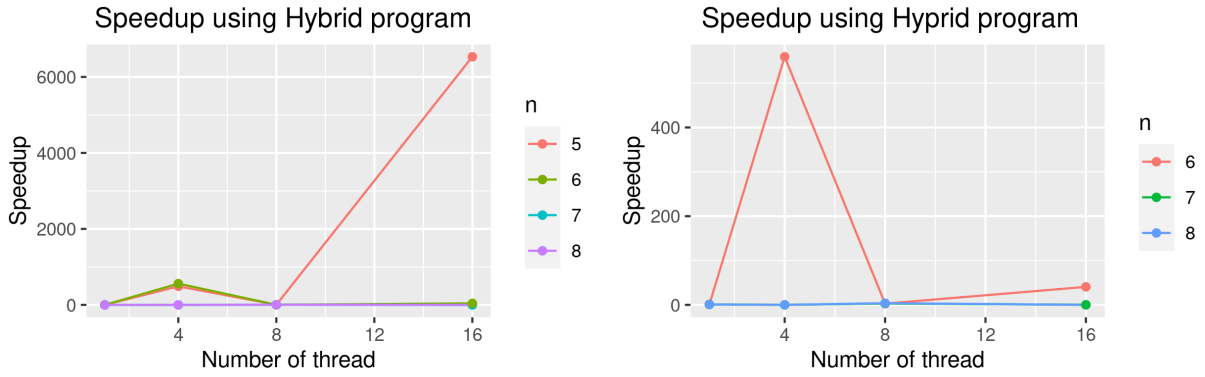

Figure 8: Strong speedup

The result of strong and weak scaling provides an indication for the more or less best match between job size and the number of resources that should be requested for a given size n. However, our OpenMP implementation does not have a perfect speed-up, the possible cause could be the serial code section that was not parallelized, the thread creation and synchronization overhead, the memory access with coherence, the load balancing, or not enough work for each thread.

Another observation was that, for n=4, our experiment demonstrates scaling behaviors up to 34 threads, the overheads of parallelism eventually reach the maximum and then decrease.This is parallel slowdown.

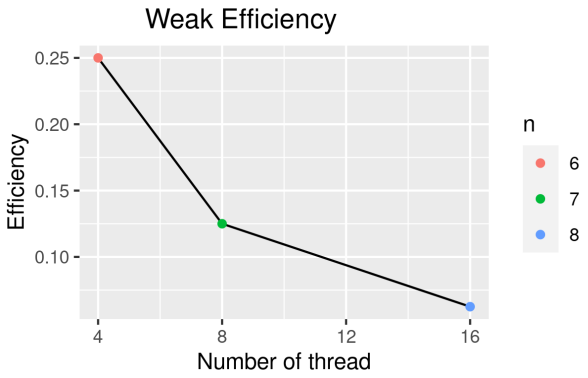

Figure 9: Efficiency using Hybrid program

Adding more threads implies that each thread spends more time communicating than actually doing something useful.

6.2 Hybrid Implementation

The Kd-tree implementation was designed using the linked list structure which is not a contiguous datatype and does not have a pre-defined MPI structure, hence there is no simple manner to send the linked list amount process. Furthermore, each MPI processes have their own address space, therefore, do not share the same memory space. A pointer to an address in another process's space is useless and not possible since they communicate only by message passing interface. So, classically implementing a linked list isn't exactly possible in MPI unless we want to do a lot of work that does not make sense (Sending one element at a time, rebuilding the linked list in the receiving rank, alternatively, we could convert the linked list to an array, send it, and then construct a new linked list in the receiver). Consequently, we did not merge all the subtrees constructed individually by each process.

Furthermore, the hybrid implementation we did is worst than the OpenMP implementation stemming from the fact that the process spend more time in communication (sending data) than actually building a KD-tree. The load imbalance is highly significant when implementing the KD-tree using MPI because there are significant dependencies amount processes (see section 4).

When compared to the OpenMP program, the hybrid program performs the worst. As the problem size and the number of threads increase, the program's efficiency decreases figure 9. With 4 threads, the hybrid program speeds up, but with more threads, there is a parallelism overhead figure 7. The optimal number of threads is then 4.

Optimum ways to improve the propose code

At every recursive piece of code for each subtree, our design code sorts the data; removing this process could result in a perfect speedup by removing recursive sorting. To accomplish this, the second algorithm we presented before could be useful. During the implementation of that algorithm, we encountered a segment fault when creating the serial code, especially when using $n > 4$. Debugging that piece of code and parallelizing it could improve performance since it is less complex than the current one.