Parallel k-d tree

Second assignment of Foundations of High Performance Computing

Walter Nadalin

21/03/22

Contents

1 Introduction

In computer science, a k-d tree[1] is a **space-partitioning data structure** for organizing points in a k-dimensional space [3]. A useful application of this data structure is the m nearest neighborhood search. This algorithm won't be implemented in this work. However, a method for obtaining any element of the tree is given even though, as will be illustrated, the tree will be scattered among several processes that do not share the same memory.

In this assignment we are required to write a parallel code that builds a kd-tree for $k = 2$. We must implement both the **MPI** and the **OpenMP** version or, alternatively, we could go for a single hybrid version. This last path is followed in this work.

In order to simplify the task, the following *2 assumptions* are allowed:

- the **dataset**, and hence the related kd-tree, can be assumed **immutable**: so we can neglect the insertion and deletion operations;
- the **data points** can be assumed to be **homogeneously distributed** in all the k dimensions;

moreover to do the parallel versions it is possible to use only a number of tasks that is a power of 2.

More information are reported at the course's page and in the references written at the end of the report.

2 Algorithm

The algorithm used to build the tree is based on the algorithms described in [2], [1] and it is the following.

```
// Function which build the tree
build(node, dataset, first, last) {
  // Condition to exit from the recursion
  if(size < 2) {          // If the dataset has only one point
    node.point = dataset // Add that point to the current node
    return               // Stop the recursion
  }

  // Looking for the direction in which to cut and the pivot
  node.axis = most_spread(dataset, first, last)        // Direction of maximum spread
  node.point = find_median(dataset, first, last, axis) // The split point is the median

  // Conditional construction of the left node
  if(size > 2) {                              // If we have more than 2 points
    build(node.left, dataset, first, median) // Building the left node
  }

  // Recursively calling the function
  build(node.right, dataset, median, last) // Building the right node
}
```

Notice that, in our case, finding the median on a given direction d can be simplified to taking the mid element of the sorted dataset along that direction, more details later. This is due to the assumption that the data points can be assumed to be homogeneously distributed in all the $k = 2$ dimensions.

[1]Short for k-dimensional tree.

3 Implementation

The implementation of the algorithm has been done in `C++`: this language has been preferred to `C` because it allows to write classes and structs in a more handy way. Notice that the syntax and the features of the functions contained in the `omp.h` and `mpi.h` libraries, used to implement the parallel version of the code, do not change between this two programming languages.

One last note: the code snippets shown are *pseudo-code* for readability reasons but their syntax is close enough to the real implementation in `C++`. Feel free to skip them while reading as they should (hopefully) not add much to the explanation. They are there to help capture the actual implementation. I think it's better to have them directly after the explanation to do so even though sometimes they are long and ugly.

3.1 Serial version

Let's start discussing the simplest case: implement a k-d data structure using **one process** and **one thread**. This will also be core of the hybrid implementation of this data structure: everything explained here is later exploited in the final version of the code. This means that the functions and classes used in the serial implementation will be basically the same in the parallel implementation apart from some addition to exploit more than one process and thread.

3.1.1 The *dataset* data structure

The first thing that we have to take care is the data: where do we allocate them? How do we find the cut direction? How do we order them? We need more than one copy of them?

A good idea could be to implement a *smart* 2 dimensional array allocated in the heap, that we'll call **dataset**. This would be a dynamic array with an **enhanced interface** that allows it to:

- know how many 2 dimensional points it contains;
- implement the **RAII**[2]: in this way we don't have to remember to free its memory each time that it goes out of scope;
- find in which direction the points that it contains are more spread;
- *sort itself* on a given direction **in place**: so we don't need another copy of it.

The first two features of our *dataset* are relatively simple to obtain using a **struct**, a **constructor** and a **destructor**. They are even easier to obtain if we use members which implement the RAII[3]. This also allow us to use the **default move semantics** saving us some time in the parallel implementation.

```
// Signature of the dataset data structure
template <typename T> struct dataset {
  // Members of our class
  unsigned long cardinality // Number of points in the dataset
  unique_ptr<T[]> points    // Smart pointer which points to an array of type T

  // Default constructor and destructor
  dataset() = default
  ~dataset() = default

  // Custom constructor: constuct a dataset with value points allocated
  dataset(value) : cardinality{value}, points{new T[2 * value]} {}
}
```

[2]Resource Acquisition Is Initialization.
[3]Such as the *smart pointers* of the **STL**.

Finding the **most spread direction** is another story but for sure it is easier than sorting the dataset in place. So let's continue with it. The steps are simple and could be intuitively generalized in the case in which we have more than one direction:

1. *find* the **maximum** and **minimum** coordinate in each direction;
2. *computing* the spread in each direction as the **difference** between the maximum and the minimum;
3. *comparing* the values obtained and *find* the **minimum**;

notice that to find the maximum and the minimum for each direction in 2 dimension we just need to **iterate twice** through our data structure.

At last the **sorting**. So, it sure would be cool to do it **in place**. And for sure it would be nice to be able to **select only a given portion** of the array to be sorted: this would help a lot in the recursive construction of the tree. One possible way to achieve this is by using a quick sort algorithm in which the partition step is done by exploiting the solution to the **Dutch national flag problem** [4]. This solution provides a method for *sorting the data with respect to* a given **pivot**: with **just one iteration** through our array we are able to put all smaller (bigger) elements of the pivot to its left (right) in the data structure. With this algorithm properly implemented and modified, in order to select only a portion of the array, the sort in place is doable with the following steps:

1. *select* a **direction** and a **first** and **last element**, that do not necessarily have to be the first and last element of the array;
2. *select* as **pivot** the last element of the ones selected considering the direction selected;
3. **recursively** *apply the algorithm* on the lower and upper halves (with respect to the pivot) always along the same direction.

```
// Sorts all the elements between the first and the last chosen along the given direction
sort(dataset, direction, first, last) {
  // Condition to exit from the recursion
  if (last <= first) { // If we don't have any more elements to sort
    return             // Stop the recursion
  }

  // Variables needed
  lower = first, upper = last, current = lower // We start from the bottom
  pivot = dataset[upper][direction]            // The pivot takes the value of the last

  // Adapted solution to the Dutch flag problem
  while (current < upper + 1) {                      // Go once through all the elements
    if (dataset[current][direction] > pivot) {       // If the current value is greater
      swap(current, upper--)                         // Swap it with the upper element
    } else if (dataset[current][direction] < pivot) { // If it is smaller
      swap(current++, lower++)                       // Swap it with the lower element
    } else {                                         // If it is equal
      ++current                                      // Just consider the next element
    }
  }

  // Here the points with a value smaller (greater) than the pivot are on its left (right)
  sort(dataset, direction, first, lower - (lower > 0)) // Recursion on lower half
  sort(dataset, direction, dir, current, last)        // Recursion on upper half
}
```

Note that this code *also works with* **repeated values**, that is, if two different points have the same coordinates along some direction.

3.1.2 The *node* data structure

A **tree** is a data structure *composed by* **nodes**. Each node could have at maximum 2 **children**: one on the left and on of the right. Each node has a **parent** except the **root node** from which the tree branches. In our case each node has to store the following information:

- **a 2 dimensional point** in the space: this point will be a 2 dimensional array which will contain the coordinates of the point along the 2 directions;
- the **direction** (axis) **of the cut** along which the points contained in the sub trees branching from the node itself are split;
- which are its **left and right children**: this information will be contained in two pointers, each one related to a different child.

Again, by using as members built-in types and classes which implement the RAII we don't have to take care to free the memory related to a node because the **memory** will be **released automatically** at the end of the program through a call to the destructor: less work for us and especially less probability of inserting a bug because we forgot to free memory somewhere.

```
// Signature of the node data structure
template <typename T> struct node {
  unsigned short axis    // Direction of the split
  array<T, 2> point      // Point contained in the node
  unique_ptr<node> left  // Pointer to the left children
  unique_ptr<node> right // Pointer to the right children
}
```

The construction of the tree can be implemented as a node's **method**, i.e. a function inside the class node. This function, given a dataset, will **build the tree** and *return* a pointer to its **root node**. The advantage of using a method is that we have direct access to the members of the node. In this way we can intuitively implement recursion as in the previously reported algorithm following these steps:

1. if the dataset **size** is strictly **smaller than 2**, i.e. the dataset has only 1 points, then put the point in the current node and stop the recursion;
2. *find* the **direction** d on which the points are most spread;
3. if the data are not yet sorted along the direction d then **sort** them along that direction;
4. *find* the **median** (split point) as the mid point of the sorted array and put it in the current node;
5. if the size is strictly greater than 2 then *construct* the **left node** and *call* **recursively** the method considering only the elements to the left of the median;
6. *construct* the **right node** and *call* **recursively** the method considering only the elements to the right of the median[4].

The choice **not to build the left node** if we are left with two points is made for the following reason: suppose we have $2^n + 2^{n-1}$ nodes with $n \in \mathbb{N}$. If we also build the left nodes when we are left with only two points we would build 2^{n+1} nodes in total, i.e. we would build 2^{n-1} nodes for nothing **wasting** a lot of **resources**[5]. Also, suppose we choose to build them, then what should we put in them? Should the be invalid nodes? How do we know that they are invalid nodes?

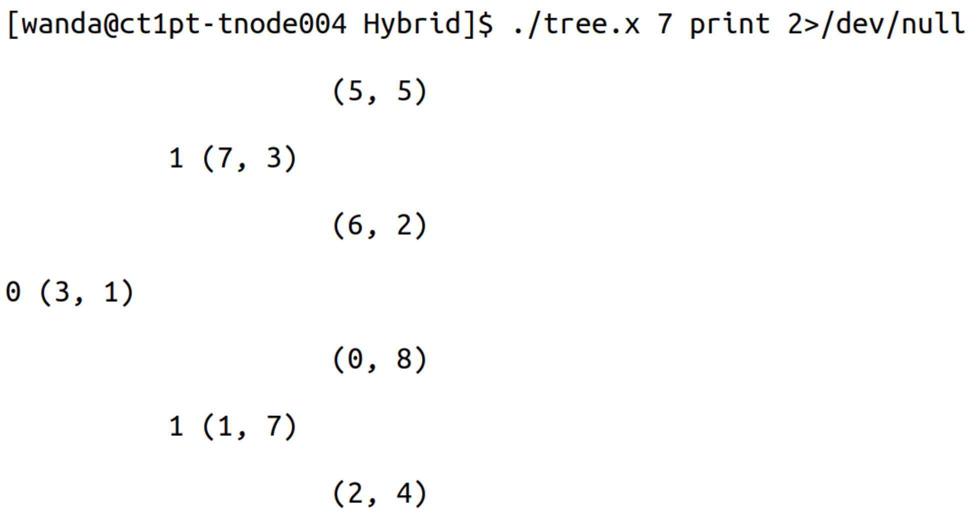
Now, once we have implemented the code and corrected all the syntax errors, how can we **debug** it from the *logic errors*? That is to say, how can we make sure that our algorithm builds correctly the tree?

One way could be to print the tree directly on terminal 1: in this way we have a fast and cheap way to check its correctness. In order to do it we can exploit recursion.

[4]Note: this algorithm doesn't return anything, for this reason there is a wrapper function defined outside the class node that takes care of allocating the root node, building the tree from it and returning a pointer to it.

[5]The resources intended here are the memory occupied and the time to the solution.

Figure 1: **visual representation of a tree** obtained with the serial implementeation considering 2 dimensional points with integer coordinates. For each node is printed the direction of the cut (if the node is not a leaf) and the point's coordinates. Indeed, with a little bit of patience, one could verify that given these points the tree which is obtained in the figure is correct.



```
[wanda@ct1pt-tnode004 Hybrid]$ ./tree.x 7 print 2>/dev/null

                    (5, 5)

          1 (7, 3)

                    (6, 2)

0 (3, 1)

                    (0, 8)

          1 (1, 7)

                    (2, 4)
```

3.2 Parallelization

Nowadays (almost) each one of us has access to a **parallel computer**: the power of a single computational unit is not enough anymore. A parallel computer is a computational system that offers *simultaneous access to* **many computational units** fed by some memory units. Orchestrating properly the work of these different units we can achieve more computational power. In this framework enters **parallel processing** which is the ensemble of techniques and algorithms that makes you able to actually use a parallel computer to successfully and efficiently solve a problem.

We may want to **parallelize** the construction of our k-d tree for 2 reasons:

- we may want to **optimize the time** required to build a tree:
  - either we would like to build the *same tree* in a *smaller time*;
  - or we would like to build a *larger tree* in the *same time*;
- we may want to **increase the size** of a tree: maybe a very large tree could not fit in the memory addressable by a single computational unit.

3.2.1 Shared memory

Going parallel, one possibility is that our computational units *share* the **same memory**. In this case to parallelize our algorithm we may decide to exploit **OpenMP**, which is a representative of the shared memory paradigm. In this paradigm we just have **one process**[6] that *spawns* **many threads**. Each thread has its own stack and can be run by a different processor. However all the threads share the same memory space, i.e. the **heap is unique** among them. This also means that the processors involved in the execution of the program have access to the same memory.

In particular, OpenMP offers us a *special kind of parallelism*: the **tasks**. Each task is a **unit of work** and we may think of it as a *description*[7] of the work that must be done. So, in this case, we decompose our work in tasks which are **parallel** and **can be executed by anyone**, i.e. by any thread.

[6]Therefore an unique memory space and only one copy of the code.

[7]A task is a description of work, comprehensive of the instructions and the data, in which there's written that someone, whoever will be, has to run those instructions with those data.

In our case, we can notice that the *creation* of a certain node is **independent** on the creation of other nodes on different branches or on the same level of the tree. The construction of a node depends only on its parents and grandparents: once they are all created, it can be built independently from any other node. Its construction is even independent from the creation of its brother or sister, because it will consider a different portion of the data.

For this reasons we can create **one task for each node**. In practice, one thread will execute the function to build the tree and will *create* **two sticky notes**, one for the right child and one for the left child, with written: *please create a tree with this data branching from this node.* Then the first available threads will pick those sticky note up and follow the instructions. In doing so they will repeat the same operation creating two others sticky notes.

```
#pragma omp task shared(dataset) firstprivate(size, axis, first, median)
if (size > 2) {                                          // With 3 or more points
  left.reset(new node{})                                 // Allocates the left node
  left->build(dataset, first, median - (median > 0), axis) // Builds a tree from it
}

#pragma omp task shared(dataset) firstprivate(axis, last, median)
{
  right.reset(new node{})
  right->build(dataset, median + (median < last), last, axis)
}
```

Note that **different threads** work on the **same dataset**, they just consider different portions of it. so the dataset must be **shared** between them, i.e. all threads will have access to the same region of memory. This could create, in general, a conflict but, in this particular case, such a thing does not happen. In fact, as we said, node creations are independent of each other and each thread always works on a different portion of data with respect to all the others.

Now, as almost everything else in OpenMP, a task must be *generated inside* a **parallel region**. Moreover, to guarantee that *each task* is **created only once** we have to make sure that only one thread creates a specific task. Therefore the first time that we call the function to build the tree we do it inside a **single region**. Subsequent times, since only one thread will perform a certain task and thus build the node, a task creation will also be protected.

```
// Wrapper function which exploits the function overloading
build(dataset) {
  tmp = (new node{})                       // Allocates the root

#pragma omp parallel shared(dataset)       // The dataset is shared
#pragma omp single                         // The tasks creation is protected
  tmp->build(data, 0, data.cardinality - 1, 2) // Calls the void method

  return tmp;                              // Returns a pointer to the root
}
```

To conclude, this implementation gives us the possibility exploit better the time required to build a tree but it has **some limits**: how many processors can share the same memory? What if our processors don't share the same memory?

Moreover, we **don't** have the ability to **increase the size** of the tree from a certain point on, even if we are using more than one processor, because all the computational units we are exploiting can only access the same shared memory.

3.2.2 Hybrid

We can *overcome this limitations* exploiting more processors that do not necessarily share the same memory. In this case we will work with a **distributed memory** paradigm. An implementation of it that we'll use is **MPI**.

In the distributed memory paradigm we have a **number of processes** that are *created*. Each process has its own memory space that is not accessible to other processes. Therefore we have several stances of the same program and several copies of the same code. Indeed, the fact that they don't share the same memory is a problem that we have to overcome somehow through **point-to-point communication**. But, once we do that, we get in return that the **memory will be scalable** and we can work with as many processors as we want[8].

However, **spawning threads** within processes is much **more convenient** than spawning processes, because there are far fewer operations to do in the first case. So, if we want to get the most out of our parallel machine, one idea might be to go with a **hybrid approach** where we use *both OpenMP and MPI*: we create one process for each group of processors sharing the same memory and let that process spawn as many threads as there are processors sharing that memory.

As for the hybrid implementation, everything we have done so far is still valid, but we have **two major challenges** to overcome when using distributed memory: how can we properly *split* the **tree-building** operation (hence the data) across different processes? How can we *travel* through the tree scattered in different locations that do not share the same memory[9]?

Let's discuss a possible answer to the first question supposing to work with a **number of processes which is a power of 2**. Let's start with the simplest case: *only 2 processes*. Let one process be called **master** and the other **slave**. In this case one can simply assign the *construction of* the **left branch** of the tree (with respect to the root) to a process, let's say the master, and the construction of **right branch** to the other. We can devise the following strategy:

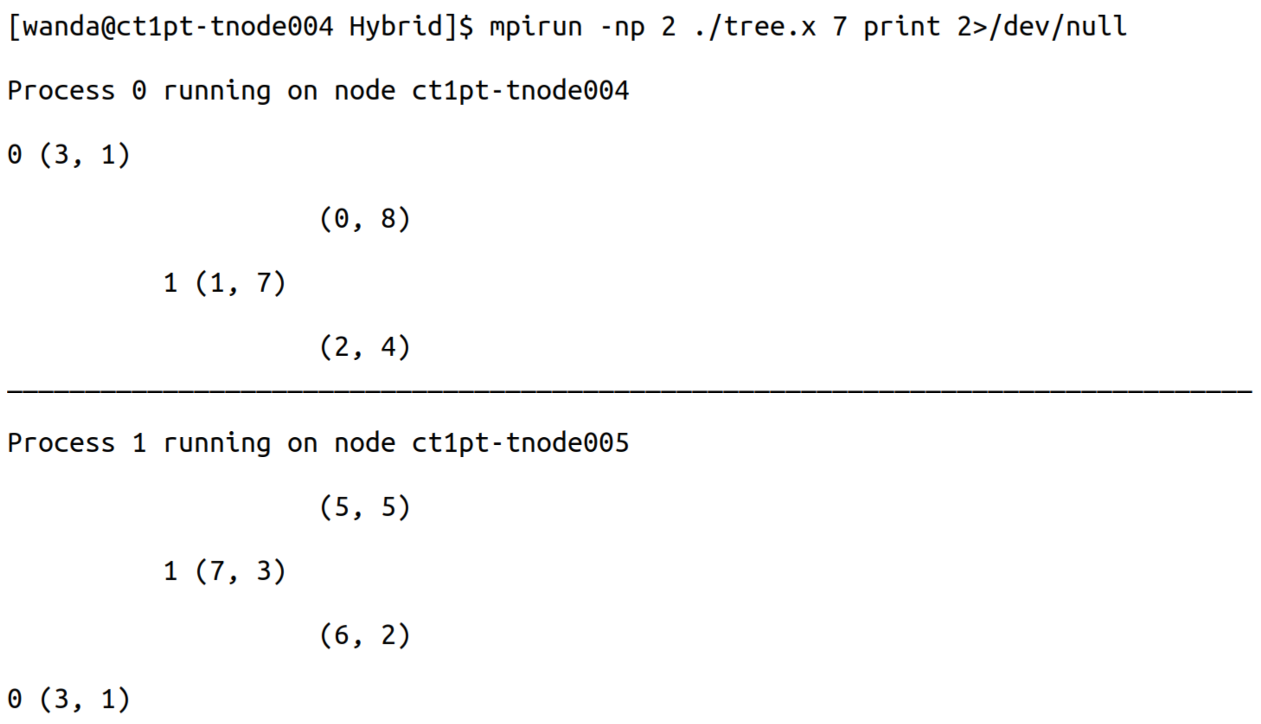
- the master:
    1. *allocates* the **entire dataset** in its memory;
    2. *finds* the **direction** along which the points are more spread;
    3. *sorts* the data along the direction found at the previous point and *finds* the **median**;
    4. *creates* the **root node** and *sends* its information (that is to say its point and its axis) to the slave which owns a personal copy it;
    5. *splits* the dataset in **two chunks**: one that contains all the points to the left of the median in the sorted array and the other which contains the remaining points;
    6. *keeps* one half (the lower one to keep things consistent) for itself and *sends* the other one to the slave;
    7. *builds* the **tree starting from the left children** of its root with the data that he has kept for itself;
- the slave:
    1. *receives* the information about the root and saves them into its own copy of it;
    2. *receives* the **upper half** of the splitted dataset;
    3. *builds* the **tree starting from the right children** of its root with the data that he has received from the master.

This strategy works as intended as we can see in the picture 2 which shows the same tree built in 1 but using two different processes on two different nodes. Note that with this procedure we will end up with a **tree** which could be **scattered** among different memory units.

[8]Hardware limitations aside: in theory we could work with as many processors as we want, but, in reality, we'd probably have a finite number of processors to work with anyway.

[9]The solution to this second problem will be discussed in the appendix since it was not explicitly required in the assignment. Take a look at it if you like: it works fine.

Figure 2: visual representation of the tree scattered among different processes. The **process 0** is considered to be the **master** and owns the left branch of the tree while the **process 1** is the **slave** and owns the right branch of the tree.



```
[wanda@ct1pt-tnode004 Hybrid]$ mpirun -np 2 ./tree.x 7 print 2>/dev/null

Process 0 running on node ct1pt-tnode004

0 (3, 1)

                    (0, 8)

          1 (1, 7)

                    (2, 4)
________________________________________________________________________________

Process 1 running on node ct1pt-tnode005

                    (5, 5)

          1 (7, 3)

                    (6, 2)

0 (3, 1)
```

Now we have to **generalize** this procedure considering a number 2^n of processes with $n > 1$. In order to do this we can *exploit* **recursion**. Before diving into the implementation, a couple of **remarks**:

- at *depth* d we have a number 2^d of nodes;
- **each node** could be considered the **root** of a tree which branches from it;
- let s be the *size* of the **available process pool**: at the beginning $s = 2^n$;
- let's assume that the **entire dataset** is **already** been **allocated** by the process 0 before the beginning of the following algorithm.

With this observations and assumption, an idea to generalize the parallel construction of the tree considering more than 2 processes could be the following:

1. let the process $m = 0$ be the **master** and the process $m + s/2$ be the **slave**: they perform the **same operations** illustrated in the case with only 2 processes but they *won't build the tree*[10];
2. the **master repeats recursively** the first step with a size $s \to s/2$ and considering as new root of the tree the *left child* of the current root.
3. the **slave repeats recursively** the first step as a master (so with $m \to s$), with a size $s \to s/2$ and considering new root of the tree the *right child* of the current root.

Of course we need a **criterion to stop** this procedure: we stop when the size at the current iteration is $s = 1$. When a process realizes that there are no more processes available to do another iteration it will perform the send or receive procedure, stop the recursion and return the current root. Then it will simply build the tree from the root that has been returned. In this way, **each process** is *assigned a* different **node** of depth n from which to build the tree and a **different portion of the dataset** to consider.

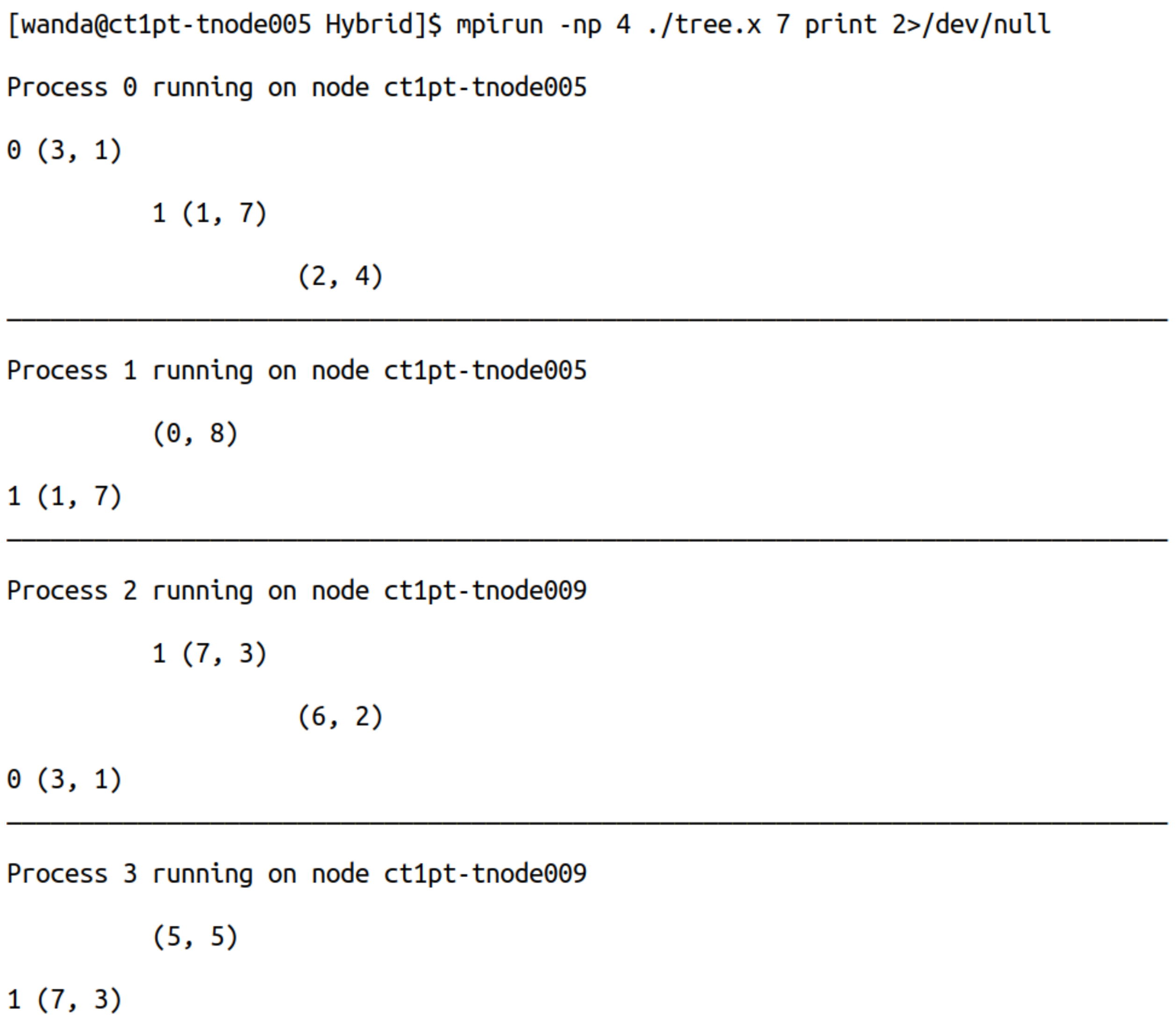
[10]And the $m = 0$ process does not allocate the entire dataset again since it is assumed that it has already been allocated previously: it will just split what it has available.

As an example suppose to have $2^n = 2^2 = 4$ processes. Then at the end of the recursion, i.e. when the execution of the function just reported terminates, we will have that:

- 0 builds from the **left child of the root's left child** with the first quart of the dataset;
- $2^{n-2} = 1$ builds from the **right child of the root's left child** with the second quart of the dataset;
- $2^{n-1} = 2$ builds from the **left child of the root's right child** with the third quart of the dataset;
- $2^{n-1} + 2^{n-2} = 3$ builds from the **right child of the root's right child** with the remaining part;

indeed this is what happens in 3 where we build the same tree as in 1 but we use four different processes on two different nodes. Note that in 3 we end up that each process will have to build a tree considering a dataset containing only one point. This happens because we have an initial dataset with only 7 points. This means that the leaves of each branch displayed in the figure are actually the root nodes of the various processes. From these roots each process will build its own sub-tree in case we have a larger initial dataset.

Figure 3: visual representation of the tree scattered among four different processes. As we can see each process owns a different branch of the tree and has one ore more **nodes in common** with some other process: this is the **key** to allow **to travel through the tree** scattered among different memory regions as explained in the appendix.



```
[wanda@ct1pt-tnode005 Hybrid]$ mpirun -np 4 ./tree.x 7 print 2>/dev/null

Process 0 running on node ct1pt-tnode005

0 (3, 1)

          1 (1, 7)

                    (2, 4)
________________________________________________________________________________

Process 1 running on node ct1pt-tnode005

          (0, 8)

1 (1, 7)
________________________________________________________________________________

Process 2 running on node ct1pt-tnode009

          1 (7, 3)

                    (6, 2)

0 (3, 1)
________________________________________________________________________________

Process 3 running on node ct1pt-tnode009

          (5, 5)

1 (7, 3)
```

4 Performance model and scaling

At this point we have to study the performance. In theory, the parallel version should allow it to **speedup** if we did things properly. Therefore we'd like to measure how much the program speeds up, that is to say, how much the *time-to-solution decreases.* However makes an implementation good or bad is not just how much the program speeds up: we also have to consider how much it will speedup *with respect to* the **resources** used. Indeed, performance is a measure of how well the computational requirements are met and, at the same time, of how well the computational resources are exploited. Hence, in the following analysis we'll consider 2 quantities, the speedup of the code which is given by:

$$Sp(n,p) = \frac{T_s(n)}{T_p(n,p)} \tag{1}$$

and its **efficiency**:

$$Ef(n,p) = \frac{T_s(n)}{p \cdot T_p(n,p)} = \frac{Sp(n,p)}{p}$$

where:

- n is the **problem size**, in our case the number of 2 dimensional points;
- $T_s(n)$ is the **serial run-time**;
- $T_p(n,p)$ is the **parallel run-time**;
- p is the **number of computing units** or processors[11].

We can expand $T_p(n,p)$ as a function of $T_s(n)$ by introducing:

- f_n intrinsic sequential fraction of the problem of size n;
- $k(n,p)$ parallel overhead;

and assuming that parallel fraction of the computation is perfectly parallel:

$$T_p(n,p) = T_s(n) \cdot f_n + T_s(n) \cdot \frac{1-f_n}{p} + k(n,p)$$

notice that as f_n increases the parallel gain decreases and, no matter what, as $p \to \infty$ we have that the speedup asymptotically reaches an **upper bound**.

Before going on with the analysis a couple of remarks on the **hardware** used. I exploited 4 `thin` nodes of the `ORFEO` supercomputer. Specifically they were: `ct1pt-tnode006`, `ct1pt-tnode007`, `ct1pt-tnode008` and `ct1pt-tnode006`. These are Intel CPUs (Xeon Gold 6126) with **2 sockets**, **12 cores per socket** and *hyperthreading disabled.* Using the `likwid` module one can learn that the socket 0 hosts the processors 0, 2, 4, . . . , 20, 22 and the socket 1 the processors 1, 3, 5, . . . , 21, 23. Moreover the cache topology is the following:

- the **level 1** cache has size 32 kB (2^{15} B) and is *not shared* between processors, so each processor has its own;
- the **level 2** cache has size 1 MB (2^{20} B) and is also *not shared*;
- the **level 3** cache has size 19 MB and is *shared* between processors on the same sockets, so each socket has a level 3 cache shared among its processors.

It is also possible to observe that the RAM is shared between processors on the same socket, therefore there are **2 NUMA domains**.

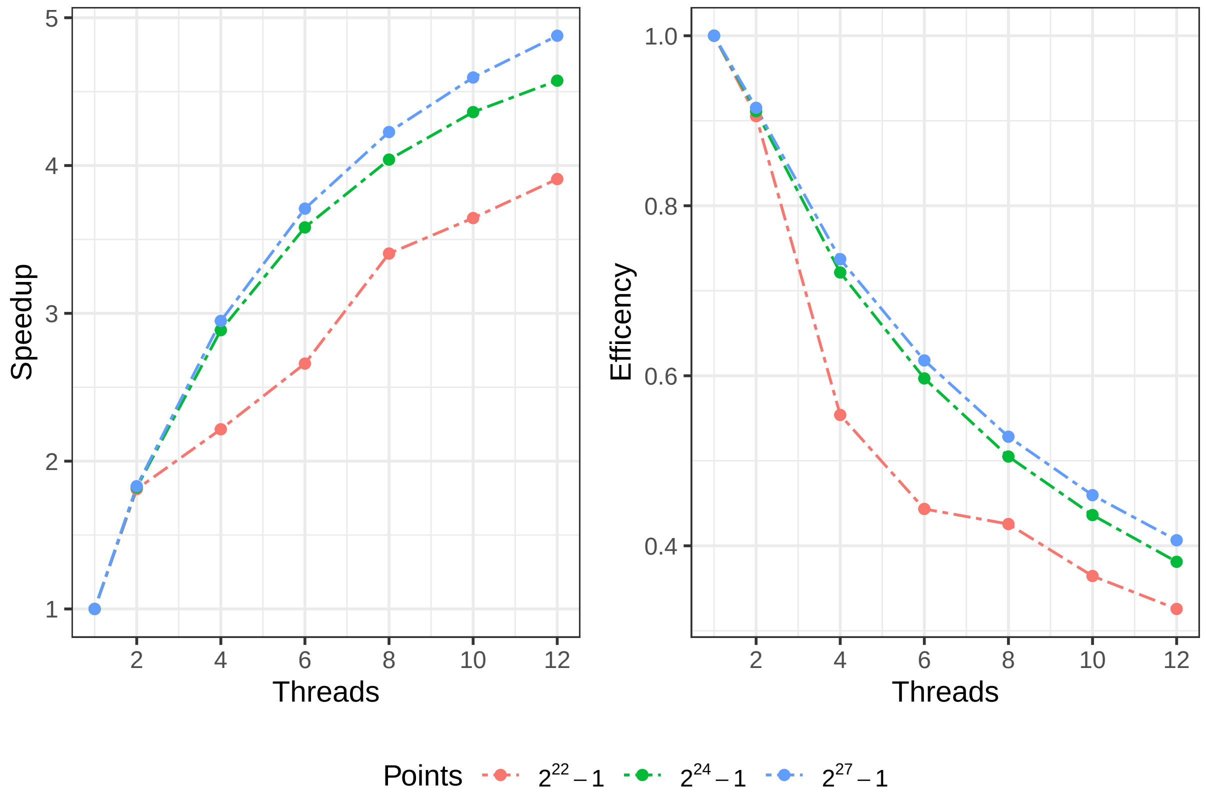
[11]In the following analysis at each thread or process corresponds one processors, for this reason we can say that each processor is a single computing unit.

4.1 Open multiprocessing

The implementation allows to *choose* to work with a **single process or** with one **single thread** respectively running the parallel version with the `mpirun -np 1` or with the `tree.x $points $information 1` flags (where `$points` is the number of points to consider to build the parallel tree, `$information` tells the program what information to print, and `1` is the number of threads). We can leverage this to analyze the speed and effectiveness of the shared memory approach before moving on to the performance of the hybrid implementation.

Let us consider a **single process**, therefore a single socket[12]. Since hyperthreading is disabled we can spawn maximum **12 threads** per socket, each thread will be hosted by a different core. To study the speedup and the efficiency 3 problem sizes are considered: $2^{21} - 1$, $2^{24} - 1$ and $2^{27} - 1 > 10^8$. This numbers are chosen because a full tree with depth d is grown with $2^{d+1} - 1$ points. In 4 the results are reported.

Figure 4: Speedup and efficency with respect to the number of threads spawned by a single process considering various sizes of the problem (strong scaling).



In the picture we can see how the **efficency decreases** as we increase the **number of threads**, hence computational units, involved in the construction of the tree. To explain this behavior let us suppose that we can *neglect the parallel overhead*. This assumption implies that what caused the performance loss is actually the **serial fraction** of the code. Indeed, this is intuitive if we think about the algorithm: the **firsts iterations** that *work with* the **largest portions** of the data are not parallelize. That is to say, the master thread will have to sort all the initial dataset by itself before being able to create two tasks that will be picked up by 2 threads. And also after that the two halves of the initial dataset obtained will be sorted by only 2 threads even if we have 12 or more.

[12]That is a single NUMA domain in this case.

Also, in the figure 4 it can be observed that the **speedup increases** with the **size** of the problem. Certainly the first iteration is the most expensive and its cost increases as the size increases. However, as the problem size increases, subsequent iterations also become more expensive and thus it becomes cheaper to parallelize. Again, this is caused by the fact that from a certain point on, nothing is gained by adding more workers because there will always be a few doing the really expensive part. So, in a sense, as the size of the problem increases, that point also increases. Looking at the picture it is clear that considering $\sim 10^6$ or less points makes the constuction of the tree **not scalable** at all using more than 2 threads.

Now let's try to create a **model** considering, without losing generality, $n = 2^m - 1$ points with $m \in \mathbb{N}$. Let $T_b(n)$ be the time to build the tree using one single thread and one single process. There are 2 expensive operation that we have to perform for each node:

1. *find* the **direction of the maximum spread** (*always*), which has a **linear cost**;
2. **sorting**, which happens only *if* the points are yet ordered along the direction of the maximum spread, and has a cost of $n \cdot \log_2(n)$.

So, for sure we have to order the points for the **root node** and this operation is always performed in **serial**, no matter how many processes and threads we're using. Therefore, the complexity to build the first node is always $n \cdot \log_2(n)$. The following times the sorting may or may not be performed depending on which is the direction of the maximum spread after the split. Let's put ourselves in the **worst case scenario** in which the spread along the 2 directions is equal[13].

Moreover notice that:

- at each iteration we **split the points** and so the cost to build a single node at a depth d will be computed considering $2^{m-d} - 1$ points;
- the **leaves**[14] *doesn't perform* the above operations. Indeed, only $2^{m-1} - 1$ nodes perform them and the ones at the level of depth $m - 2$ are the last ones to do so;
- at a depth d the tree has 2^d nodes and has to perform, in general, for each one the search and the sort.

Therefore, taking $m > 2$, we could say that:

$$T_b(m) \propto \sum_{i=0}^{m-2} 2^i (2^{m-i} - 1) \log_2 \left(2^{m-i} - 1\right) = \sum_{i=0}^{m-2} \mathrm{C}\left(2^{m-i} - 1\right) 2^i$$

where $C(n) = n \log(n)$ is the **cost function** and i is an index that refers to the depth.

Now, Let p_t be number of threads spawned. Suppose, to ease the computation, that $p_t = 2^k$ with $k \in \mathbb{N}$ and $k > m$. In this case we have that, in theory, at the level $k < d$ there are enough threads to perform the above operations for each node completely in parallel. Hence:

$$T_t(m) \propto \sum_{i=0}^{k} \mathrm{C}\left(2^m - 1\right) + \sum_{i=k+1}^{m-2} \mathrm{C}\left(2^{m-i} - 1\right) 2^{i-k}$$

may give an *estimate* of the **time required to build** a tree with $2^m - 1$ points with 2^k threads. Of course one should also consider the parallel overhead given by the queue system which assign at the various threads new work as soon at it is available but we choose to neglet it.

[13]The measures are taken with this hypothesis.

[14]That are 2^{m-1} since we build a full tree having maximum depth $m - 1$ if we consider $2^m - 1$ multi-dimensional points.

4.2 Hybrid

Now to the **performance** analysis of the **hybrid implementation**. As we said previosly the strategy for this one is to create **one process per socket** and make that process *spawn* a certain number of **threads**: each thread will be assigned to a different core. We can achieve this setting the environmental variable `OMP_PLACES` to `cores` and then asking for two process per node pinned by socket with `mpirun -np $processes -npernode 2 --map-by socket` as is done in 5. In this way we can exploit each core of our 4 `thin` nodes and build our tree using up to 96 computational units.

Figure 5: printed information about 2 processes pinned on 2 different sockets of the same `thin` node which spawn 4 threads each: we can observe that they are spawned on different socket by the numbers of the processors on which they are running. This information can be obtained by running the program using `./tree.x $points info 4`.

```
Processor 0 running on node ct1pt-tnode006

4 threads in execution | The places are cores with a close binding policy
------------------------------------------------------------------------------
Additional info for each thread:
Thread number 0 in place number 0 | Processors here: 0
Thread number 1 in place number 1 | Processors here: 2
Thread number 2 in place number 2 | Processors here: 4
Thread number 3 in place number 3 | Processors here: 6
------------------------------------------------------------------------------

Processor 1 running on node ct1pt-tnode006

4 threads in execution | The places are cores with a close binding policy
------------------------------------------------------------------------------
Additional info for each thread:
Thread number 0 in place number 0 | Processors here: 1
Thread number 1 in place number 1 | Processors here: 3
Thread number 2 in place number 2 | Processors here: 5
Thread number 3 in place number 3 | Processors here: 7
------------------------------------------------------------------------------
```

4.2.1 Strong scalability

Recalling what was done in a previous section if f is the intrinsically sequential fraction of code then the **speedup** is:

$$Sp(n,p) \leq \frac{1}{f + \frac{1-f}{p}} \leq \frac{1}{f}$$

neglecting the parallel overhead. This is the **Amdahl's law** and describe what is usually performed as **strong scalability**: we **fix the size** of the problem and increase the number p of processors. In this case, to study the strong scalability we'll consider:

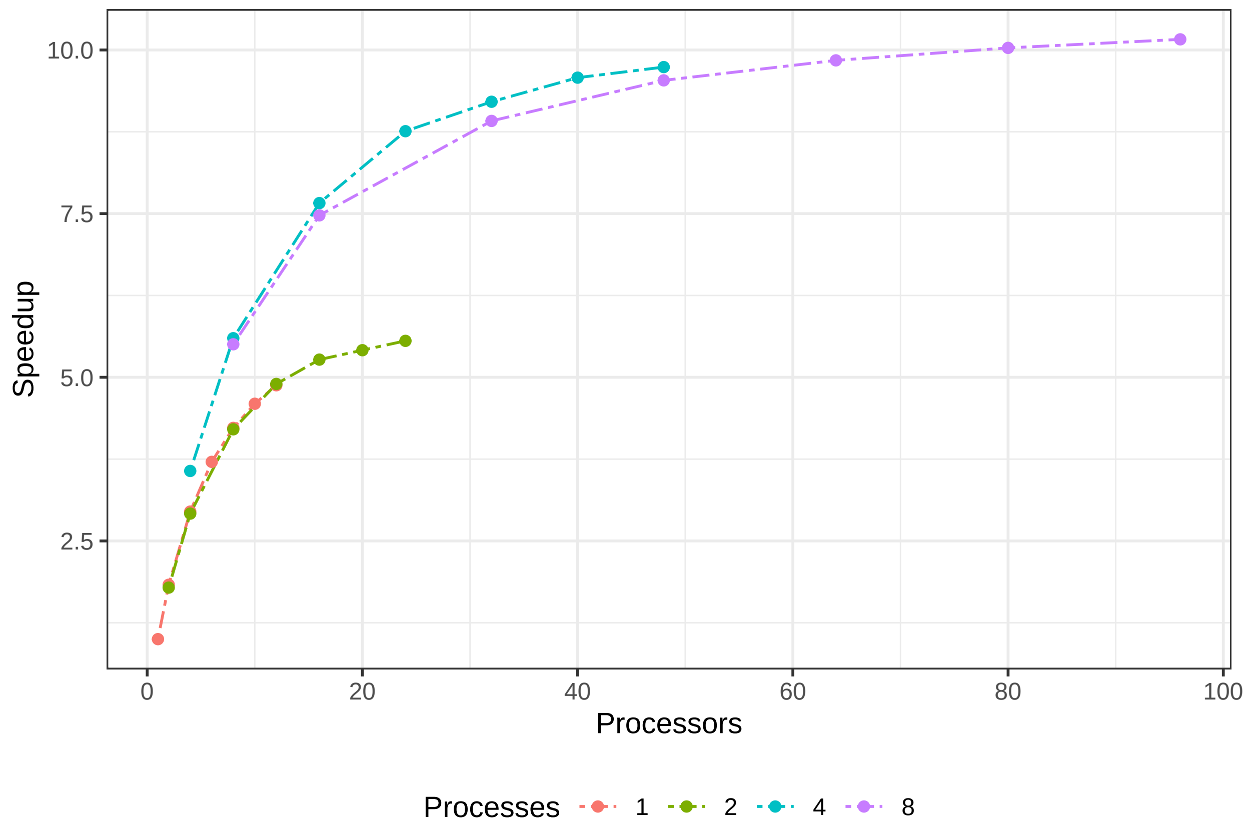
- a number of points $n = 2^{27} - 1 > 10^8$;
- a number of threads $p_t = 1, 2, 4, 6, 8, 10, 12$;
- a number of processes $p_p = 1, 2, 4, 8$.

To visualize this quantity the number of processes is fixed and the number of threads spawned by each process is modified, measuring at each change the time required by the slowest process to build the tree. The **results** are reported in 6.

From the figure below we can observe that for the same number of processors used the executions that consider 1 process have similar results to those that consider 2 processes, as one expects intuitively. This is also true between the runs that consider 4 or 8 processes. However the two runs that consider a number of processes equal to 4 and 8 seem to outperform the other ones when the number of processors increases.

Modelling the behavior of the hybrid code may be tricky since there are a lot of things to take into consideration. In any case, one contribution to what we observe above maybe the **false sharing**. Recall that when we're working in a shared memory paradigm we have only **one shared dataset** on which to work. This means that **many threads** are working on the same region of memory even tough they consider a different portion of it. Supposing to work with an array of 2 dimensional points of which each coordinate it's a `float`, it's possible that two threads work at two different parts of the dataset which are *contiguous*[15]. Hence, when the first thread access and modify the location, to maintain the coherence the cache must *write-back and reflush*. False sharing may be a **major performance killer** and should affect less the performance loss if we **split the dataset** in more parts and less threads are working on each part: this is exactly what happens when we *distribute the dataset* between different processes on different nodes.

Figure 6: Speedup measured with respect to the number of processors considering different numbers of processes that spawn different numbers of threads and keeping the problem size fixed (strong scaling). We can observe similar behaviours when we use 1 or 2 processes and when we use 4 or 8 processes. However the last two cases seem to outperform the first two.



[15]That is to say with part of the array which closer 8 points or less: the size of a cache line is 64 bytes and the size of one 'float' is 4 bytes.

4.2.2 Weak scalability

As we saw, when we increase the problem's size, the *parallelizable part increases more* than the sequential part. If we consider the workload as the sum of the parallel and serial part, and we assign the same amount of workload that would amount to a serial run-time. Hence, we can write the **Gustafson's law** for the speedup:

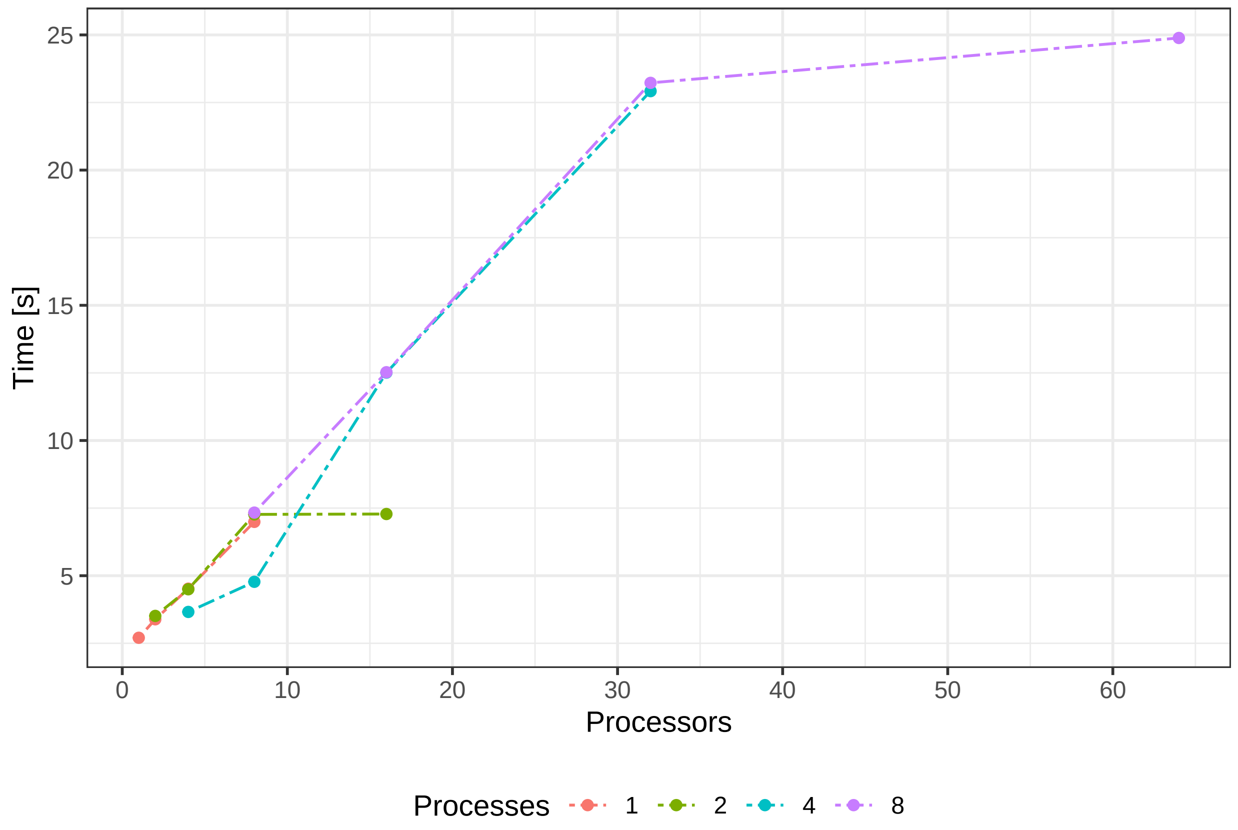
$$Sp_G(n, p) = p - (p - 1)f_n \leq p$$

which is also referred as **weak scalability**. In other words: the **workload is fixed**, while the problem size and p increase. To measure this quantity I considered:

- a size of the problem $n = p \cdot 2^{21} - 1$ where $p = p_t + p_p$ number of processors used;
- a number of threads $p_t = 1, 2, 4, 8$;
- a number of processes $p_p = 1, 2, 4, 8$.

The result obtained are reported in 7.

Figure 7: Time-to-solution measured with respect to the number of processors considering different numbers of processes that spawn different numbers of threads and different sizes of the problem keeping the workload for each processor fixed (weak scaling).



From the image we see that, as expected, the behavior is **worse than ideal**, i.e. the slope of is less than 1. Also a **performance drop** for 2 and 8 processes when considering a number of threads greater than 4 in both cases: I think this is *related to* the fact that the *similar performance* for strong scaling between 1 and 2 or 4 and 8 processes. This denotes that the program no longer scales and may be caused by the less significant communication overhead with 1 or 4 processes.

5 Discussion

It seems like using a combination of **4 processes** that spawns **many threads** may be a *good solution*: in this way can exploit a fair number of computational units without sending much data between different nodes. However this also depends on how much memory we want to occupy on each node: if we want to occupy less memory we may opt to go with 8 processors and spawn less threads.

Before continuing with some possible improvements, a comment on the existing code: **why not round-robin**? Of course, if the data is distributed equally between the two directions it seems that we just do *another useless operation* when we search for the direction in which the points are more spread: each time we would have to sort the data along the other direction anyway. Despite this, I decided to implement the maximum direction search for **3 main reasons**:

- this should **not** be as big a **performance killer** as it seems. Even if we have to do this at each iteration it has a linear cost that is *obscured* by the $n \cdot \log(n)$ cost of sorting: for $n \to \infty$ the contribution of finding the direction of maximum spread becomes negligible and for small n it is very fast;
- suppose that at the beginning **one direction** has the points **much more scattered** than the other: then the first iteration will sort the data along that direction and the following iterations will not, this *improves* drastically the performance of the code;
- it could be that, even if at the beginning the points are equally distributed along the 2 directions, *at a certain iteration* we find ourselves in a situation where one direction has the points much more spread than the other.

Now for some possible **improvements** and a little discussion of why they might or might not work and how difficult they might be to implement:

- we can **preallocate all the memory** needed to store the tree at the beginning instead of doing it independently for each node: in theory this could improve the performance a lot. However, it's *far from easy* to do for one main reason: if we want to preallocate memory we obviously have to preallocate a **contiguous chunk** of memory and then start building nodes one by one along it. I've been able to do it serially, but **parallelization** is really **hard** and can lead to *worse performance* overall. The reason is that we have to **coordinate** writing the node to the end of memory when the tree is built.

- we can **allocate two datasets** containing the same points, each sorted along a different direction. However, doing so *doubles* the **initial cost** of searching and sorting, and we also use twice as much **memory**, which as n increases can lead to not having enough memory to build the tree. Also, what happens if we have more than 2 dimensions?

- there is definitely still a lot of work to be done on **single core optimization** and code cleanup. I focused a bit on that for the dataset class but everything else is the first (or second) working version of the delivered algorithm. So I'm sure there is a lot of room for improvement in many details.

- as we discussed earlier the **first iteration** is always completely and tragically **serial**. This means that one process or thread is doing all the work while the others are watching: so *why not use them* to help? **Parallelizing** sorting in the shared memory paradigm is fairly easy, and I was able to do it with my version: we just have to create a task for each recursion, similar to what was done for the tree. I also observed that it scales quite well. So, if we have 2 threads, the **first sort** could be done **in parallel** and then the other could be done as it is done now. If we have more we can parallelize the first and second iteration and so on: this could bring a good improvement I think.

That, more or less, concludes everything I wanted to say about this project. I would have gladly done and written more since I enjoyed doing the work, but the time I've spent on it is already a lot and I need to shift my attention and efforts to other topics now. **Thank you for reading**!

6 Appendix

In this appendix I will discuss how I was able to **travel through the scattered tree**. I'm actually not very happy with this implementation: I feel like there's a better way. In any case, let's see how it works.

First of all, to make this happen each node has to contain **more information**: in which **process** *is stored* its **left child** and in which process is stored its **right child**. So we have to add two integers to our data structure: `left_prc` and `right_prc`. Doing this we **increase the size** of each node and we also have to **correctly assign** this information during the construction of the parallel tree. This last step requires only to be careful during the initial distribution of the work because after that each process will always work on the construction of the same sub-tree. In any case, doing this is pretty easy: we'll focus on the actual function that allows the movement supposing to have each node properly constructed.

Second of all, there are **2 functions** *implemented* to move through the tree: one to go to the **left** and another to go to the **right**. These functions take **2 arguments**: the **pointer** *to a node* and the number of the **process** in which that node is stored. The functions will *modify the pointer* passed and make it point to the node to the right or to the left (depending on which function we are using) and will *return* the number of *the process* where that node is stored. Hence, using these functions and the information provided by them one can go from the root of the tree (of which a copy is always contained in the process 0) to the bottom, even if in the middle we pass through different processes and memories.

The idea behind the implementation is the following and exploits the common nodes between the processes[16]:

1. if we want to go to the left (right) then the processor which contains the node passed to the function *broadcast* to everyone which **process** contains the node to left (right) of the passed node;
2. *distinguish* **2 cases** based on which process contains the left (right) node:
   - the process is the one containing the node passed to the function: *update* the **pointer** passed to make it point to the left (right) of that node;
   - otherwise: *find* the **common node** shared with the processor containing the node passed to the function and *update* the **pointer** passed to make it point to the left (right) of that node;

```
// Function that allows to go to the left of any node in the tree
left(pointer, process) {
  MPI_Comm_rank(rank)

  // Where is the left node?
  if (rank == process) {     // If the current process contains the nodes
    next = pointer->left_prc // Recover the process that contains the node on the left
  }

  MPI_Bcast(next) // Broadcast to everyone the information of where the left node is

  // Recovering the left node
  if(rank == next) {                              // If the process has the left node
    if (rank == process) {                        // If the process has also the node
      pointer = pointer->left                     // Update the pointer
     } else {                                     // If it doesn't have also the node
      pointer = find_common(pointer, process, rank) // Find the common node
      pointer = pointer->left                     // Update the pointer
    }
  }

  return next // Everyone returns the information of where the left node is stored
}
```

[16]See 3 to understand what is meant by it.

References

[1] M. Overmars M. de Berg M. van Krevel and O. Schwarzkopf. *Computational Geometry*. Second, Revised Edition. Springer, 1998.

[2] Martin Skrodzki. “The k-d tree data structure and a proof for neighborhood computation in expected logarithmic time”. In: *CoRR* abs/1903.04936 (2019). arXiv: 1903.04936. URL: http://arxiv.org/abs/1903.04936.

[3] Wikipedia. *k-d tree*. 11 January 2022. URL: https://en.wikipedia.org/wiki/K-d_tree.

[4] Wikipedia. *Dutch national flag problem*. 16 May 2021. URL: https://en.wikipedia.org/wiki/Dutch_national_flag_problem.