DSSC
2021-22

Assignment 2

TRIESTE, 28/02/2022

GABRIELE CIMADOR – SM.3500523

Contents

Introduction

K-d tree

Citing Wikipedia, *"a k-d tree is a space-partitioning data structure for organizing points in a k-dimensional space."*[1]. They are a special case of binary space partitioning trees.

Let $P = \{p_0, p_1, \dots, p_n\} \subset R^k$ a finite point set. A k-d tree can be recursively defined as:

$$tree(P) = \begin{cases} \text{empty tree}, & \text{if } P = \emptyset \\ (P_{median},\ axis,\ tree(P_{left}),\ tree(P_{right})), & \text{otherwise} \end{cases}$$

Where:

- $axis$ is one of the k possible dimensions
- P_{median} is the median of P according to the direction $axis$.
- $tree(P_{left})$ is the left child of the node, and P_{left} is the partition of points for which the points are < P_{median} along the $axis$.
- $tree(P_{right})$ is the right child of the node, and P_{right} is the partition of points for which the points are > P_{median} along the $axis$.

With this definition, every non-leaf node can be thought of as implicitly generating a splitting hyperplane that divides the space into two parts. The direction for which compute the median can be decided with various criterions; for instance it can be chosen the direction of maximum extent of the set, or can be decided with a round robin approach. Also the median point can be found with various techniques with different complexity.

Aim of the assignment

The aim of this assignment was to develop a parallel program able to build a static kd-tree, with $k = 2$, given a set of points. To simplify things, the input parameter for the program will be the number of points of the 2-dimensional space; the points will then be randomly generated by the program itself.

The chosen programming language was C, the compiler used was GCC version 9.3.0; there have been developed two versions of the parallel program:

- MPI: the parallel part has been implemented using the MPI Standard ⇒ Distributed Memory Hypotesis;
- OpenMP: the parallel portion has been implemented using the OpenMP Standard ⇒ Shared Memory Hypotesis.

Algorithm

OpenMP

For the OpenMP version of the program, a recursive approach has been chosen. Since the data is balanced among all dimensions, the axis for the split were chose via round-robin.

Algorithm 1 function build_kdtree

```
$N \leftarrow \#\ of\ points$
$points \leftarrow \{P_0, P_1, ..., P_N\}$
$Ndim \leftarrow k$
**function** BUILD_KDTREE($points,\ N,\ Ndim,\ axis$)
    **if** $N$ is 0 **then**
        **return** Empty node
    **end if**
    $axis \leftarrow (axis + 1) \mod Ndim$
    $median \leftarrow$ median point in $points$ according to $axis$
    $child_{left} \leftarrow$ computed by another thread:
                $build\_kdtree(points$ before $median, N_{<median}, Ndim, axis)$
    $child_{right} \leftarrow$ computed by another thread:
                $build\_kdtree(points$ after $median, N_{>median}, Ndim, axis)$
    $node \leftarrow$ Node with$(median, axis, child_{left}, child_{right})$
    **return** node
**end function**
```

From *"Algorithm 1"* we can see that when the split is computed, the two partitions of the points are assigned to other two threads. While the two threads compute the child nodes, the calling thread is free to continue and thus becomes free to get work from other threads. This algorithm is perfectly suitable for the OMP Tasks.

MPI

```
function BUILD_KDTREE($points, N, Ndim, axis$)
    if $N$ is 0 then
        return Empty node
    end if
    $axis \leftarrow (axis + 1) \mod Ndim$
    $median \leftarrow$ median point in $points$ according to $axis$
    if there is an idle process then
        send $points$ after $median$ to idle process
        $child_{right} \leftarrow NULL$
    else
        $child_{right} \leftarrow build\_kdtree(points$ after $median, N_{>median}, Ndim, axis)$
    end if
    $child_{left} \leftarrow build\_kdtree(points$ before $median, N_{<median}, Ndim, axis)$
    $node \leftarrow$ Node with$(median, axis, child_{left}, child_{right})$
    return node
end function

function START_BUILD( )
    receive $points$ from requesting process
    tree $\leftarrow build\_kdtree(points, N, Ndim, axis)$
    return $tree$
end function
```

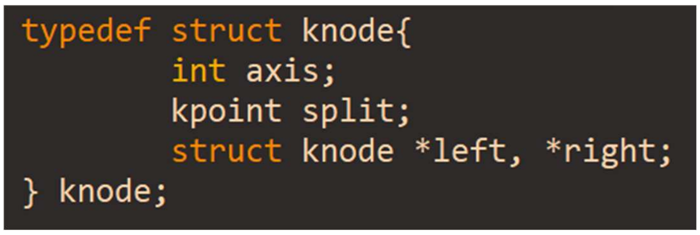
Also for the MPI version a recursive approach has been chosen. This time, only the right partition is sent to an idle process if available. If all the processes are busy, the computation becomes strictly serial.

Implementation

Common implementation choices

Node implementation

The node of a tree has been implemented with the following struct:



```
typedef struct knode{
        int axis;
        kpoint split;
        struct knode *left, *right;
} knode;
```

Where:

- Axis is the axis of the split.
- Kpoint is an array of floating point of size 2 (the two coordinates).
- Left and right are pointers to the child nodes of the current node.

Median selection

To select the median, the points are sorted using a quicksort algorithm; then, the point that lays in the middle is chosen as the median point.

OpenMP

The implementation of the algorithm is quite simple. Thanks to the simple syntax of OpenMP, it was just necessary to add a few OpenMP directives inside the serial version to make it parallel.

A parallel region is created and the master thread calls build_kdtree for first. The first node is generated and then the two children nodes will be computed by other threads by means of the OpenMP task. While the two threads are busy in computing the children, the thread that created the tasks is free, and so it can handle a new task that is created recursively. When no tasks are created, all the threads are located at the end of the parallel region and thus the tree is completed.

MPI

In the MPI implementation there is a strong assumption that comes handy: it is assumed to use a number of processes that is equal to a power of 2. Apart from that, the MPI version is similar to the OpenMP one. In this case, the left partition is kept by the original process; the right partition, instead, is sent to an idle process if available, otherwise it is kept by the original process and the computation becomes serial. Two big questions arise:

1. *How to identify an idle process?*
2. *How to start this recursive algorithm and how it is expected to behave?*

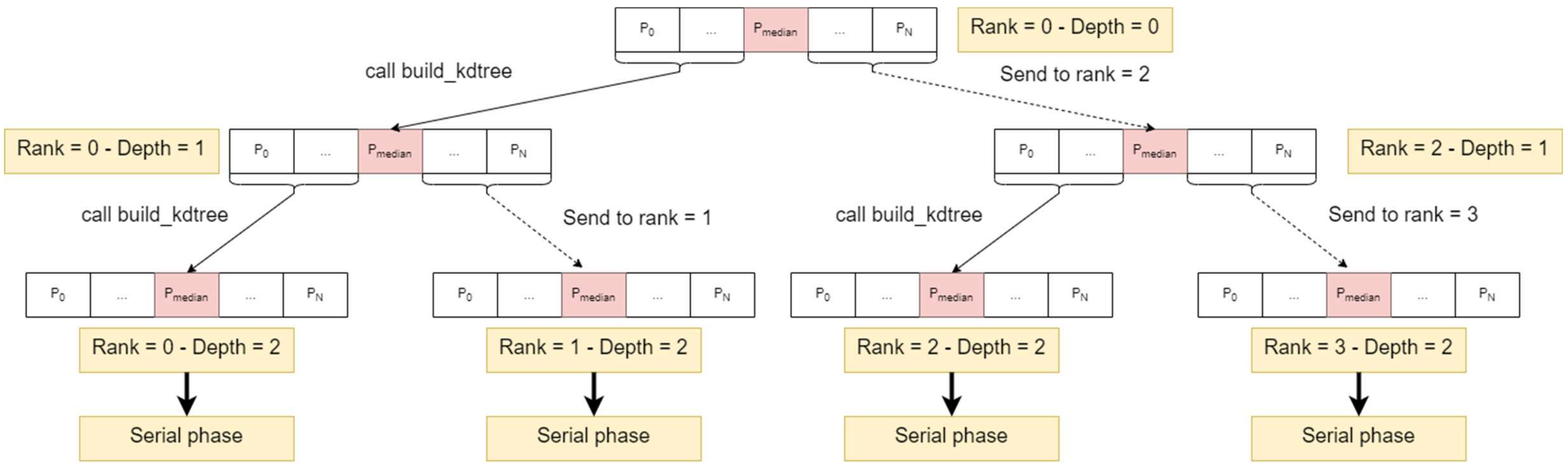


Figure 1: Example of execution of the MPI version with 4 processes.

How to identify an idle process?

The concept here is to use a strict pattern that can allow every rank to determine which process it has to send the partition to based only on its rank and on the current depth of the generating tree. To fulfil this purpose, the depth of the tree is tracked.

When an MPI process splits its portion of data, it generates a node. Then if $depth < log_2(size)$, it means that a further split in terms of working processes can be made. The process will then calculate the rank of the receiver process using the formula:

$$send\_rank = current\_rank + \frac{size}{2^{depth+1}}$$

This pattern ensures that the rank given by the formula is free and ready to work. Otherwise, if $depth \geq log_2(size)$, no further split can be made, and the computation becomes independent and serial for every process.

How to start the algorithm and how should it behave?

Suppose to have 4 processes. At first the master process ($rank = 0$), calls build_kdtree, while the others wait for data with an MPI_Recv call. The master sorts the points according to the first axis, selects the median point and creates the root node; at this point $depth = 0$.

The master sends the right partition to $send\_rank = 0 + \frac{size = 4}{2^{depth+1} = 2} = 2$, while it recursively calls build_kdtree with the left partition, which will wake other processes ($rank = 1$) due to the increase of depth. Meanwhile the $rank = 2$ process receives the data and calls build_kdtree, which will compute the node and then proceed as the master process.

When $depth = log_2(size)$, no further send will be made because all the processes will be busy; the computation becomes serial for every process.

Figure 1 shows a visual representation of the example described.

Performance model and scaling

In this section the attempt is to build a performance model of the *execution time* of the parallel program. Since the master process/thread is the first to work, the model takes in consideration its execution time, which will be the longest among all the others and thus the *execution time*. The model will be a function of the number of processes p and the size of the problem N: $T_{exec}(p, N)$.

MPI

The MPI version has two "phases":

1. **Distribution phase:**
   i. Sort the data
   ii. find the median
   iii. send half of the data to another process
   iv. recursively call build_kdtree with the other half.
2. **Independent phase:**
   i. Sort the data
   ii. find the median
   iii. recursively compute build_kdtree with first half.
   iv. Recursively compute build_kdtree with second half.

The model tries to describe the execution time considering this two phases.

Distribution Phase

In the distribution phase, the major of the workload is due to the MPI_Send call and the sorting. The timing of the MPI_Send is modelled with a linear model that takes in consideration the latency and the bandwidth of the network as parameters. The independent variable will be the number of points to be sent, and thus in the model the total amount of bytes is calculated. The model for the MPI_Send call is thus:

$$SEND(N) = \lambda + \frac{N\_dim * N * size\_type}{B}$$

Where:

- N is the number of points to be sent.
- λ is the latency of the network [s].
- N_dim is due to the fact that for every point, N_dim coordinates has to be sent.
- $size\_type$ is the total amount of bytes that a coordinate uses [b].
- B is the bandwidth of the network $\left[\frac{b}{s}\right]$.

Since a quick sort algorithm is being used, the average complexity of the sorting is $O(n \log n)$.

The distribution phase is executed until there are no idle processes; thanks to the strict pattern implemented, it is possible to know the number of times this phase will be executed: $log_2(p) - 1$.

The distribution phase can thus be modelled with the following series:

$$T_{distribution}(N,p) = \sum_{i=0}^{log_2(p)-1} \text{SEND}\left(\frac{N}{2^{i+1}}\right) + \gamma\left(\frac{N}{2^i} log\left(\frac{N}{2^i}\right)\right)$$

Where γ is a proportionality constant for the execution time of the sorting.

Independent Phase

In the independent phase there is no MPI_Send calls, because there is no possible parallelization. The algorithm becomes thus independent for every process, which executes a serial algorithm through a recursive approach. Again, thanks to the pattern used in the distribution of the data, it is possible to estimate the number of iteration of this phase. Since the algorithm stops when $N = 1 \vee N = 0$ and every time the data is split in half but the median, the independent phase is composed of $log_2(N) - log_2(p)$ iterations. The model for this phase is thus:

$$T_{independent}(N,p) = \sum_{i=log_2(p)}^{log_2(N)} 2^{i-log_2(p)+1} * \gamma\left(\frac{N}{2^i} log\left(\frac{N}{2^i}\right)\right)$$

Where:

- $2^{i-log_2(p)+1}$ is a multiplicative term which takes into account that the sorting operations are doubled at every further split of the set of points.
- γ is a proportionality constant for the execution of the sorting.

The overall *execution time* is then the sum of the two phases:

$$T_{exec}(N,p) = T_{distribution}(N,p) + T_{independent}(N,p)$$

Please recall that the complexity of the independent phase could also be obtained with the *Master's Theorem*, leading to a complexity of $T_{independent} = \Theta(n\, log^2(n))$ and thus $T_{independent}(N,p) = \omega\left(\frac{N}{p} log^2\left(\frac{N}{p}\right)\right)$ where ω is a proportionality constant. This approach lead to similar results.

Scaling

All the tests for the study of the performance of the program were run on the Orfeo GPU Nodes with the following specifications:

- CPU name: Intel(R) Xeon(R) Gold 6226 CPU @ 2.70GHz
- Sockets = 2
- Cores per socket = 12
- Threads per core = 2
- L1 cache size = 32 kB (core private)
- L2 cache size = 1 MB (core private)
- L3 cache size = 19 MB (core shared)

Evaluation of the model

To understand the correctness of the performance model, there have been compared the theoretical timings given by the model in respect to the measured timing, when the size of the problem was fixed to $N = 10000000$. The parameters of the SEND function where:

- $\lambda = 4.2 * 10^{-7}$ [s].
- $B = 12.5\ [MB/s]$.

Those values have been obtained via a PingPong benchmark on the same nodes where the experiment took place.

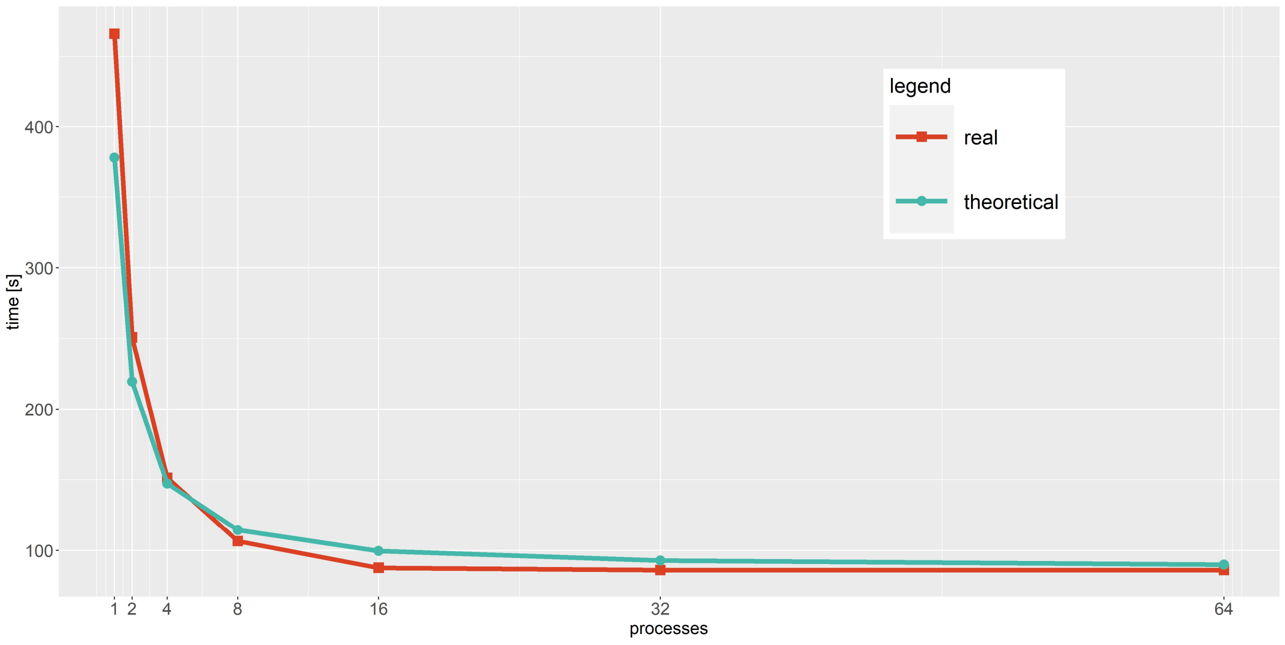


Figure 2: model vs measured

From the graph, it is possible to deduce that the program scales as the theoretical model predicted. The overall complexity described by the model seems to be correct.

Strong scalability

To study the strong scalability of the code, it has been used:

- $N = 100000000$
- $p \in \{1, 2, 4, 8, 16, 32, 64\}$

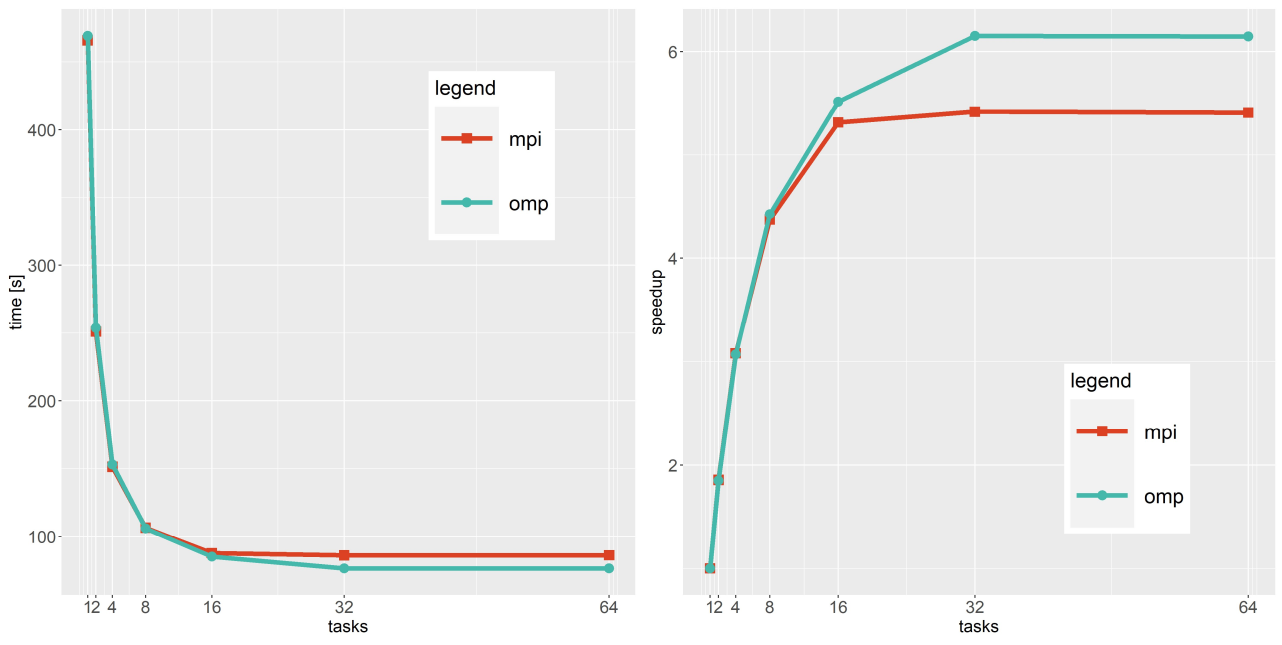


Figure 3: timing and speedup of the strong scaling

Both versions have similar performances. The program seems to scale decently until 16 processes/task; for higher numbers of tasks, the program suffers from the overhead of the parallelization. The speedup is slightly better in the OpenMP version, and confirms the evidence from the first graph.

Weak scalability

For studying the weak scalability, there have been used the following parameters:

- $p \in \{1, 2, 4, 8, 16, 32, 64\}$
- $N = 10000000 * p$

With this configuration, it was possible to linearly increase the dimension of the problem with the increase of the number of tasks.

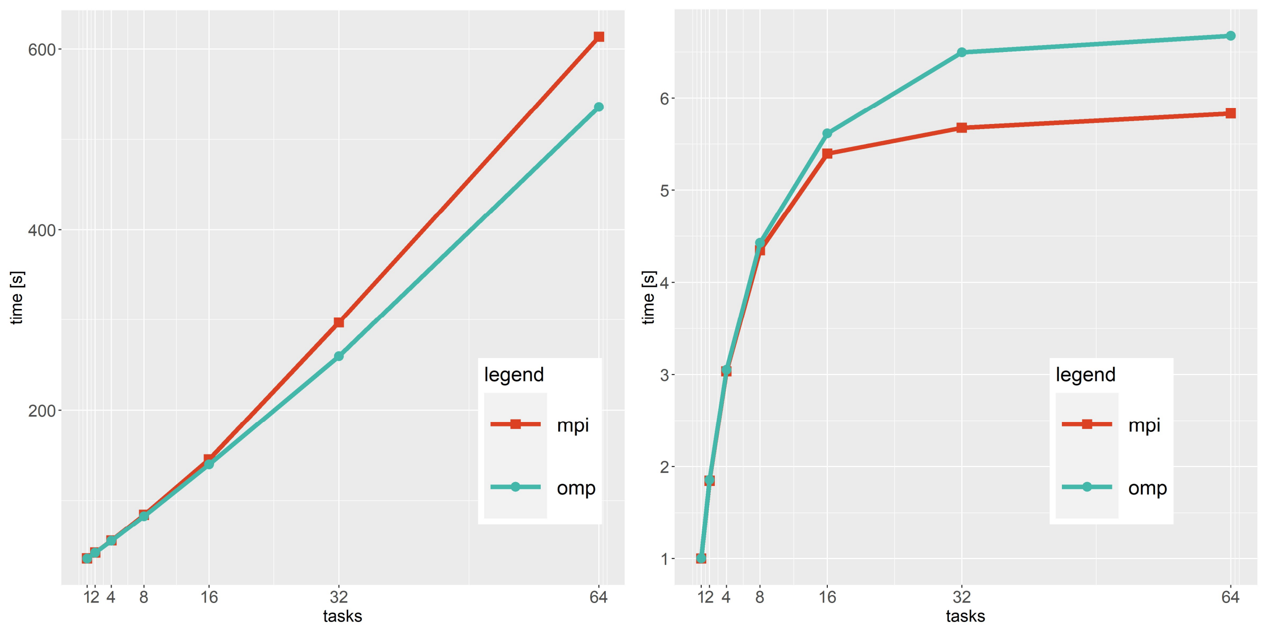


Figure 4: timing and speedup for the weak scaling

Both versions of the program are not weakly scalable. The linear increase of work and processors does not lead to a constant time. The reason probably is the load imbalance of the implementation: the workload is balanced from a certain depth only. In the generation of the first nodes, very few tasks are sorting big amounts of data. For instance, the master process/thread must sort the entire dataset in the very first node generation; this means that for increasing workload, the master has an increase in the computational effort. Hence, the program will hardly scale weakly, as the graphs indicate.

Discussion

The program described and analysed in this assignment could be improved in several aspects.

Selecting the median

In the implementation of both algorithms, the median point is selected via a complete sorting of the dataset. Although the sorting is sufficient to find the median, it is not necessary. A better way (in terms of complexity) would be to partition the dataset in two unordered branches, one with $p < median$ and the other with $p > median$. With the hypothesis of an uniform distribution along all the directions, this operation could be performed with a complexity of $\Theta(n)$, which is better than the quicksort. This implementation would increase the scalability of the program, especially the weak one, where the sorting operations seems to be a huge problem.

OpenMP - False Sharing

In the OpenMP version there is a problem of False Sharing. If we suppose that the coordinates are float type of size 4 bytes and the cache lines are 64 bytes long, it means that a cache line can store $\frac{64\ b}{4\ b*2} = 8$ 2-dimensional points. It means that when two threads are sorting two partitions of points which are smaller than 8 points or that are partially stored in the same cache line, there will be an increase in the workload due to the cache coherency protocol. For increasing the performance of OpenMP version, this issue might be resolved. For example with a copy of the partition to be sorted in another memory location, which will be exclusive for the thread. Even though this might seem a major issue, the belief is that sorting operations that are less than 8 points big is not that problematic.

MPI – Number of processes

A major issue with the MPI version is that the program is obliged to work with a power of 2 number of processes. This might be quite limiting, also because scalability becomes a problem when there is the necessity to work with an high number of processes. A possible solution to this issue might be to do the splitting of the workload among the processes as in the original version, and then to do a further splitting in the first processes, using the remaining $size \bmod 2^{\hat{p}}$, $\hat{p} = \lfloor log_2(size) \rfloor$ processes.

Code optimizations

Another optimization that could be done is to simply write better code, which takes in consideration all the hardwired optimizations that should be taken in consideration when writing a code.