ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

Scuola di Ingegneria e Architettura

Dipartimento di Informatica - Scienza e Ingegneria (DISI)

Corso di Laurea Magistrale in INGEGNERIA INFORMATICA

Design, Implementation and Evaluation of Parallel Solutions for a Nested Explainability Algorithm

Tesi di laurea in: SISTEMI OPERATIVI M

Relatore:
Prof.ssa Daniela Loreti

Candidato:
Gabriel Cortesi

Corelatore:
Dr. Giorgio Visani

Anno Accademico: 2021-2022
Sessione III

ALMA MATER STUDIORUM · UNIVERSITY OF BOLOGNA

School of Engineering and Architecture

Department of Computer Science and Engineering - DISI

Master's Degree in COMPUTER ENGINEERING

Design, Implementation and Evaluation of Parallel Solutions for a Nested Explainability Algorithm

Dissertation in: OPERATING SYSTEMS M

Supervising Professor:
Prof. Daniela Loreti

Presented by:
Gabriel Cortesi

Co-Supervisor:
Dr. Giorgio Visani

Academic year: 2021-2022
Session III

Abstract

In the field of Machine Learning and Data Science there is an escalating need for performance as workloads become more and more complex. Parallelization over multiple cores and machines (clusters) is often employed as a means to significantly improve performance.

This work specifically considers the explainability algorithm GLEAMS (Global & Local ExplainAbility of black-box Models through Space partitioning) and the poor performance offered by its sequential Python implementation. GLEAMS is a post-hoc, model agnostic explainability technique capable of giving a global understanding of the original model through recursive partitioning of the input space into non overlapping cells, each featuring a local linear approximation of the black-box model.

The purpose of this work is the analysis, development, implementation and testing of a parallel distributed solution for the sequential GLEAMS explainability algorithm. The algorithm poses certain interesting parallelization challenges such as a recursive binary tree and nested parallelism. Notably, the nested nature of the parallelism is of marked relevance due to the complexities it introduces and the poor support that existing Python frameworks and solutions offer for it.

Multiple solutions were designed and implemented, and this paper describes the steps taken for their development, justifies the choices made, explains their workings, illustrates their differences and extensively analyses the performance offered. In particular, this work proposes an `asyncio` based approach, in combination with the Ray framework, as a practical solution to many of the limitations encountered with the current state of nested parallelism support in Python. Additionally, some theoretical and more general approaches and solutions inspired by other languages are proposed and discussed.

Table of Contents

Introduction

In the field of Machine Learning and Data Science there is a strong and increasing need for performance as more applications and algorithms are developed and more data is being processed. To obtain the required performance, parallelization is often employed, but in the context of workloads that have increasingly larger sizes, there is a particular interest for parallelization over distributed environments, such as High Performance Computing (HPC) clusters or Cloud data centres. For this reason there is special interest in implementing solutions that could not only offer higher performance on local multi-core machines, but also support execution over distributed cluster environments.

In particular, with the increasing growth of machine learning applications, explainability has become a progressively more important domain, as laws, ethics and security concerns increase the required explanations and reasoning for decisions and results given by a machine learning model. GLEAMS (Global & Local ExplainAbility of black-box Models through Space partitioning) is a novel approach in this field that aims to offer explainability for black-box machine learning models such as neural networks [1].

GLEAMS is a post-hoc method and as such it is applied to an already trained model to generate a surrogate one that can be used to obtain explainability information. It aims to provide a global understanding of the original model through recursive partitioning of the input space into non overlapping cells, each featuring a local linear approximation of the black-box model. GLEAMS is inspired by the MOB (MOdel-Based recursive partitioning) [4] algorithm for the partitioning of the domain which allows it to maintain high accuracy and precision, while still giving a global understanding of the input model.

A sequential Python implementation of the GLEAMS algorithm was developed at the University of Bologna [1] to test and apply the approach in practice, but the performance offered by said implementation is limited by its sequential nature. The long execution time is very detrimental to testing and to the development of improvements, as well as severely hindering the usability and effectiveness of the algorithm as a whole. In particular the size of pragmatically solvable problems is limited by their computation complexity and consequently their impractical execution times, rather than their actual size in memory.

The purpose of this work is the analysis, development, implementation and testing of a parallel distributed solution for the sequential machine learning explainability algorithm, GLEAMS. The algorithm poses certain interesting parallelization challenges such as recursive binary trees and nested parallelism. In particular, the nested nature of the parallelism is of marked relevance due to the complexities it introduces and the generally poor support that existing frameworks and solutions offer for it.

Multiple solutions were developed for the purpose of the project and this paper will describe the steps taken for their development, justify the choices made, explain their workings,

illustrate their differences, analyse the performance offered and discuss the limitations encountered.

The requirements for the parallel solution to be implemented in this work are to offer high performance in both local and distributed environments, be simple to both implement and maintain, and require the minimum amount of changes to the existing sequential code as possible to not disrupt ongoing development. The main objective of the distributed parallel version is that of gaining the highest performance possible, and not that of supporting extremely large problem sizes. There are approaches that are geared towards handling larger than RAM datasets by leveraging multiple machines, but in this work the focus will primarily be on performance.

This text will not give an in depth description of GLEAMS as an explainability approach, and it will instead refer to other more complete papers [1] where the technique is described in full detail. A more concise and practically oriented explanation will be provided with the main objective of giving enough information to understand the structure of the algorithm and the general operating principles, as the focus of this work will be the parallelization techniques used for the algorithm and not the algorithm itself.

The thesis is structured in 4 chapters:

1. **Analysis of the sequential algorithm:** This chapter gives an overview of the GLEAMS explainability method and describes in more detail how the MOB-based recursive partitioning works. The structure of the code is then analysed to identify and evaluate parallelization opportunities.
2. **Selection of the technologies and frameworks:** here the choice of the language is discussed, followed by an analysis and comparison of various available frameworks, complete with testing pertaining their suitability for the current use case.
3. **Implementation of the parallel solutions:** this chapter presents all the steps taken for the development of the parallel solutions. It covers profiling, regression testing and performance measuring. It describes a series of optimizations that were used to obtain significant performance improvements and then discusses in detail the implementation of four different parallel versions, including how they function, the differences they bear and the reasons they were implemented.
4. **Performance tests and analysis of results:** it is the final chapter in which we report the performance results of the implemented solutions tested on both a single multi-core machine and on a laboratory with 98 machines in a cluster. The results are analysed using different metrics and through the use of graphs, with the purpose of comparing the different versions and determining the best one. The obtained results are then further discussed with a particular focus on some of the limitations encountered. Considerations are made regarding alternative approaches that could overcome them.

Chapter 1

Analysis of the sequential algorithm

This chapter will give an overview of what GLEAMS and MOB are, explore the structure of the algorithm and go into details about the specific characteristics of this parallelization problem.

1.1 GLEAMS

GLEAMS, also referred to as Global-Lime when first presented in a dissertation by V Stanzione [1], is a newly developed model agnostic, post-hoc, global technique that aims to offer explainability for generic black-box machine learning (ML) models. The idea is to generate an assortment of spatially separated interpretable submodels (linear regressions), that can approximate the behaviour of the unknown ML function. It can be applied for both regression and classification scenarios.

Defined $M(X)$ as the function of the black-box ML model, with $X \in R^d$ d-dimensional space, the equation $Y = M(X) \in R$, expresses the behaviour of the ML function, with Y indicating either the continuous variable predicted by the model, in the case of regression, or the probability a sample X being predicted as class c out of the set of possible C outcomes, namely $Y = P(C = c \mid X)$ in the case classification [1]. Each dimension of the R^d input domain indicates one of the d input variables, or features of the ML model.

In very simple terms, what a linear approximation of $M(X)$ permits, with X being a d -dimensional point of the input domain, is to obtain coefficients that are able to give a measure of weight to the d input variables. These weights can help to understand which of the model features influenced the particular result the most, giving the desidered explainability information.

Depending on the shape of the ML model, a particular linear regression could be valid for a bigger or smaller section of the input domain. Usually, a linear regression will only accurately approximate the behaviour of the ML function in a neighbourhood of a particular input. This is the basic concept used by some popular local explainability methods such as LIME (Local Interpretable Model-Agnostic Explanations) [2]. The Figure 1.1.1 shows a 3D example of a GLEAMS model emulating an ML function through many different linear approximations.

What GLEAMS tries to do, is to recursively split the input domain into non overlapping subdomain cells, until the linear regression for each cell satisfies a certain fitness criterion. What it ends up with is a binary tree where each node represents a subdomain. Leaf nodes contain the linear regression models that approximate the original ML function in their subdomains. Each node is obtained through a binary partition of the parent domain, through

the use of a hyperplane. The hyperplane splits the parent domain along one of the d domain dimensions, in a way that tries to maximise the quality of the linear approximation in the two subdomains.

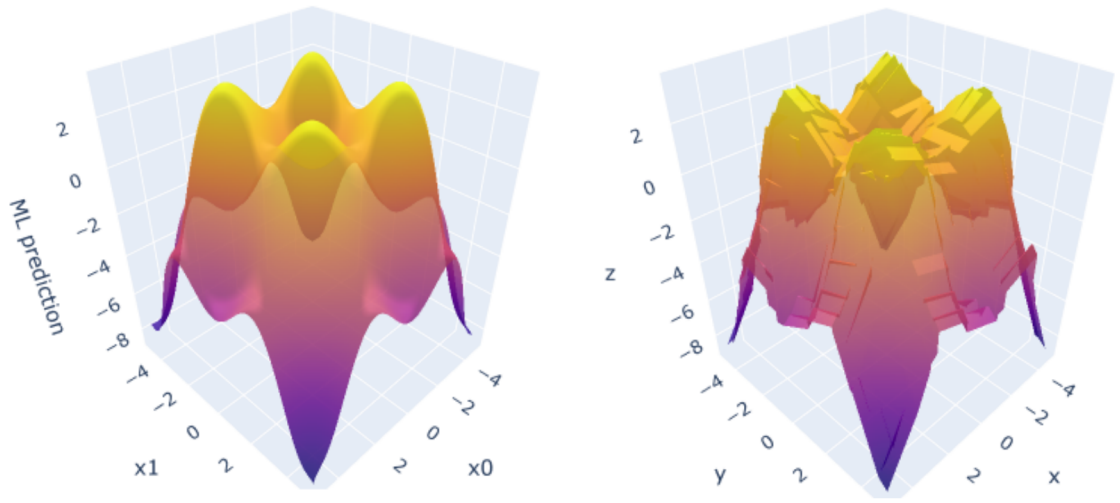
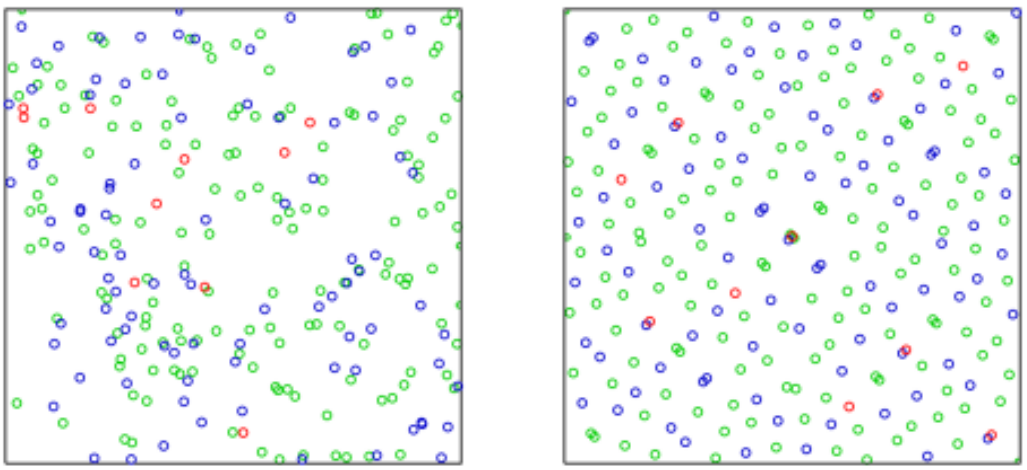


Figure 1.1.1: A graphical representation of a GLEAMS linear approximation model on the right with the original model on the left as comparison. In this example the input domain is two dimensional and the prediction value is represented on the third axis [1].

For its operations GLEAMS requires sample points from all over the input domain, which it then uses to generate the binary tree and give global understanding of the original ML model. To obtain an even sampling covering the space, GLEAMS proposes the use of Sobol sequences [3] which should offer better results than normal uniform sampling. In Figure 1.1.2 a two dimensional sampling example is shown, comparing the spatial distribution of a normal pseudorandom sequence and a Sobol sequence. In the Sobol case the points are more regularly spread without giving up randomness and variation, but more importantly, they don't haphazardly leave vaster regions without samples.



(a) 2D Pseudorandom sequence.

(b) 2D Sobol sequence.

Figure 1.1.2: Two dimensional Sampling Examples [1].

In order to achieve good results with Sobol sequences though, the number of samples is required to be a power of two, $num\_samples = 2^m$. More information on Sobol sequences and their use in GLEAMS can be found in chapter 2.1 of Stanzione's dissertation [1]. The total number of samples is thus controlled by the m exponential input variable.

GLEAMS therefore generates an array X of 2^m points with d dimensions and uses the original black-box model to obtain an array of corresponding predictions y. With the X and y arrays, GLEAMS is able to generate the tree of linear regression submodels that offer the desidered global explainability. Figure 1.1.3 shows a graphical representation of the relationship between GLEAMS and the input sample arrays.

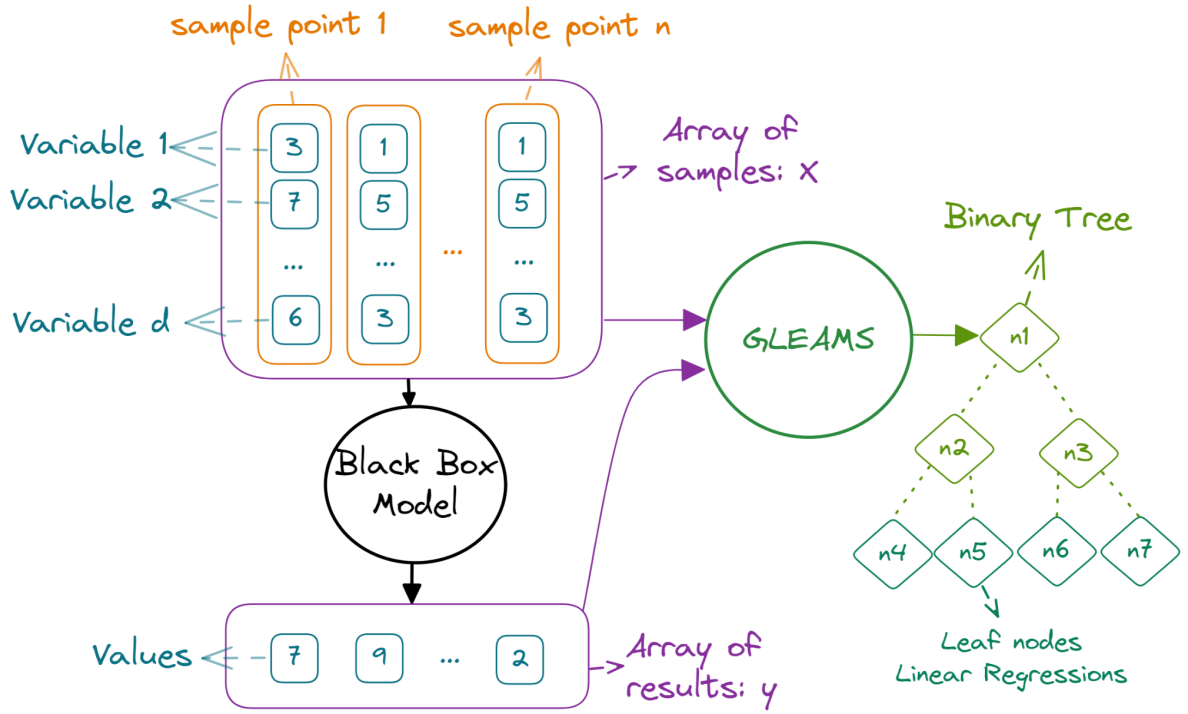


Figure 1.1.3: A graphical representation of the X and y arrays, as well their relationships with the Black-Box model and GLEAMS.

1.2 MOB

MOB is a generic algorithm for model-based recursive partitioning (Zeileis, Hothorn, and Hornik 2008) [4]. Instead of fitting a global model to the data set, it estimates a number of local models, on different subsets of the data "learned" by recursive partitioning. It works by first fitting a parametric model to the dataset followed by testing for parameter instability over a set of partitioning variables. If there is some overall parameter instability, the model is split with respect to the variable associated with the highest instability. The procedure is

repeated for each subset generated by the split recursively until all the partitions of the model are stable to a satisfactory degree.

GLEAMS internally uses an implementation of the MOB approach to handle the recursive partitioning of the sample space into a binary tree structure where leaf nodes contain the linear regression explainability submodels.

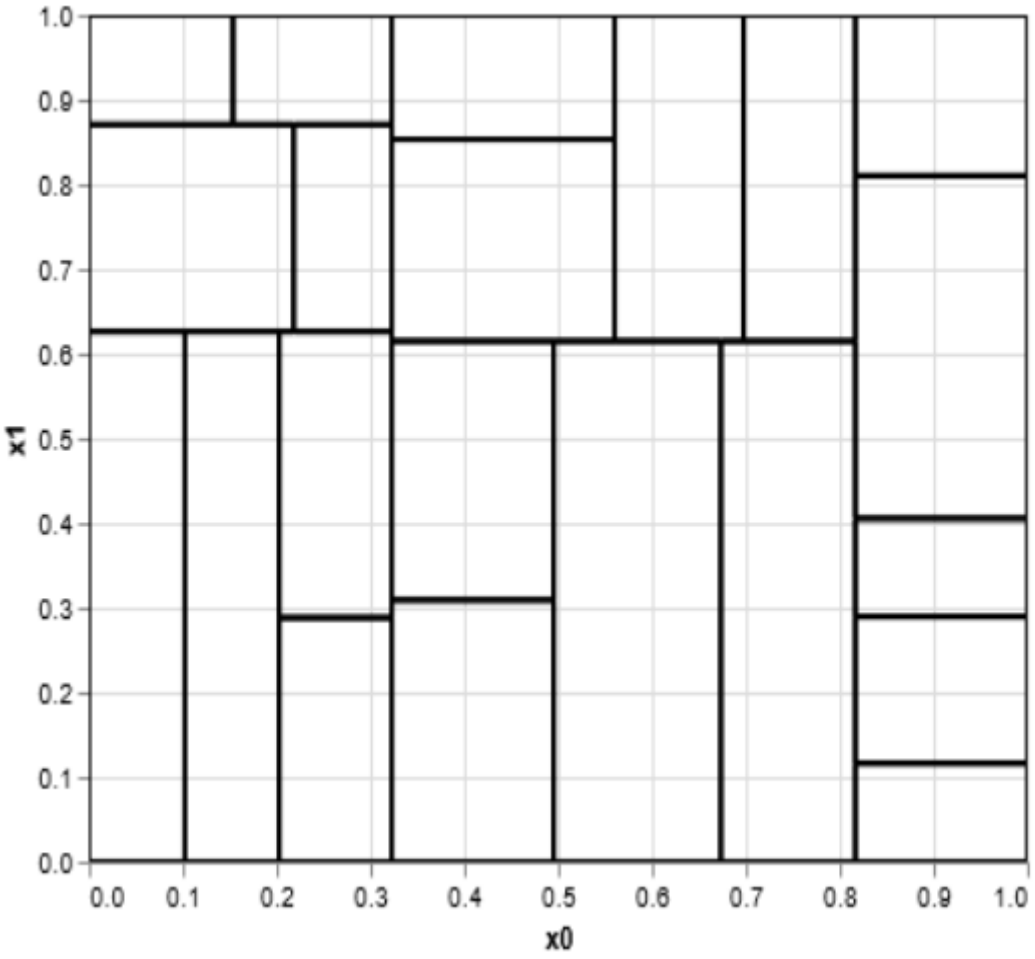


Figure 1.2.1: Grid structure of GLEAMS domains. The initial domains for the variables are x0 ∈ [0, 1) and x1 ∈ [0, 1) and the tree maximum depth was fixed to 5. In each rectangle, a plane with random coefficients is inserted. Note that this figure represents the partitioning of the domains of x0, x1; y is not depicted [1].

Considering a 2-dimensional case for the input domain, the Figure 1.2.1 shows the result of a MOB recursive partitioning. For each partition line (hyperplane in multi-dimensional spaces), the algorithm determined along which dimension to place it and at what value. To make this decision MOB computes the best split value along each dimension, then uses a scoring function to compare the quality of the obtained partitions from all dimensions, selecting the best one.

The algorithm stops once each partition satisfies a certain fitness criterion. Intuitively, the finer grained the partitions, the more accurate the result, but a too fine grained approach would result in the complete fragmentation of the domain and a very slow performance model. The granularity of the partitions also depends on the number of available sample points. If the number is too low the granularity will be too coarse, leading to less accurate results, while higher numbers of samples allow for finer grained partitions, at the cost of more calculations required to process them.

The computation of the best split-point along a particular dimension can be done using various methods/criterions. GLEAMS proposes its own method [1], which essentially involves the iterative calculation of a score for each possible split value, using the result of the previous iteration as a starting point, to reduce the overall processing required for the computation of the score for each value. The highest scoring value is selected as the split-point and is later compared to the best split-point computed along other dimensions.

Each partition generates two subsets of samples, each represented by a node in a binary tree. Each node in the tree has an associated subsection of the domain, which can be further split in the case of nodes with children. Leaf nodes are those where MOB stopped the recursion and where a linear regression submodel is stored.

1.3 Structure of the sequential implementation

GLEAMS's algorithm was developed as a Python script split into many modules and components. The main functionality of the algorithm is exposed by the `gleams` module, through a `Gleams` class and series of methods. The class allows its users to fit the explanation model around the black box one, obtain predicted values using the faster linear surrogate model, compute the local importance for a specific sample as coefficients of the input variables, compute the global importance as average coefficients of input variables and locally plot what-if changes.

The actual core of the script is in the fitting of the surrogate model. This is performed in the `pymob` module where the `MOB` class provides an *`sklearn`* like estimator [5], whose behaviour is to recursively partition the input space and fit a linear model inside each partition.

Pymob's code is itself divided into many files, the most important ones being `mob.py`, `mob_utils.py`, `split_criterion.py` and `classes.py`. The mob class is defined in `mob.py` and there is also the implementation of the `MOB.fit()` method. `MOB.fit()` takes in input the sample array `X` with its prediction `y`, and performs the MOB algorithm to recursively generate the binary tree structure. The tree is stored inside the `MOB` class object with a flag indicating that the model was fitted.

To perform the algorithm, the utility method `MOB.mob_fit_recursive()` is used. This method takes subsets of `X` and `y` as inputs and returns a tree node. If for the current sample

subset the computed linear regression is accurate enough, the node is marked as leaf and returned directly. If on the other hand the regression isn't accurate, `mob_fit_recursive()` will discard the regression and attempt to split the samples into two more subdomains. To do this it calls the `compute_partition()` utility function defined in the `mob_utils.py` file. This function iterates over all the dimensions of the domain and for each of them computes the best split-point.

For each dimension, `compute_partition()` calls `get_best_splitpoint()` which essentially sorts the `X` and `y` subsets along the currently considered dimension and then proceeds with computing the best splitting point and a quality score for it, following a specific criterion. The aim is to identify an index at which to split the sorted array in two. Geometrically, this is equivalent to dividing a multidimensional space with a hyperplane. The implementation supports the use of many different criterions for the evaluation of the best split point and its score, and they are all defined in `split_criterions.py`. For this project, support was maintained for all of them, but for performance and optimization purposes, only one was considered, the new method proposed by GLEAMS, labelled as `MSE_fluc_process` in the code.

The way the method works is by sequentially iterating over all possible values in the `X` array and computing for each of them a score indicating the quality of that particular split value. Each iteration uses the results of the previous one to reduce the number of computations necessary. The mathematical operations used to compute the quality score and their explanation are quite complicated and beyond the interests of this work, but are available for consultation in Stanzione's dissertation [1]. Of interest is the fact that, even considering the optimization of reusing partial results, all the maths operations end up being quite computationally intensive. In fact, anticipating the results of the profiling outlined in Chapter 3.1, they take up the majority of the execution time of the entire script.

After all the scores for each value in the sorted array are computed, the highest one is selected as the best one and returned by `get_best_splitpoint()`.
Since `get_best_splitpoint()` was invoked once for each different dimension of the input domain, another selection is performed to pick the highest scoring split-point along all dimensions. The variable and the splitting value are then returned to the calling `compute_partition()` function which proceeds with splitting the input `X` and `y` subset arrays using the identified split variable and point, generating two subdomains. For each new subset of X and y, `mob_fit_recursive()` is invoked, to compute and generate a left and right child node. Once the entire subtree is populated (leaves are found for each sub-branch), the function finally returns to the higher level caller. In this implementation the binary tree is constructed following left-most, depth first traversal.

The Figure 1.3.1 shows a flowchart approximating the way the code is structured.

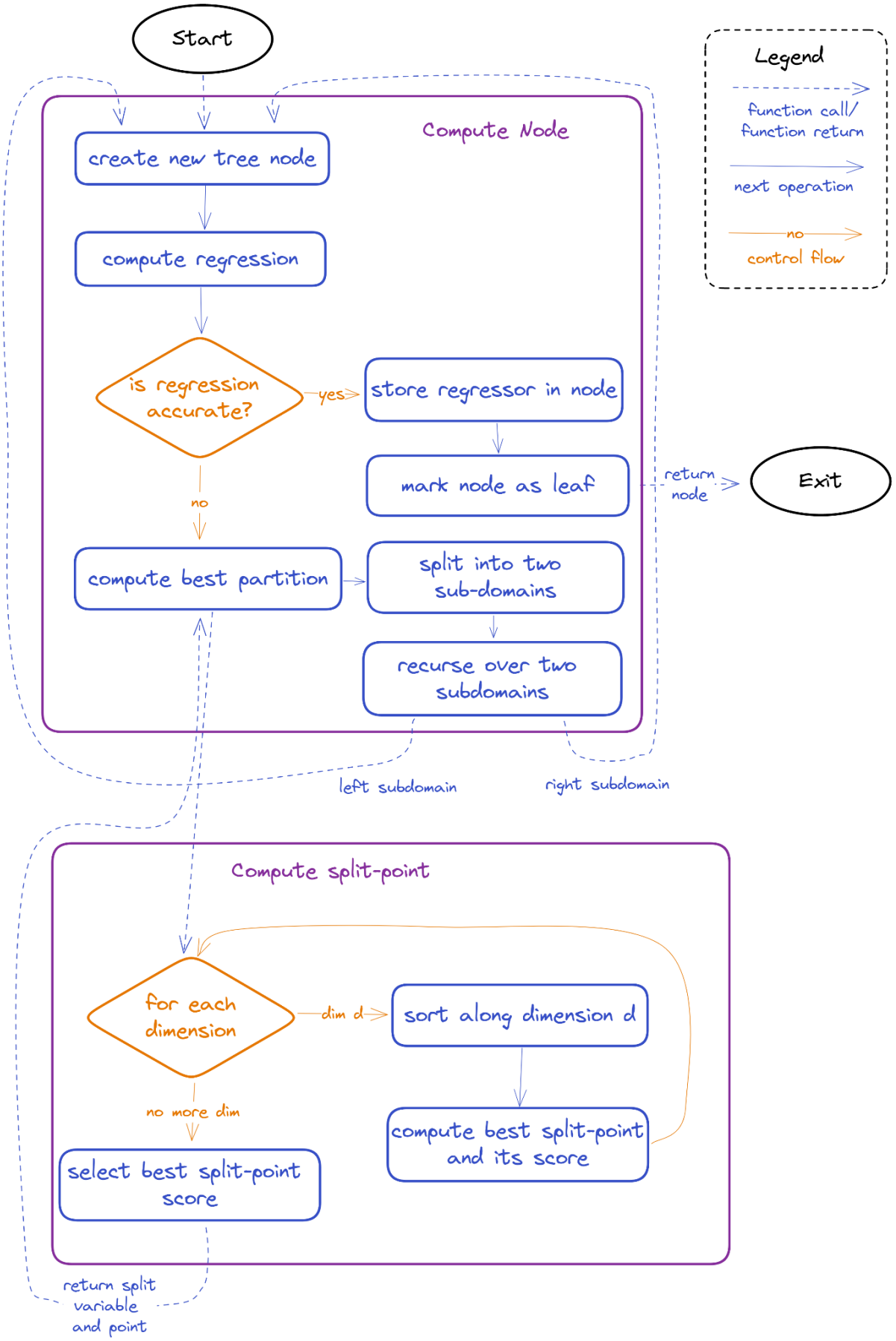


Figure 1.3.1: Flowchart overview of the MOB algorithm inside GLEAMS.

The following is a very simplified pseudocode version of the sequential algorithm. Changes were made to make the code easier to understand and many things were greatly simplified or even completely omitted to only highlight the relevant structure of the code and computations. The original code is obviously much more complex.

```
import numpy as numpy

def mob_fit_recursive(X_subset, y_subset):

    regression = compute_regression(X_subset, y_subset)

    if pre_stopping_condition(regression, X_subset, y_subset):
        return TreeNode(regression, X_subset, y_subset, is_leaf=True)

    split_dimension, split_value = compute_partition(X_subset, y_subset)

    if post_stopping_condition(split_dimension, split_value, X_subset,
y_subset):
        return TreeNode(regression, X_subset, y_subset, is_leaf=True)

    left_X_subset, right_X_subset, left_y_subset, right_y_subset = split_data(
        X_subset, y_subset, split_dimension, split_value)

    node = TreeNode(X_subset, y_subset, split_dimension, split_value,
is_leaf=False)

    node.left_child = mob_fit_recursive(left_X_subset, left_y_subset)
    node.right_child = mob_fit_recursive(right_X_subset, right_y_subset)

    return node

def compute_partition(X_subset, y_subset):
    dimensions = X_subset.shape[1]
    dimension_scores = numpy.zeros(dimensions)
    dimension_splitpoints = numpy.zeros(dimensions)

    for dim in range(0, dimensions - 1):
        # compute split value for each dimension
        # return the dimension with the best split value
        score, splitpoint = get_best_splitpoint(X_subset, y_subset, dim)
        dimension_scores[dim] = score
        dimension_splitpoints[dim] = splitpoint
```

```
    best_dimension = numpy.argmax(dimension_scores)

    return best_dimension, dimension_splitpoints[best_dimension]

def get_best_splitpoint(X_subset, y_subset, dim):
    # compute the best splitpoint for a given dimension
    # return the score and the splitpoint

    sorted_indices = numpy.argsort(X_subset[:, dim])
    sorted_X = X_subset[sorted_indices]
    sorted_y = y_subset[sorted_indices]

    # exclude indices where the value is the same as the next value
    # or the groups are too small
    split_indices = extract_valid_indices(sorted_X[:, dim])

    split_scores = compute_mse_fluc_process_criterion(sorted_X, sorted_y, dim,
split_indices)

    best_score_id = numpy.argmax(split_scores)
    best_score = split_scores[best_score_id]
    best_splitpoint = sorted_X[best_score_id, dim]

    return best_score, best_splitpoint

def compute_mse_fluc_process_criterion(sorted_X, sorted_y, dim,
split_indices):
    num_samples, num_variables = sorted_X.shape
    split_scores = numpy.zeros(num_samples)

    previous_index = 0
    # stores partial result of previous iteration to simplify computation
    previous_partial_result = numpy.zeros(num_variables, num_variables)

    for split_index in split_indices:
        split_scores[split_index], previous_partial_result =
mse_fluc_process_score(
            sorted_X, sorted_y, dim, split_index, previous_partial_result,
previous_index)
        previous_index = split_index
```

```
    return split_scores

def mse_fluc_process_score(sorted_X, sorted_y, dim, split_index,
previous_partial_result, previous_index):
    # compute the score for a given splitpoint
    # return the score and the partial result for the next iteration
    # to simplify computation

    newlines = sorted_X[previous_index:split_index]
    # compute new partial result using previous partial result
    # ... heavy maths computations ...
    # compute score using new partial result
    # ... heavy maths computations ...

    return score, partial_result
```

1.4 Analysis of parallelization opportunities

As mentioned before, the purpose of the project is to modify the existing sequential version to enable parallelism and not to develop a parallel solution ex novo. The less impactful the changes are to the code structure the better, to simplify the transition for other developers working on the mathematical implementation of the GLEAMS algorithm.

The sequential code was thus analysed to identify opportunities for parallelization. The aim is to find situations where there are independent operations that can be performed simultaneously.

```
def mob_fit_recursive(X_subset, y_subset):

    regression = compute_regression(X_subset, y_subset)

    if pre_stopping_condition(regression, X_subset, y_subset):
        return TreeNode(regression, X_subset, y_subset, is_leaf=True)

    split_dimension, split_value = compute_partition(X_subset, y_subset)

    if post_stopping_condition(split_dimension, split_value, X_subset,
y_subset):
        return TreeNode(regression, X_subset, y_subset, is_leaf=True)

    left_X_subset, right_X_subset, left_y_subset, right_y_subset = split_data(
```

```
        X_subset, y_subset, split_dimension, split_value)

    node = TreeNode(X_subset, y_subset, split_dimension, split_value,
is_leaf=False)

    node.left_child = mob_fit_recursive(left_X_subset, left_y_subset)
    node.right_child = mob_fit_recursive(right_X_subset, right_y_subset)

    return node
```

The pseudocode above shows the rough implementation of the `mob_fit_recursive()` function, which is basically the entry point of the MOB algorithm. From the pseudocode it is possible to see two exit conditions from the recursion, pre and post the computation of the split-point. In case of exit, a leaf tree node is returned, otherwise two child nodes are computed recursively after the input data is partitioned.

An important observation is the fact that the computation of the two child nodes is independent from one another. The two subdomains are separate by design and there are no interactions or shared data between the two child nodes. This means that the computation could be performed in parallel without any data race or synchronisation issue.

```
def compute_partition(X_subset, y_subset):
    dimensions = X_subset.shape[1]
    dimension_scores = numpy.zeros(dimensions)
    dimension_splitpoints = numpy.zeros(dimensions)

    for dim in range(0, dimensions - 1):
        # compute split value for each dimension
        # return the dimension with the best split value
        score, splitpoint = get_best_splitpoint(X_subset, y_subset, dim)
        dimension_scores[dim] = score
        dimension_splitpoints[dim] = splitpoint

    best_dimension = numpy.argmax(dimension_scores)

    return best_dimension, dimension_splitpoints[best_dimension]
```

Looking at the deeper levels of the computation through more simplified code, it can be observed how the algorithm uses a for loop to compute the best split-point along each dimension of the input domain, then picks the highest scoring one as the final result. The computation of the best split-point along each particular dimension is independent from the

rest and could be executed in parallel. Here as well there is no shared write data or communication between the different invocations of `get_best_splitpoint()`. It is important to note that this parallelization opportunity is deeper in the call stack than the previous one, and not at the same level, making this a situation of nested parallelism.

With "nested parallelism", a situation is indicated where a parallel task could itself generate further parallel tasks. Two nested for loops being parallelized could be a simple example of such a situation. In this case though the outer level isn't a for loop but the recursive tree structure.

```
def get_best_splitpoint(X_subset, y_subset, dim):
    # compute the best splitpoint for a given dimension
    # return the score and the splitpoint

    # sort sample arrays following ascending dim variable values
    sorted_indices = numpy.argsort(X_subset[:, dim])
    sorted_X = X_subset[sorted_indices]
    sorted_y = y_subset[sorted_indices]

    # exclude indices where the value is the same as the next value
    # or the groups are too small
    split_indices = extract_valid_indices(sorted_X[:, dim])

    split_scores = compute_mse_fluc_process_criterion(sorted_X, sorted_y, dim,
split_indices)

    # identify splitting index with the best score
    best_score_id = numpy.argmax(split_scores)
    best_score = split_scores[best_score_id]
    best_splitpoint = sorted_X[best_score_id, dim]

    return best_score, best_splitpoint
```

Looking at the top level of the `get_best_splitpoint()` function implementation, there aren't really any operations that could be performed concurrently. To reiterate, the goal of the function is to obtain the best split-point along a specific dimension, to do so it first sorts the input arrays along said dimension. In particular, the sample points are ordered following ascending values of the variable corresponding to the considered dimension. Then it uses a criterion to obtain a collection of scores for each possible partition value. The highest score is used to identify the index of the best split-point.

Pretty much every operation requires the result of the previous one. The only exception might be computation of `sorted_X` and `sorted_y`, but they are obtained by indexing the array with already sorted indexes, and not through two separate sorting steps. The potentially saved time of performing the indexing operations in parallel is unlikely to be worth the parallelization overhead, regardless of the parallelization solution used.

```
def compute_mse_fluc_process_criterion(sorted_X, sorted_y, dim,
split_indices):
    num_samples, num_variables = sorted_X.shape
    split_scores = numpy.zeros(num_samples)

    previous_index = 0
    # stores partial result of previous iteration to simplify computation
    previous_partial_result = numpy.zeros(num_variables, num_variables)

    for split_index in split_indices:
        split_scores[split_index], previous_partial_result =
mse_fluc_process_score(
            sorted_X, sorted_y, dim, split_index, previous_partial_result,
previous_index)
        previous_index = split_index

    return split_scores
```

Proceeding deeper in the pseudocode implementation of the split-point computation, another for loop is encountered. This time the iteration is over all the indexes of the valid sample points inside of the `X` array, instead of over the number of dimensions that those points possess. A very important detail is the presence of the partial result of the previous iteration, which is used to compute the result of the next one. This means that each iteration requires the completion of the previous one to be able to be computed, creating a sequential dependency. For this reason there is no way to parallelize this loop, unlike one encountered previously.

```
def mse_fluc_process_score(sorted_X, sorted_y, dim, split_index,
previous_partial_result, previous_index):
    # compute the score for a given splitpoint
    # return the score and the partial result for the next iteration
    # to simplify computation

    newlines = sorted_X[previous_index:split_index]
    # compute new partial result using previous partial result
```

```
    # ... heavy math computations ...
    # compute score using new partial result
    # ... heavy math computations ...

    return score, partial_result
```

The pseudocode of the scoring function is even more simplified than the rest, as all the maths operations, which make up the bulk of it, are omitted. This function is where the majority of the execution time is spent as shown later in Chapter 3.1 when profiling is discussed. The maths operations all pretty much depend on the previous operation's result. There are no opportunities here for parallelization.

The only two opportunities that were identified were the simultaneous computation of the recursive call for the left and right child nodes, and the deeper parallelization of the for loop over each dimension of the sample points. The loop case is the most straightforward one of the two. The heavy computation of the best split-point along each dimension can be done in parallel as schematized in Figure 1.4.1. The total parallelism obtainable this way depends on the number of domain dimensions.

This number, it should be reiterated, represents the number of input variables of the model being analysed for explainability. The variables are also called features of the model. The more features the model has, the more computations the algorithm needs to perform. Fortunately the new computation can potentially be done in parallel, but each new dimension also means that each point in the array of input samples `X` also has an additional coordinate. This makes the array bigger and thus slows down any operation that needs to manipulate it.

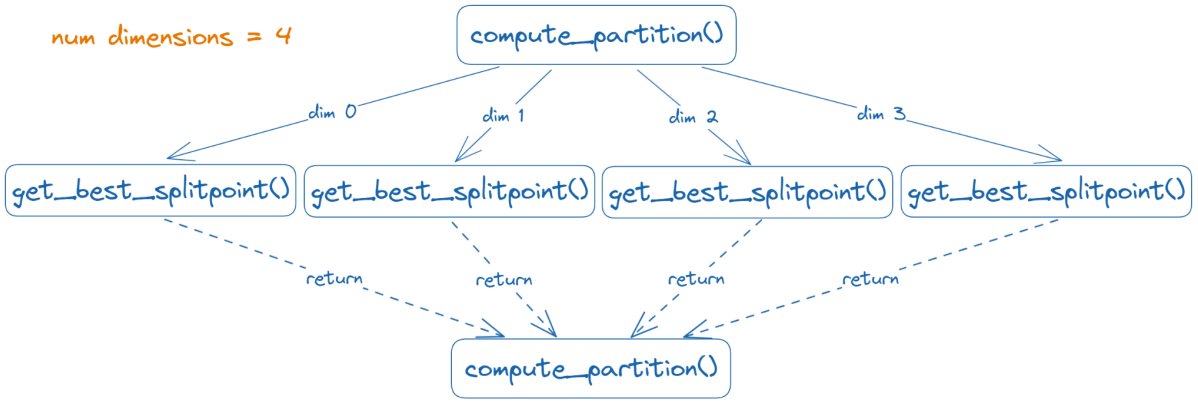


Figure 1.4.1: Computation of best split-point along each domain dimension. Each dimension is independent and could be computed in parallel.

The parallelization of for loops is a pretty common use case and should be supported in a simple way by most parallelization tools and frameworks. More uncommon is the case of the binary recursive tree where each child node could potentially be computed in parallel.

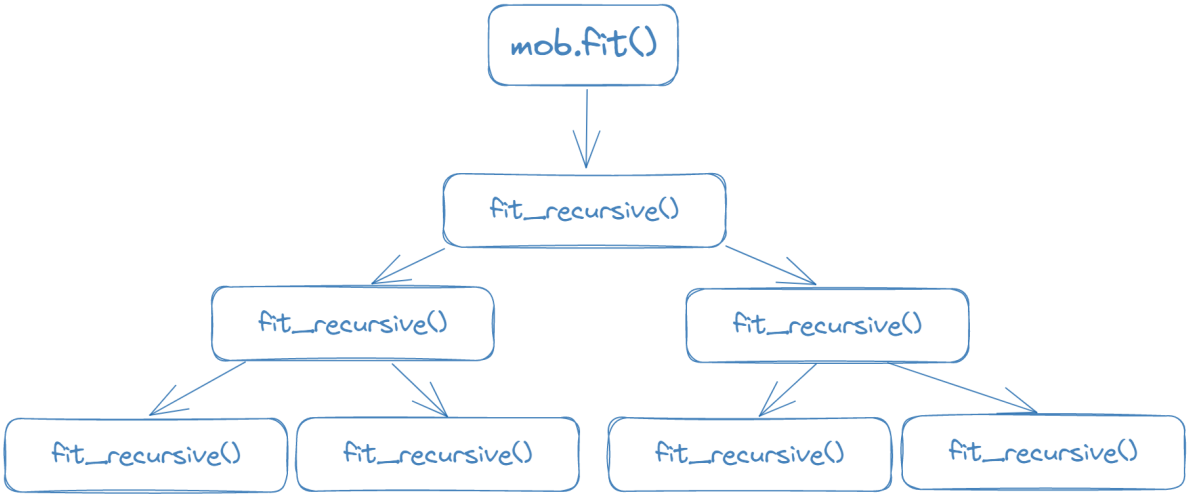


Figure 1.4.2: Recursive computation of mob.fit(). It generates a binary tree structure where each branch is independent.

This model of parallelism is more complex than the loop one. Each call on the same level in the Figure 1.4.2 could be executed in parallel, but not only. There isn't the need for any kind of synchronisation between the levels, so if one side were to finish before the other it could proceed with the deeper levels without having to wait on the other side of the tree. The parallelism would in addition have to be nested by nature of the tree.

To understand the level of parallelism that could be achieved in this spot, the characteristics of the tree need to be analysed. The tree is binary, so there is a maximum breadth that each specific depth level can have, and it corresponds to 2^d with d indicating the depth of the tree with the first level starting with $d = 0$. A node can either have two children or zero children, it can't have only one child.

The total depth of the tree heavily depends on how articulated and complex the original black box model is. The harder the model is to approximate with linear regressions, the deeper the tree needs to grow, to allow further domain segmentation and so greater accuracy. Having a particularly low amount of samples could also limit the depth of the tree, but this would mean losing accuracy as the algorithm would continue to partition if given the chance but doesn't have enough points to do so. Assuming that enough samples are provided, increasing their number beyond that doesn't actually increase the depth of the tree, but can still provide improved accuracy. The number of features or domain dimensions doesn't have any direct impact on the tree. The main controlling factor is the complexity of the original black box model function.

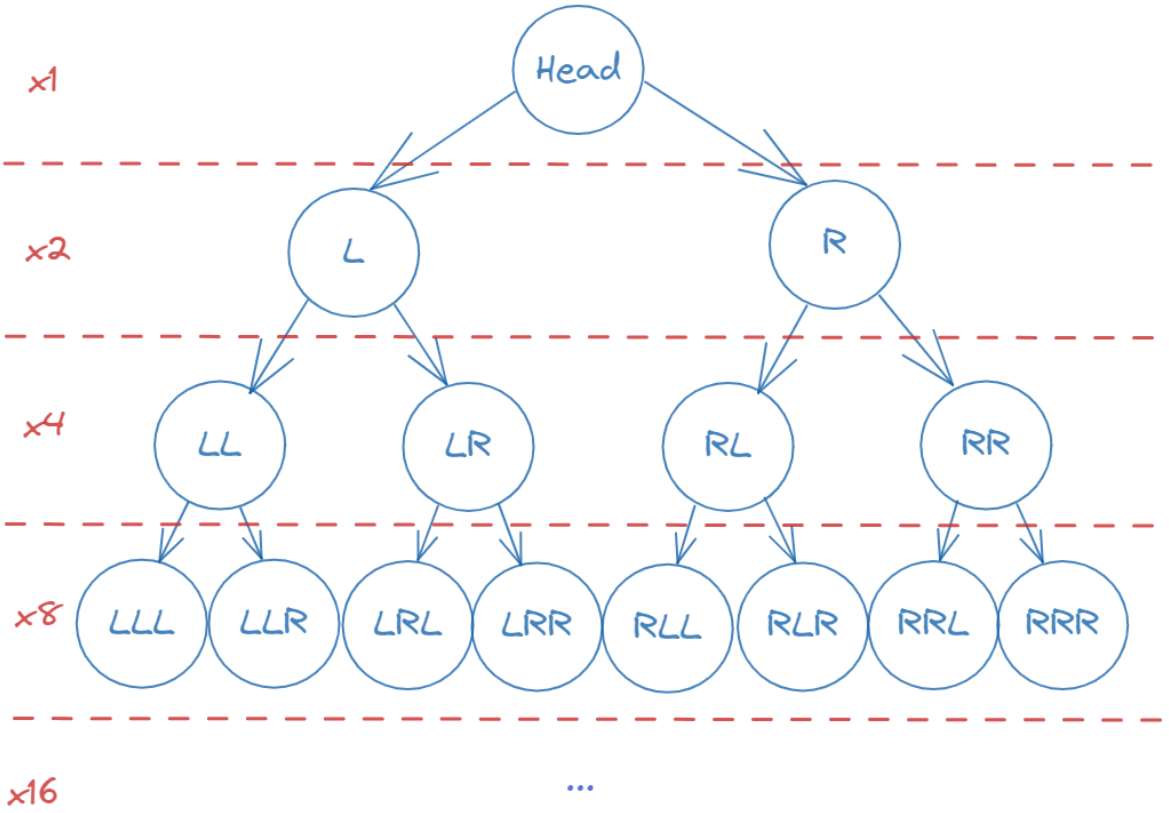


Figure 1.4.3: Maximum parallelism of balanced binary tree computation.

In the above Figure 1.4.3, the tree structure is depicted in the ideal scenario of no leaf nodes and perfectly balanced loads on all branches of the same level. In red on the left is indicated the level of parallelism per each depth level. In this abstract and unlikely scenario the level of parallelism increases exponentially, following the power of two. At the 4th recall there are already 16 nodes being computed at the same time.

In a more realistic scenario though, the tree is not going to be balanced or maintain maximum breadth at all times. In the example shown in Figure 1.4.4, green nodes are leaf nodes, which don't produce further computations, and the level of parallelism is substantially lower than that shown in Figure 1.4.3.

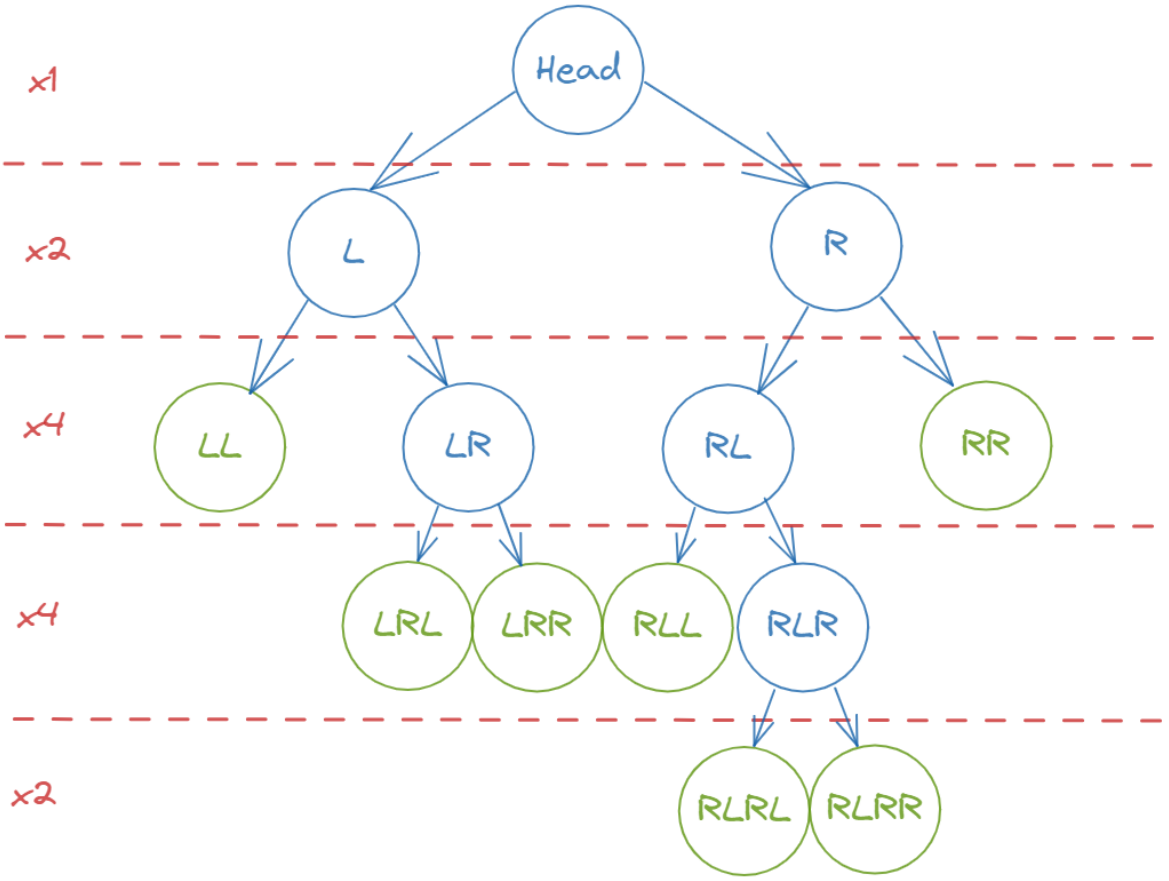


Figure 1.4.4: Parallelism of unbalanced binary tree computation. In green are leaf nodes.

In addition the workload of different branches of the same levels could vary a lot depending on how balanced the split of the subdomains is. A branch could easily have twice the workload of its sibling, making the tree even more unbalanced and potentially reducing the maximum parallelism even further.

Support for nested recursive parallelism is much less common than simple for loop parallelization. Any recursive function can theoretically be rewritten using iterative constructs. This was considered to simplify parallelization, though some important points of this particular recursion case were noted. The total number of `fit_recursive()` iterations that happen can't be determined beforehand.

Whether a particular tree node decides to proceed with further partitions, and thus recursions, can only be known for certain after the computation of the partition is completed and the post partitioning check is performed. That check uses data computed in the partitioning step and its outcome can't be predicted without performing the computation. This means that the iterative version of `fit_recursive()` would also not have a known number of iterations and instead have some sort of structure to keep track of whether more nodes need to be computed.

The simplest solution would be the use of a centralised stack or queue where each node computation would add child nodes that need to be computed. The presence of such a shared data structure though would have a negative impact on parallelization as access to it would need to be synchronised potentially causing overhead due to contention. To avoid such synchronisation issues, approaches with decentralised structures could be considered, but further investigations showed that such solutions would inevitably fall back to resembling the original recursive design, making the conversion to iterative rather futile.

To summarise the findings in this Chapter: Two main parallelization points were identified. One recursive in nature for the parallel computation of nodes of the binary tree structure. The other, deeper in the code and thus nested to the first one, for the parallelization of the computation of the best split-points along each domain dimension. The second one is pretty straightforward and its level of parallelism is directly linked to the number of features of the model. The first one is more complex and requires nested parallelism support.

The most computationally intensive parts of the code, i.e. the maths calculations performed to obtain the score of each potential split value in `compute_mse_fluc_process_criterion()`, aren't parallelizable due to the presence of partial results being used in consecutive iterations. These partial results create a sequential dependency between the iterations but in exchange significantly reduce the total number of computations required to obtain all the scores for a particular domain dimension.

Chapter 2

Selection of the technologies and frameworks

An important step of the development of the parallel solution was the selection of which tools to use to implement it. This chapter will give an overview of all the alternatives that were considered and will start by reiterating the requirements and goals of our particular use case, which are the determining factors behind the choices taken.

2.1 Requirements and goals

To be able to correctly select the best tools and technologies, a list of goals for them was outlined as follows.

A solution should:

1. Offer parallelism on a distributed environment such as the cloud or an HPC system.
2. Offer parallelism on a single local machine, such as a desktop computer or a laptop.
3. Make the conversion from existing sequential code as simple as possible.
4. Offer good performance.
5. Be intuitive and not overly complex.
6. Be easy to maintain and debug.
7. Support dynamic nested parallelism.
8. Be implementable within a limited amount of time.

During the selection of the best alternatives, these requirements were used as determining factors.

2.2 Selection of the Language

The existing sequential code was written in Python and was meant to be used as a Python package, thus whatever solution was chosen it would have to be in some way compatible with it, so that it could still be used as a Python package.

While remaining in Python would seem like the natural solution, Python itself is not the best language when it comes to performance [6]. The main reason for this can be attributed to the fact that Python is an interpreted language and as such it can't make use of optimizations that compiled languages can take advantage of. But this isn't the only reason. Another major limiting factor for Python's performance is the presence of the Global Interpreter Lock, or GIL [7].

This construct is basically a mutex that makes it so only one thread at a time can have access to the interpreter. Its presence is required in order to guarantee the correctness of Python's reference counting and thus memory safety, but it effectively makes CPU bound parallelism

impossible using threads. Python threads can still be useful for IO bound concurrency operations.

Developers can work around this by using multiprocessing, so multiple interpreters, at the cost of resources as processes are much heavier than threads.

The usual way a Python library or package is able to offer good performance is by implementing any performance heavy part in a compiled, lower level, performance-oriented language, such as C or C++ [8]. This approach also has some benefits for thread parallelism because, while a script is executing external code, such as a C or C++ compiled functions, it does not hold the GIL, allowing multiple threads to at least execute external code in parallel.

Python offers a C [9] interface which makes it easy to extend using other languages that also support the C interface, such as C++ and Rust. These two languages in particular were mentioned as they are both modern compiled performance oriented languages.

C++ is a compiled system level language that introduces modern abstractions such as classes and other facilities to C. As performance is one of the main goals of C++, all these abstractions are Zero overhead, or come as close to it as possible [11].

Rust [10] is a much younger compiled system level language (released in 2015), that also aims to offer top tier performance, but with a strong emphasis on compile time correctness. One of the major advantages of Rust is its static memory model [12] which allows checking at compile time for errors which would traditionally either not be checked or have to be managed at run time through a garbage collector. It also offers a more modern and lightweight syntax when compared to the much older C++. On the other hand, being much younger, Rust does not benefit from the vast ecosystem of libraries and frameworks that C++ has.

The use of such languages allows for a level of optimization in the final machine code which is normally not achievable by interpreted languages such as Python. For example, a simple reimplementation of the sequential version in C++ or Rust would be very likely to offer significant performance improvements, potentially even order of magnitude improvements.

As a counter argument to using such languages there is the fact that they would require effectively rewriting the entire algorithm in a different language, potentially even parts that were currently offloaded to other Python libraries, which are not guaranteed to have a counterparts in C++ or Rust, which both have a much smaller ecosystem of libraries, especially in the scope of artificial intelligence.

This could still have been considered if the goal was only to implement a parallel solution for a local setting, so for single machine parallelism. But in our use case, the requirement was to support distributed environments as well, which adds a layer of complexity. Enough

complexity in fact to require the use of an existing solution to abstract from it to respect requirements 3,5,6 and 8.

Many distributed frameworks for C++ and Rust (OpenMP [13], MPI [14]) are pretty low level and still require a lot of work on top of them to implement the final solution.

Python on the other hand, offers many more high level framework options, many of which provide a significantly better experience in terms of ease of use, especially if compared with the C++ ones. Some performance would be traded in exchange for ease of implementation and other advantages such as ease of maintainability.

There are a lot of Python developers who don't have the knowledge or expertise required to use a different language in order to improve performance of their projects and take advantage of parallelization. Learning said language would take time and not guarantee a high quality result. Our situation is common enough that there exist various alternatives and solutions in the form of frameworks and libraries.

2.3 Selection of the Framework

An investigation in this space resulted in the identification of the following frameworks which could be relevant: Spark, Dask. Ray and Numba.

Numba [15] is different from the other libraries mentioned in the sense that its purpose isn't related to parallelization or distributed computation, but rather it simply focuses on performance. Numba is a just-in-time compiler for Python that works best on code that uses numpy arrays [56] and functions. The aim is to offer better performance by compiling very computationally intensive parts of the code, this way removing the interpreter's overhead.

The idea would be to pair Numba with one of the other frameworks to obtain native like performance, whilst remaining in Python land and with minimal changes to the code. Numba offers unintrusive ways to quickly and easily mark functions for JIT compilation. We decided it would be worth trying Numba for our use case. There are some limitations and cases where Numba doesn't work well. These will be examined later in the implementation chapter.

Spark [16] is an engine for large scale data processing. It was started in 2009 and has seen widespread adoption in commercial settings. It is a mature framework, but it isn't Python native. It is developed in Java and relies on the JVM (Java Virtual Machine) to run and Python is just one of the languages that it is able to interface to.

Its primary focus is large scale clusters and not single machine parallelism. It is possible to run a Spark cluster on a single machine, but it does not offer a straightforward interface for that purpose.

Its wide adoption would mean likely better integration, support, and a community of other users that might have already encountered and resolved most common problems from which to learn. On the other hand Spark is the oldest framework from our list, making its concepts and the tools it offered also seem dated in comparison to the newer solutions. In particular Spark seemed to offer a more rigid approach from the other frameworks and had a higher learning curve.

Ray [17] is a unified framework for scaling AI and Python applications. It offers a toolkit of libraries for Machine Learning computation as well as a simple yet powerful Core API which can be used to parallelize custom applications on a local machine or cluster.

Ray Core [18] is what would be of interest in our use case. It is supported by an underlying C++ layer that handles most of the heavy tasks, making it very performant. The focus of the Ray project though is mostly on ML and Artificial Intelligence high level distributed computation libraries, which in turn are implemented using Ray Core primitives. The utilities offered by the ML centric libraries are completed solutions to generate ML models, meant for end users, and are not tools to parallelize algorithms such as MOB. This made Ray a bit less attractive than the last option, which was Dask.

Dask [19] is a Python native library for parallel computing. It was developed as a way to easily scale Python applications across multiple machines. The original focus was the parallelization of numpy operations, but it expanded to offer a lot of high level functionalities.

Dask is implemented completely in Python which means potentially lower performance than Spark or Ray, but has some benefits, which are ease of debugging and maintenance. There is no obscure underlying layer, everything is comprehensible for a Python developer.

In addition to parallelization of CPU intensive tasks, Dask also offers a lot of tools to handle data exceeding the size of the physical memory. This is of particular interest in Data science and Machine Learning as arrays and collections of data to analyse and compute are getting larger and larger. It was determined at the start of the project that the focus of the effort would be on improving performance and not on handling large data, but Dask undoubtedly offered an interesting proposition in that sense.

Dask's API is more diverse and higher level than what Ray is offering, but Ray offers higher performance according to many comparisons available online [20], [21], [22]. One interesting thing that was found was the Dask-on-Ray project [23], which offered a subset of Dask's API running on the underlying Ray scheduler, for higher performance.
At this point we decided to try using Dask for the project and fallback on Dask-on-Ray if needed.

2.4 Testing Dask

One of the peculiarities of our use case is the nested parallelism inside the recursive tree of unknown depth. Thus what we needed from the framework was support for dynamic execution graphs.

The standard way to use Dask is to adopt one of their collections, such as Dask Array, Dask DataFrame etc, and employ the methods of those collections to generate a Task Graph of operations to perform on them. All operations are delayed in the sense that nothing happens until the user calls the `collection.compute()` method, which actually schedules the tasks on the distributed cluster. Dask allows arbitrary operations to be scheduled on the graph in addition to the ones from their collections methods.

This delayed graph way of computation can be very efficient as Dask is able to figure out automatically the dependencies between all the tasks in the graph and schedule independent tasks on parallel cores. In general, having the complete graph offers many opportunities for optimizations, both memory and performance wise.

This kind of computation though is not compatible with our use case, where we have a recursive binary tree, which is of unknown depth. The decision to continue in further depth depends on the results of the computation at the last tree node, which makes creating a complete task graph of all the operations to perform impossible. The graph does not allow optional branches or any kind of control flow.

Dask offers a second method of operation, which it refers to as their Futures API [24]. This API is a real-time task framework that extends Python's `concurrent.futures` interface [25]. It allows arbitrary task scheduling, but in this case the tasks are executed immediately and not lazily like for the delayed operations.

While recognizing it as a very powerful API, the Dask documentation encourages users to prefer the Dask delayed interface when possible [24], as Dask futures are lower level and require explicit handling of client concurrency.

For the purpose of the project, the Dask documentation was studied in detail and the first tests were performed with a recursive problem (recursive fibonacci sequence computation) on a single machine with these specs:

- 8-core 16-thread AMD Ryzen 7 3700x Processor (3.6GHz - 4.4GHz).
- 32GB 3200MHz DDR4 Memory.
- 970 EVO Plus NVMe M.2 SSD 1 TB Storage (read/write speed: 3.500/3.300 MB/s).

The following code was used:

```
import time
```

```
# Sequential Fibonacci series using recursion
def fibonacci(n):
    if n <= 1:
        return n
    return fibonacci(n-1) + fibonacci(n-2)

n = 40

start_time = time.perf_counter()
res = fibonacci(n)
end_time = time.perf_counter()

elapsed_time = end_time - start_time
print(f"Timing: {elapsed_time}, fibonacci({n}) = {res}")

# Output:

# Timing: 23.755997299999763, fibonacci(40) = 102334155
```

The first attempt to parallelize the recursion using Dask was the following:

```
from dask.distributed import Client, get_client

def fibonacci_dask(n):
    if n <= 1:
        return n

    # Get locally created client
    client = get_client()

    a = client.submit(fibonacci_dask, n-1)
    b = client.submit(fibonacci_dask, n-2)

    a, b = client.gather([a, b])

    return a + b

client = Client()
n = 40
start_time = time.perf_counter()
```

```
res_ref = client.submit(fibonacci_dask, n)
res = res_ref.result()

end_time = time.perf_counter()
elapsed_time = end_time - start_time
print(f"Timing: {elapsed_time}, fibonacci({n}) = {res}")
```

This code apparently deadlocked and never completed execution. The reason behind this is the recursive nature of the tasks, where each task has to wait for the result of the children's tasks. Effectively each task has to stay alive until all children have completed.

This behaviour causes issues because, in order to schedule tasks, Dask uses a pool of workers, which are distributed on the available machines. To avoid using resources that are not available, the scheduler will wait for a worker to be free. If no workers are free, no further tasks are scheduled. In the above case all available workers are quickly blocked in waiting for further results, but those results will never be computed because no workers are available to handle them.

This is clearly a big problem, but to remedy it Dask proposes a solution in the form of two primitives, `secede()` and `rejoin()` [26]. The user would have to make use of them inside the worker that is about to block on results to effectively remove the worker from the active workers pool and later rejoin it. Upon the seceding of a worker, Dask instantiates a new one to replace it, which allows computations to progress. Upon rejoining, the Dask documentation isn't as clear, but it can be assumed that the excess workers are destroyed once they have completed their tasks.

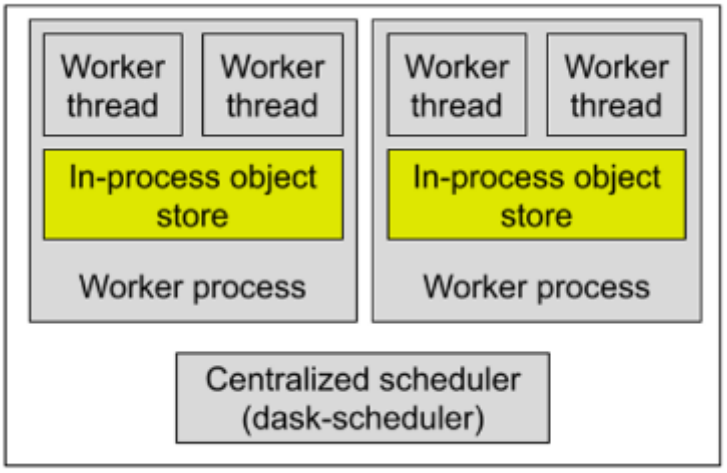


Figure 2.4.1: Dask workers structure. Each machine can have multiple worker processes, each of which can have multiple worker threads. The total number of worker threads on a machine should equal the number of CPUs cores available. [22]

In Dask, workers are Python threads, hosted by multiple Python processes, where each process hosts multiple threads as shown in Figure 2.4.1. The number of workers is normally kept equal to the number of CPUs on the machine while the number of processes is usually half or a fourth of that. This allows for lightweight workers but at the cost of potential parallelism, as Python threads of the same process have concurrency limitations due to the presence of the GIL. The assumption is that most of the tasks that would actually execute on these workers would be in the form of external compiled code anyway, either from the standard python library or from libraries such as numpy, which all use compiled native code underneath to improve performance, avoiding most GIL locking during the long operations.

The modified code looks like this:

```
from dask.distributed import Client, secede, rejoin, get_client

def fibonacci_dask(n):
    if n <= 1:
        return n

    # Get locally created client
    client = get_client()

    a = client.submit(fibonacci_dask, n-1)
    b = client.submit(fibonacci_dask, n-2)

    # Remove the current worker from the active workers pool
    secede()
    a, b = client.gather([a, b])
    rejoin()

    return a + b

client = Client()
n = 40
start_time = time.perf_counter()

res_ref = client.submit(fibonacci_dask, n)
res = res_ref.result()

end_time = time.perf_counter()
elapsed_time = end_time - start_time
print(f"Timing: {elapsed_time}, fibonacci({n}) = {res}")
```

```
# Output:

# Timing: 0.9990950999999768, fibonacci(40) = 102334155
```

This time the code completed successfully offering very good performance when compared to the sequential version. In fact a 23x speedup.

While the speedup was very good, the `secede()` and `rejoin()` semantics aren't as good. Having an opening and closing type of API is unadvisable and can lead to user error and makes maintainability of the code worse. Their functioning and goal isn't easy to discern and future maintainers might have trouble while implementing changes to the code. Dask offers an alternative in the form of a context manager, `worker_client()`.

```
with worker_client(timeout="10s") as client:  # connect from worker back to
scheduler
        a = client.submit(inc, x)     # this task can submit more tasks
        b = client.submit(dec, x)
        result = client.gather([a, b])  # and gather results
```

The context manager implicitly secedes the worker at the start and performs the rejoin at the end of the indented section. This alleviates somewhat the maintainability issues, but also allows much less fine grained control over which operations are performed while part of the active workers pool and which to perform outside of it.

In general, we didn't like the workarounds that Dask required to handle nested concurrency and weren't impressed by it. The use of multiple threads of the same processes was also worrying, but this first test showed great performance regardless. So we decided to move onto another test, this time more similar to our use case. This time it was the implementation of a parallel version of the quicksort algorithm. The code was actually adapted from an example found in the Ray documentation about Ray nested parallelism support [27].

```
from dask.distributed import Client, secede, rejoin, get_client
import time
from numpy import random


def partition(collection):
    # Use the last element as the pivot
    pivot = collection.pop()
    greater, lesser = [], []
    for element in collection:
        if element > pivot:
```

```
            greater.append(element)
        else:
            lesser.append(element)
    return lesser, pivot, greater


def quick_sort(collection):
    if len(collection) <= 200000:  # magic number
        return sorted(collection)
    else:
        lesser, pivot, greater = partition(collection)
        lesser = quick_sort(lesser)
        greater = quick_sort(greater)
    return lesser + [pivot] + greater


def quick_sort_distributed(collection):
    # Tiny tasks are an antipattern.
    # Thus, in our example we have a "magic number" to
    # toggle when distributed recursion should be used vs
    # when the sorting should be done in place. The rule
    # of thumb is that the duration of an individual task
    # should be at least 1 second.
    if len(collection) <= 200000:  # magic number
        return sorted(collection)
    else:
        lesser, pivot, greater = partition(collection)

        client = get_client()

        sorted_lesser_ref = client.submit(quick_sort_distributed, lesser)
        sorted_greater_ref = client.submit(quick_sort_distributed, greater)

        secede()
            sorted_lesser, sorted_greater = client.gather([sorted_lesser_ref,
sorted_greater_ref])
        rejoin()

        return sorted_lesser + [pivot] + sorted_greater


client = Client()
```

```
for size in [200000, 4000000]:
    print(f"Array size: {size}")
    unsorted = random.randint(1000000, size=(size)).tolist()
    s = time.time()
    quick_sort(unsorted)
    print(f"Sequential execution: {(time.time() - s):.3f}")
    s = time.time()
    client.submit(quick_sort_distributed, unsorted).result()
    print(f"Distributed execution: {(time.time() - s):.3f}")
    print("--" * 10)

# Outputs:

# Array size: 200000
# Sequential execution: 0.031
# Distributed execution: 5.650
# --------------------
# Array size: 4000000
# Sequential execution: 6.329
#  c:\Dev\Repos\dask-tests\.venv\lib\site-packages\distributed\worker.py:2972:
UserWarning: Large object of size 18.58 MiB detected in task graph:
#   ([34854, 443542, 282663, 272069, 966786, 500222, 2 ... 3162, 358036],)
# Consider scattering large objects ahead of time
# with client.scatter to reduce scheduler burden and
# keep data on workers

#     future = client.submit(func, big_data)    # bad

#     big_future = client.scatter(big_data)     # good
#     future = client.submit(func, big_future)  # good
#   warnings.warn(
# Distributed execution: 433.067
# --------------------
```

From the results reported above we can see how Dask seems to perform quite abysmally in this scenario, two orders of magnitude worse than the sequential version in fact.

From the warning that appeared in the output we could hypothesise the problem to be related to the presence of large objects being passed around between tasks. The warning suggests the use of the `scatter()` operation, which would split a collection object into chunks to be subdivided between the workers in the Dask cluster [28]. In this use case though, scattering

the collection doesn't offer any advantage as the entire collection object is required by the individual worker that will handle further sorting.

From the documentation we were unable to find a solution that would alleviate memory overhead of transferring the data, without changing the structure of the quicksort code. Dask uses a mix of processes and threads for its workers (Figure 2.4.2), but threads from different processes don't have any kind of shared memory between each other to efficiently transfer or share data. To make the situation worse, Dask worker threads that are spawned from the same process, and consequently are actually able to share memory, suffer from the Python thread concurrency limitations derived from the presence of the GIL.

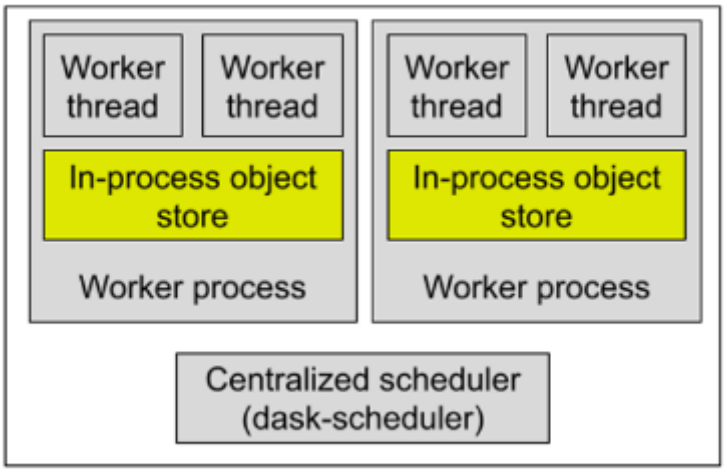


Figure 2.4.2: Dask workers structure. [22]

This result and the inability to reliably use Dask in nested concurrency scenarios finally convinced us to give up on Dask and try Ray in its place. Using Dask-on-Ray was briefly considered, but it was soon discovered that Dask Futures [24] weren't part of the subset of supported Dask APIs [23], forcing the use of Ray concepts to handle the dynamic tasks. At that point using Dask-on-Ray only meant adding complexity, and the use of pure Ray was preferred for simplicity.

2.5 Testing Ray

Like Dask, Ray required a simple pip install to get set up and after researching thoroughly its documentation we put it to the test to see whether it could prove a better solution for our use case. Before going into the tests we will quickly go over the basic concepts of Ray to have a clearer picture of how it works and how it is used. Ray Core is built around 3 key concepts: tasks, actors and remote objects [18].

Ray *tasks* allow arbitrary functions to be executed asynchronously on separate python worker processes. To define a task a user simply needs to annotate a function with the `@ray.remote` decorator like this:

```
import ray

@ray.remote
def my_func(n):
    # Do some some work
    return n * 2
```

And the task is executed by adding `.remote()` to the function call like this:

```
future = my_func.remote(10)  # asynchronous remote call
result = ray.get(future)  # blocking call to get the result
```

The remote call schedules the execution of the function on a Ray worker, either on the local machine or on a remote one in the case of a cluster. Even in the case of a cluster the worker could still be in the local machine unless all the available workers on that machine are already busy.

Ray *actors* are essentially stateful workers, and are created by annotating classes with the Ray remote decorator:

```
@ray.remote
class Counter:
    def __init__(self):
        self.i = 0

    def incr(self):
        self.i += 1
        return self.i
```

The actor is instantiated like this:

```
actor_counter = Counter.remote()
```

And its methods can be called like this:

```
future = actor_counter.incr.remote()
val = ray.get(future)
```

Each actor instance is assigned a single dedicated worker on which are scheduled all the method calls of that instance. The method calls are guaranteed to be executed sequentially

and in order, which makes mutation of the internal state simple to handle. It is possible to create special asynchronous actors which forgo the order and sequential guarantee, in which case it is the responsibility of the developer to maintain coherence of the internal state.

For the workers, Ray uses processes rather than threads, to avoid incurring in GIL mutual exclusion. Each node hosts by default a number of workers equal to the amount of logical CPU cores of the node. These workers form a pool which is then used to schedule tasks. Actors on the other hand usually get a dedicated new worker outside of the pool mentioned before.

In Ray, tasks and actors produce and process objects. Since these objects can be stored anywhere in a Ray cluster, they are referred to as remote objects and object references are used to identify them. Each node of the cluster has an object store shared between its workers, which is used as a shared-memory cache for the distributed cluster-wide object store. A remote object can reside on one or more node's object stores, regardless of who has the object reference.

The shared memory on each node allows for quite efficient memory usage, especially in the case of multiple workers having to use the same data. The data is also only transferred to the shared object storage of nodes that actually need it, avoiding unnecessary network overhead. All Ray remote objects are read-only, avoiding any consistency issue between replicas.

In addition to the shared-memory, each worker process has its own private memory, which is what is used for all the internal objects it needs to allocate or manipulate. In essence, the object store is usually used for the input and the output data of a task, while everything else resides in the worker's private memory.

The usual and suggested way to use Ray to parallelize a workload is the following [18]:

```
# Define the square task.
@ray.remote
def square(x):
    return x * x

# Launch four parallel square tasks.
futures = [square.remote(i) for i in range(4)]

# Retrieve results.
print(ray.get(futures))
# -> [0, 1, 4, 9]
```

The original Python process that starts to execute the script containing Ray code is called driver process. This process then either connects to a Ray cluster or generates a new local one to handle the remote function calls. To obtain parallelism the user simply invokes multiple asynchronous remote functions before waiting for the combined list of remote results references.

In contrast to Dask, Ray Core seems to focus more on parallel task execution rather than parallel data elaboration, providing a more limited API when compared with all the collection based APIs found in Dask. This difference can also be seen when looking at data management. Dask only offers the `scatter()` primitive [28] to divide a collection type object between workers, while Ray offers the `put()` primitive, which instead preemptively stores an entire object onto the shared object store and returns a reference to it [29].

The `put()` function combined with the shared memory store available on each node, allows developers to achieve much better control and efficiency of data that needs to be processed. This in particular stood out to us as a great advantage over what was seen with Dask.

For the actual test, the quicksort example directly from the Ray documentation was used, in particular from a section showcasing nested parallelism with Ray [27]. This same example was adapted before for the Dask tests.

```
import ray
import time
from numpy import random


def partition(collection):
    # Use the last element as the pivot
    pivot = collection.pop()
    greater, lesser = [], []
    for element in collection:
        if element > pivot:
            greater.append(element)
        else:
            lesser.append(element)
    return lesser, pivot, greater


def quick_sort(collection):
    if len(collection) <= 200000:  # magic number
        return sorted(collection)
    else:
```

```
        lesser, pivot, greater = partition(collection)
        lesser = quick_sort(lesser)
        greater = quick_sort(greater)
    return lesser + [pivot] + greater


@ray.remote
def quick_sort_distributed(collection):
    # Tiny tasks are an antipattern.
    # Thus, in our example we have a "magic number" to
    # toggle when distributed recursion should be used vs
    # when the sorting should be done in place. The rule
    # of thumb is that the duration of an individual task
    # should be at least 1 second.
    if len(collection) <= 200000:  # magic number
        return sorted(collection)
    else:
        lesser, pivot, greater = partition(collection)
        lesser = quick_sort_distributed.remote(lesser)
        greater = quick_sort_distributed.remote(greater)
        return ray.get(lesser) + [pivot] + ray.get(greater)

ray.init()
for size in [200000, 4000000, 8000000]:
    print(f"Array size: {size}")
    unsorted = random.randint(1000000, size=(size)).tolist()
    s = time.time()
    quick_sort(unsorted)
    print(f"Sequential execution: {(time.time() - s):.3f}")
    s = time.time()
    ray.get(quick_sort_distributed.remote(unsorted))
    print(f"Distributed execution: {(time.time() - s):.3f}")
    print("--" * 10)

# Outputs:

# Array size: 200000
# Sequential execution: 0.030
# Distributed execution: 0.077
# --------------------
# Array size: 4000000
# Sequential execution: 6.129
```

```
# Distributed execution: 5.122
# --------------------
# Array size: 8000000
# Sequential execution: 13.689
# Distributed execution: 8.951
# --------------------
```

As the results show, the greater the input size, the more Ray's version is able to improve performance. While the performance gains weren't substantial, Ray at least didn't introduce a massive performance regression like Dask did. Encouraged by these results we investigated how Ray handled the situation of workers blocking on results that was problematic for Dask.

Looking at whitepapers and github issues discussing this, we found that Ray's workers architecture has essentially the same issue that Dask had when it came to waiting for results inside a task, but the approach to address the issue was different. To avoid the possibility of a deadlock, where all worker processes are busy waiting on remote results, Ray automatically removes workers calling `ray.get()` from the worker slots pool and instantiates a new one if there are enough tasks still pending. This isn't particularly better than Dask, the main difference stands in the fact that in Ray this is automatic and doesn't require careful handling with `secede()` and `rejoin()` functions like Dask did. The situation in Ray could actually be a little worse when it comes to performance due to Ray workers being heavy processes instead of a mix of processes and threads, like Dask used. In exchange Ray doesn't have to worry about loss of parallelism of threads contending the GIL.

In some experiments performed by the Dask-on-Ray team and reported in a blog post [22], the difference in memory management performance between Dask and Ray were highlighted. The experiments measured the time taken to broadcast a large object to a large number of tasks. Some tests were performed where the tasks required retention of the GIL while in others access to the GIL was avoided.

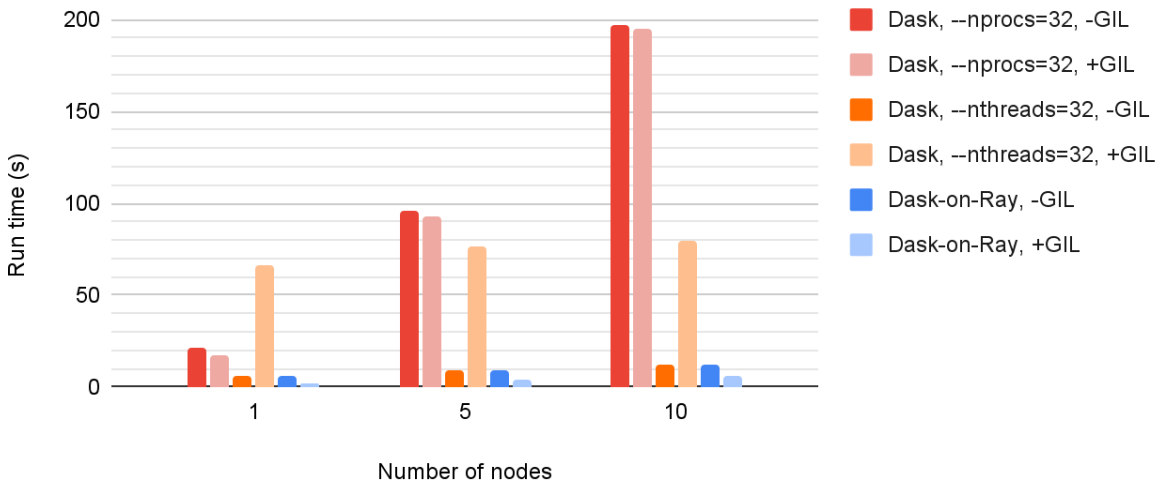


Figure 2.5.1: Measuring the time to pass a 1GB object to many tasks (100 tasks/node). [22]

Figure 2.5.1 shows how Dask using processes for workers (dark and right red) performs poorly regardless of whether the GIL was involved or not. Dask with multithreading can avoid unnecessary memory copies (dark orange), but performance suffers when the tasks require holding the GIL (light orange). The Ray based solution instead manages to perform very well regardless of the involvement of the GIL, thanks to the presence of the shared object storage, which avoids memory copies even though Ray workers are made of separate processes.

After having studied both Ray and Dask's documentations in detail and performed tests with both, we concluded that Ray still offered the better fit for the goals of our project, even though as a project it is more focused on high level ML utility libraries. Ray offers simple tools to adapt existing code, it isn't overly complex and offers the best performance.

Chapter 3

Implementation of the parallel solutions

This chapter will go over all the steps taken for the implementation of the solutions. The initial approach, profiling, optimizations and the parallel solutions themselves, explaining how they work and the differences they have.

3.1 Profiling

Before implementing anything, one of the first things we did was to look at the performance of the existing sequential code through profiling, to better understand which parts of the code were the most demanding and see whether there were any opportunities for optimizations.
Profiling is the process of analysing the code execution through tools, to extrapolate metrics regarding performance or resource utilisation. In our use case we are particularly interested in execution time and memory utilisation.

We looked at what tools were available to profile Python code. The most common solution seemed to be `cProfile`, which is a deterministic execution time profiler available in the Python standard library and has reasonable overhead [30]. A deterministic profiler distinguishes itself by monitoring all function call, function return and exception events, and precise timings are made for the intervals between these events (during which time the user's code is executing). In contrast, statistical profiling randomly samples the effective instruction pointer, and deduces where time is being spent. The latter technique traditionally involves less overhead (as the code does not need to be instrumented), but provides only relative indications of where time is being spent.

In Python, deterministic profiling may be performed without instrumented code because an interpreter is always running as the program is being executed. For each event, Python automatically provides a hook (optional callback). Deterministic profiling typically only adds a small amount of processing overhead in ordinary applications because Python's interpreted nature tends to already bring much more overhead to the execution. As a result, deterministic profiling is not overly expensive while still offering rich run time statistics about a Python program's operation.

For each function call, `cProfile` provides the call count, the cumulative time and the internal execution time. The latter is the execution time spent inside the function while not being in a lower level function, while cumulative time considers the total execution time of a function from start to end. Through these metrics, it is possible to not only find the hot path of the code, but also identify slow external functions, bugs or unexpected expansions of code such as nested loops.

In combination with `cProfile` we used *`gprof2dot`*, a Python module that converts profiling output to a graph [31]. One such graph of the sequential code is shown below in Figure 3.1.1.

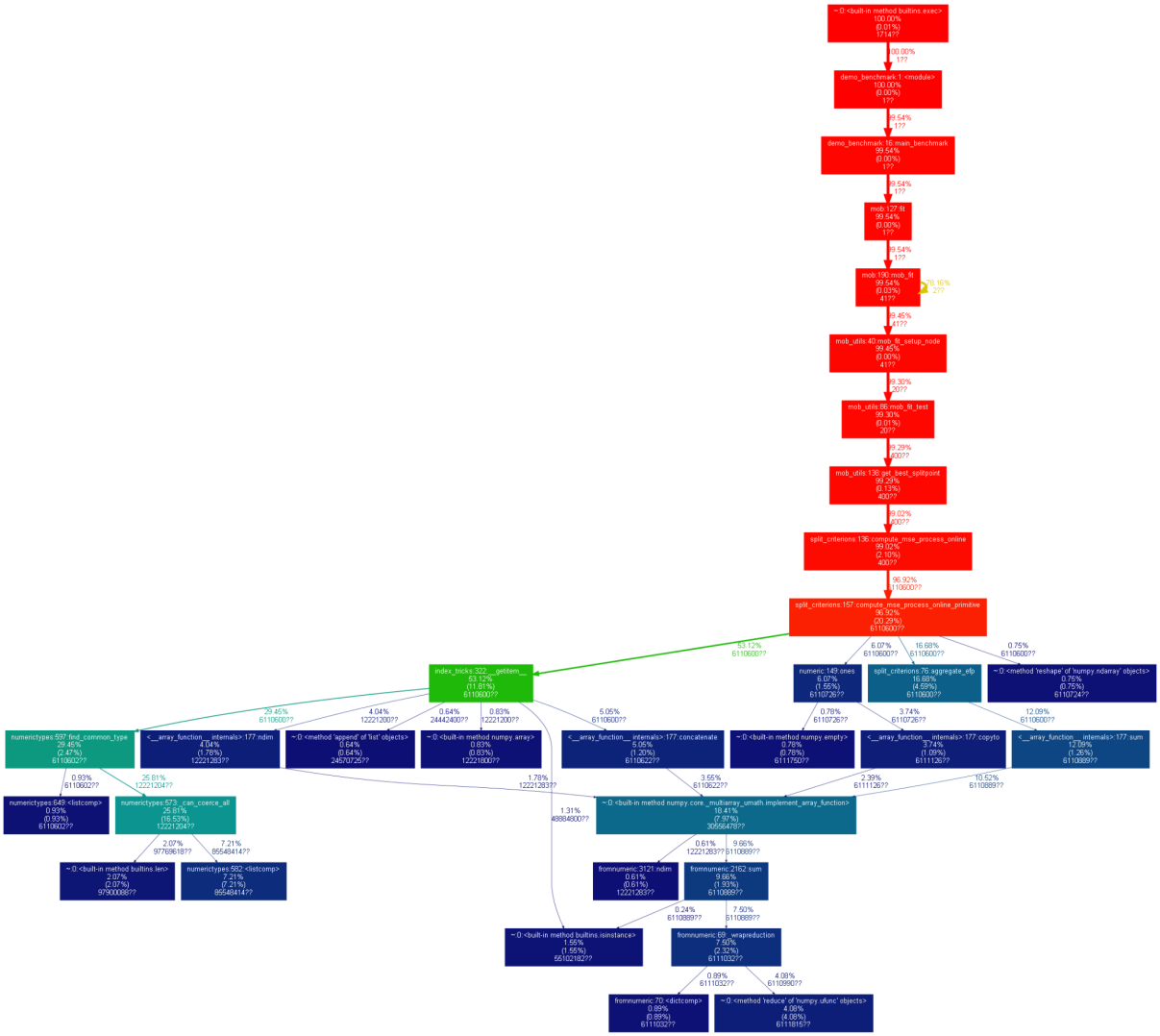


Figure 3.1.1: Example of gprof2dot profiling output graph of unoptimized sequential GLEAMS code. While the text is too small to be readable, the shape of the graph suggests a hot-path leading to a base of various expensive operations at the bottom.

While this information was useful, it didn't give a direct correlation to the code when presented this way. It gave a good overview of the hotpath and what to look out for but not the line by line timings.

For this purpose we researched alternatives that would show the cumulative and internal execution time of each line of code. We looked at both `line_profiler` [32] and `pprofile` [33], which offer outputs with both the code and the execution time, but what we ended up using was `Austin` [34].

`Austin` is a statistical Python profiler implemented in C which distinguishes itself as the only Python profiler to offer an IDE integration to visualise the results in the editor. The `Austin` developer has provided a VS Code extension which allows users to open `Austin` profile output files and view a call flame graph and, more importantly, inline code timing annotation in the editor [35].

`Austin` wasn't without issues, as its profiling output files could be several gigabytes big, and loading them could take several minutes to hours, depending on the length of the script. These issues have been brought up with Austin's developer, who acknowledged them and proposed a workaround in the meantime to use a compression step on the profiling output, which reduced both the size and load times.

With the inline timing annotation we were able to better understand which parts of the code were the ones responsible for most of the execution time. The disadvantage of using `Austin` is that it takes a long time to collect and analyse the data. During development, `line_profiler` was used instead to quickly compare the execution profile of different iterations of specific functions.

Since both `cProfile` and `line_profiler` don't offer any memory related metrics, `memory_profiler` [36] was used as an alternative to `line_profiler`. It outputs the executed code with memory usage annotations. This allows the users to see how memory usage increases or decreases line by line. The information shown is only the total memory allocated at the end of the line and not an indication of actual memory access. A particular line of code could hide a function that allocates and deallocates a large amount of memory without the profiling reporting it. Only by analysing on multiple depths it is possible to get a clearer picture of memory utilisation. Memory profiling was mainly used later in development to improve memory efficiency of the parallel solutions.

In addition to all the mentioned profiling tools, the debugger was also used extensively during development to better understand the code and verify execution behaviour of implemented changes. The Visual Studio Code `debugpy` debugger was used for this purpose [37].

3.2 Regression testing

Using the various tools detailed above it was possible to identify various opportunities for optimization, but before implementing any changes to the code, a way to verify the correctness of these changes was required. Without it, it would be too easy to introduce bugs or regressions. Especially since the output of the algorithm is an approximation and thus it can be hard to spot deviations.

Considering that some of the changes, especially for the parallel implementations, would be likely to require non minimal refactors, it was decided that the best solution would be to set

up a series of top level tests, where, for consistent inputs to the algorithm, consistent results would be expected.

To verify that the results would not change, we decided to directly compare the generated binary tree objects between a reference version, generated using the baseline sequential version (assumed to be correct), and the tree generated by the latest code modification.

To avoid having to generate the correct version for each comparison, the tree object was serialised to file using the standard Python pickle library. Some changes were required to both allow the serialisation of the object and more specifically the comparison.

For the purpose of obtaining a useful comparison, the customization of the `__eq__()` method of the tree node class and some of its member types was required. Particularly, some of the members of the tree had to be excluded from the comparison, such as the `node_id` since they would not be guaranteed to be consistent across runs, especially when considering parallel implementations, where the order of computation of the nodes could be different. Special care was also required for comparing numpy arrays and comparing the child nodes recursively.

3.3 Measuring performance

For the sake of measuring performance improvements obtained through optimizations or parallel implementations of the code, some kind of benchmark was needed. For that purpose a `demo_benchmark.py` script was created.

The script uses the regression benchmark problem *Friedman #1*, described in [38] and [39], to generate the sample data over which mob needs to fit the binary tree.

In practice, the `sklearn.datasets.make_friedman1()` [40] function is used to generate an array `X` of 2^m sample points with `dim` dimension and the corresponding array `y` of result values. To make the benchmark run computationally more complex, a random gaussian noise of fixed seed and standard deviation of 1 was included.

Friedman1 in this case would be the black box model that MOB needs to approximate. `X` are the sample points from the domain, and `y` the output of the Friedman1 function for each point of `X`. The added noise makes the Friedman1 model more obscure and complex, so a better target for MOB.

The benchmark function then measures the time it takes for MOB to fit the `X` sample points into a binary tree of sub-partitions with associated linear regression models. This timing is obtained through differences of Python's `time.perf_counter()` standard function, which should offer a high precision wall time difference.

The two most important parameters of the benchmark function are *m* and *dim*, which define the size of the input data and the length and complexity of the computation.

In addition to the benchmark itself, some more functions were implemented in `demo_benchmark.py`. It features the code required to execute the correctness check mentioned before and contains facilities to automate multiple benchmark passes with different parameters. A system to collect and save all the benchmarks parameters and results to CSV format was implemented.

As a secondary means of measuring performance, outside of the custom made benchmark, a progress bar was added inside the main MOB code. This progress bar was implemented through the use of the `tqdm` package [41] and, in addition to showing the total execution time, it also served the purpose of showing signs of life and progress during the general use computations.

The one hundred percent value corresponds to the total number of points, and each time a node is marked as a leaf, the number of points associated with that node is added to the internal counter of the progress bar, to mark the progress.

3.4 Optimizations

Through the use of profilers, it was possible to find the sections of the code that took longer to execute. This was not only valuable to better understand how to best parallelize it, but also to identify any parts of the code that overly slowed down the execution.

To try to improve the overall performance, all the functions that single handedly took more than 2% of the total execution time were examined for improvement opportunities. The considered changes ranged from completely avoiding the computation to switching to an alternative implementation that performed better.

These efforts produced great results. More than ten different optimization changes were implemented as a result of extensive profiling. The performance gains that these changes afforded will be shown in better detail in the next chapter where all the performance results are reported and examined.

Since the code was still in development during these optimization efforts, a couple of the most obvious and straightforward optimizations were integrated directly in the main code, and so are not included in the final optimization speedup numbers. To anticipate the results, the average speedup of the optimised version over the final regular one, was around 2.68x. Some additional tests suggested that if the optimizations that got included in the regular version were to be reversed, the optimised version would be more in the range of 3.3 times faster.

The optimizations can be subdivided in two kinds, the first one can be summarised with using better alternative methods to achieve the same things, while the second kind is more memory related and aims to reduce both the number of memory allocations and total memory used.

Some examples of the first kind were the use of `numpy.sum()` instead of the Python standard library `sum()`, the use of list literals `[]` rather than `list()`, the use of `numpy.concatenate()` instead of `numpy.c_()` or the use of `numpy.reshape()` instead of `array[numpy.newaxis]` to add new columns.

Some more complex examples were the switch from `sum(array)!=0` to `numpy.any()` to check whether a mask array contained any active values, or the use of the `return_index=True` flag in the `numpy.unique()` function to directly obtain an array of indexes instead of manually generating them through a loop. These two optimizations were significant enough to be included in the regular code during refactors.

The correctness check we implemented and mentioned before helped avoiding some hard to spot regressions that were introduced during the optimization efforts, such as a subtle difference between `list()` and the `[]` list literal. The list literal would always wrap the contained object inside a new list, while the `list()` function would avoid it when passed an object that was already a list. This was resolved by changing the lists in question into numpy arrays, which also happened to offer better performance.

For the second kind of changes, those regarding memory efficiency, the main goal was to remove memory allocations if possible from the innermost loop of computation. The function that through the use of the derivative of the Mean Square Error, incrementally computed the scores of the potential split-points. One of the operations in this function required the addition of a row of ones to a splice of the `sorted_X` array. In each iteration, the original code would allocate a new array of ones with `numpy.ones()` of the needed length, and concatenate it with the splice.

Instead of doing this in each loop, we decided to create a single long array of ones outside the loop, and pass it to the function so that it wouldn't need to allocate a new array on each iteration. There was a tradeoff in total allocated memory as the previous version didn't require to have a new array that was as long as the entire `X` array, but the performance improvements obtained by avoiding the repeated memory allocations were worth it.

Building on this concept, we decided to try to extract as many repeated operations as possible from the function, so instead of doing the concatenation inside the inner loop, we moved it much further up, which required extensive changes to the code, but allowed to improve overall memory footprint. Before multiple versions of the `sorted_X` array would exist, the normal one and the padded one, which was computed multiple times. Similarly the reshaping of the `sorted_y` array was moved outside of the inner loop.

The same logic was also used further up, in the recursive `mob_fit` function, where several copies of `X_subset` and `y_subset` coexisted. Changes were made to avoid unnecessary copies and reduce both memory footprint and execution time wasted on allocations.

Overall all the changes amounted to a significant performance jump and reduced memory usage. While it is likely hard to see, the following graph in Figure 3.4.1 shows a profiling run of the code after the optimizations passes, and it shows a lighter and more concentrated distribution of time in the lower half of the graph, where most optimizations took place.

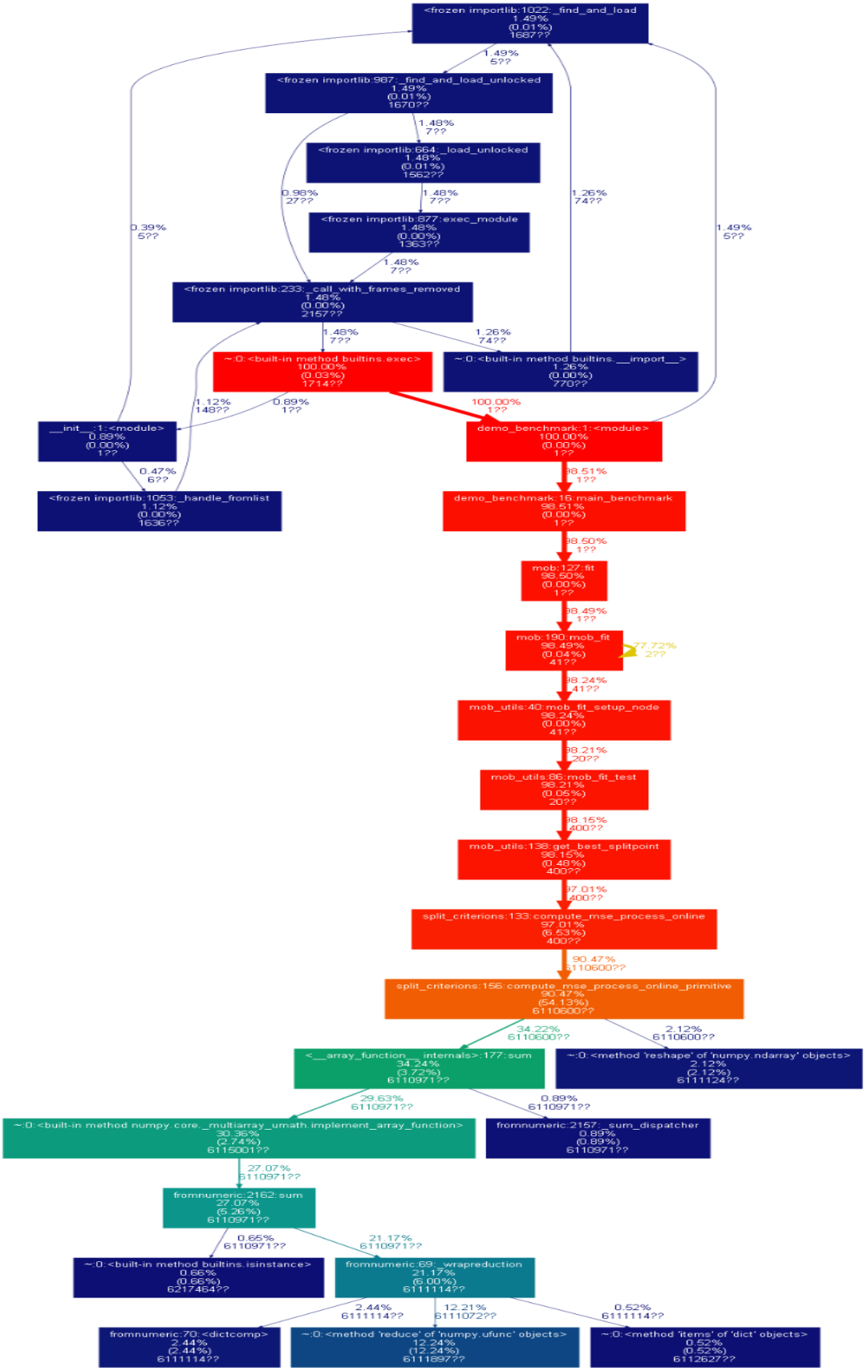


Figure 3.4.1: Example of gprof2dot profiling graph of optimised solution. When compared to the unoptimised version of Figure 3.1.1, *the bottom of the graph is lighter and more operations show up instead at the top, which were previously too small to appear.*

3.5 Design of parallel solutions

After the optimization pass was done, we had code that could be analysed for the design of a parallel solution. The profiling showed that the most time was spent inside the function computing the incremental split-point scores. Going up from there we looked at opportunities for parallelization.

As anticipated in the first chapter, there are two main points in the code where there are independent operations that could be performed simultaneously. The first one is during the recursive generation of the binary tree, where the two children of the currently computed tree branch can be computed at the same time. There is no need for synchronisation at the end of the computation of the two children, so, in theory, further children, if there are any, can also be computed in parallel.

The second place in the code that could allow for parallelism is the computation of the best split point and its score for each dimension of the points in the `X` array. Each dimension represents a different variable of the domain, also called features. The algorithm needs to figure out along which dimension it is best to split the subdomain so that the points on each side present the most similar features. To do so the best split point along all the dimensions is computed along with a score. The score is then used to select the best dimension and with it the best split-point.

The computation of the split-point and score for each dimension is independent from the rest and can be performed in parallel. The level of parallelism is limited by the number of features of the model examined, or, in other words, the number of dimensions of the domain in which the input points reside.

Assuming that both of these points could be parallelized successfully, it would still mean that the total level of parallelism available depends on details of the problem and could potentially be quite limited in some cases. The hope is that those cases would already be quite performant, and wouldn't need the performance improvements brought by parallel computation as much.

Seeing that the more complex part would certainly be the recursive binary tree, development mostly centred around finding the best solution for that frangent. As already mentioned in Chapter 1.4, to simplify matters, rewriting the recursion into an iteration was considered, as any recursive algorithm can always be converted into an iterative one.

This could theoretically allow simpler parallelization through Ray. The reason this wasn't then pursued, was that the iterative version would have needed a central data structure, such as a stack or queue, to keep track of nodes that still need to be computed. The presence of the shared data structure would actually have been a detriment to parallelism due to access contention. To avoid concurrent access to a shared stack and keep each execution stream separate, a recursive like solution would need to be used anyways so the idea was sidelined.

3.5.1 Parallel recursion

The first implemented solution was a parallelization of the recursive function through the use of the `ray.remote` decorator to transform the function into a remote one. This required a few changes and adjustments to the function itself, the most notable one was the need to change the function from a method of the mob class to a standalone function that wouldn't interact with the mob object. The functional programming paradigm [42] offered a good guideline on how to best set up the function to avoid any unwanted side effects. Figure 3.5.1.1 shows a graphical representation of the parallel tree computation.

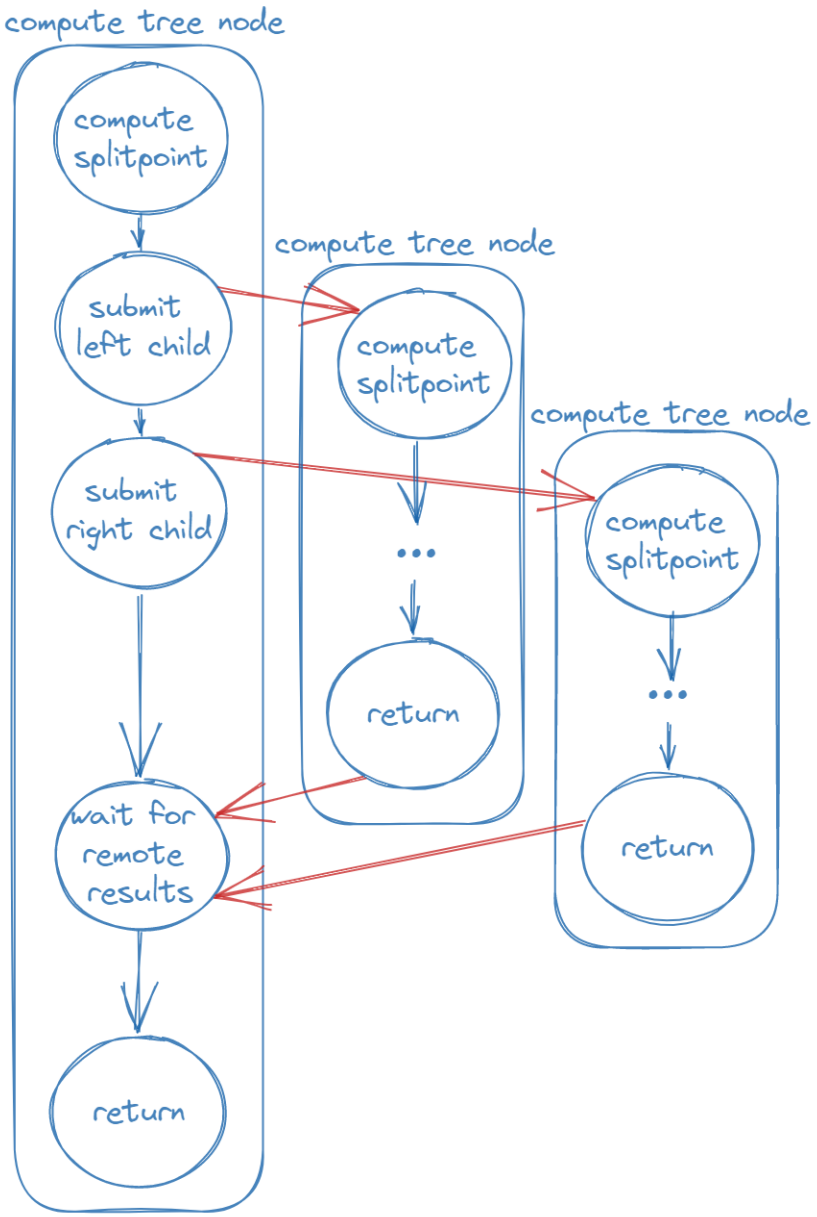


Figure 3.5.1.1: Graphical representation of Ray parallel computation model of the binary tree. Each child node could itself have further children.

One obstacle was the incremental ID given to each node of the tree, this needed to be unique for each node, so it required some synchronisation across the parallel tasks. For this purpose an `IdManager` actor class was created. Each task could call a method of the actor to obtain a new incremental ID. Since each actor executes on a single thread, each method invocation is guaranteed to be executed sequentially.

A similar solution was implemented for the progress bar, where an actor was used to make the single progress bar object accessible from all the tasks, to update the internal counter of classified points. Some more effort was required, as the progress bar needed to be located on the driver machine, so the actual progress bar object could not be located on an actor. An event was used to notify a waiting `async` function [43] on the driver each time there was progress to be marked.

Overall this parallel version didn't require too many changes, and the ones that were made weren't too intrusive, but there was a problem with resource usage. As the following diagram showcases, for each node of the tree a recursive remote task is invoked. Each node might then have to invoke further child remote functions and wait with `ray.get()` for them to complete to obtain the results before returning. This basically means that until a branch doesn't reach a leaf, the entire call stack of recursive tasks that came before is still active and their used resources can't be freed. This behaviour is portrayed in Figure 3.5.1.2.

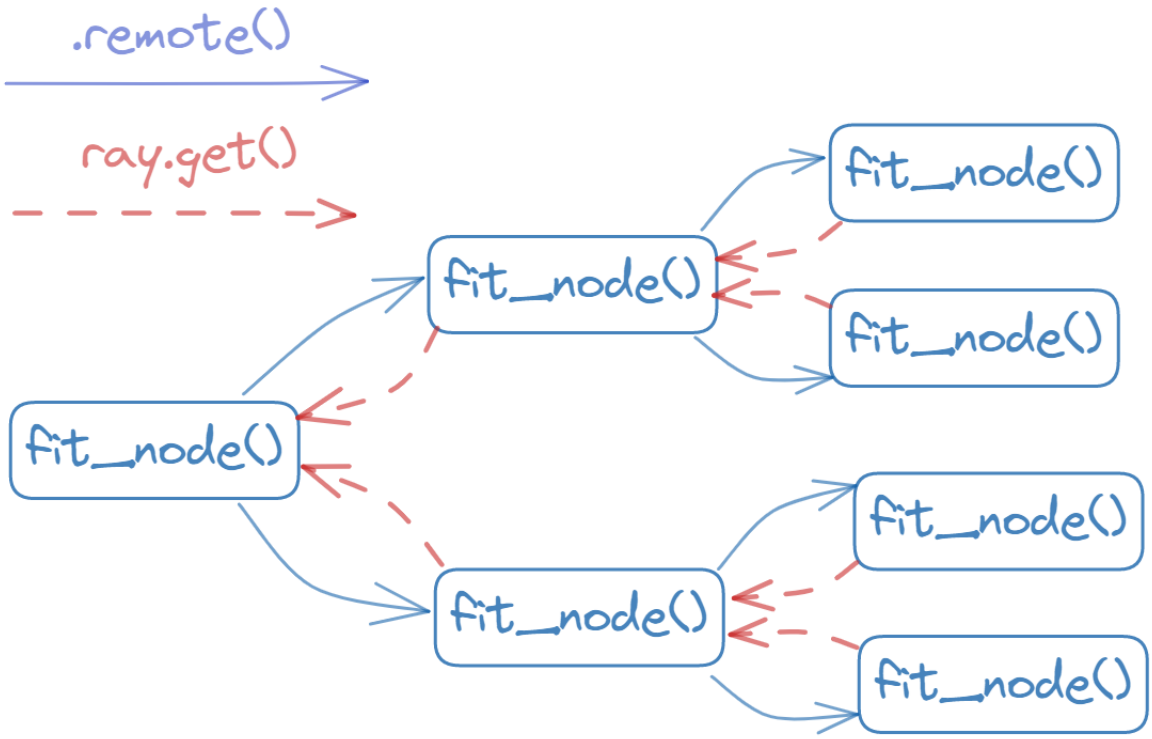


Figure 3.5.1.2: Ray parallel recursive call graph structure with each task waiting on returns of child tasks.

This has bigger implications due to the recursive function being a remote task. As mentioned before, each task, once scheduled, is assigned to a worker. The total number of workers is supposed to be limited and in the same number as the number of logical cores present in the Ray cluster. With this solution configuration though, a task that is blocking waiting on the `ray.get()` call is also keeping the worker blocked. As explained in the previous chapter, Ray avoids deadlocks in this situation by instantiating new workers, so that they may handle the still pending tasks. But, in this case, each new task blocks a worker and doesn't release it for potentially a long time, until both child nodes finish computing both their entire subtrees.

This situation resulted in a lot of resources being consumed. To avoid the problem it was necessary to not use blocking `ray.get()` calls inside the recursive task. Fortunately, the use case allowed us to change to code to avoid waiting on the results. The idea was for each task to submit the asynchronous children remote tasks but not wait on them and instead return immediately with a node containing unresolved remote references of the children instead of already computed references of child nodes, as demonstrated in Figure 3.5.1.3.

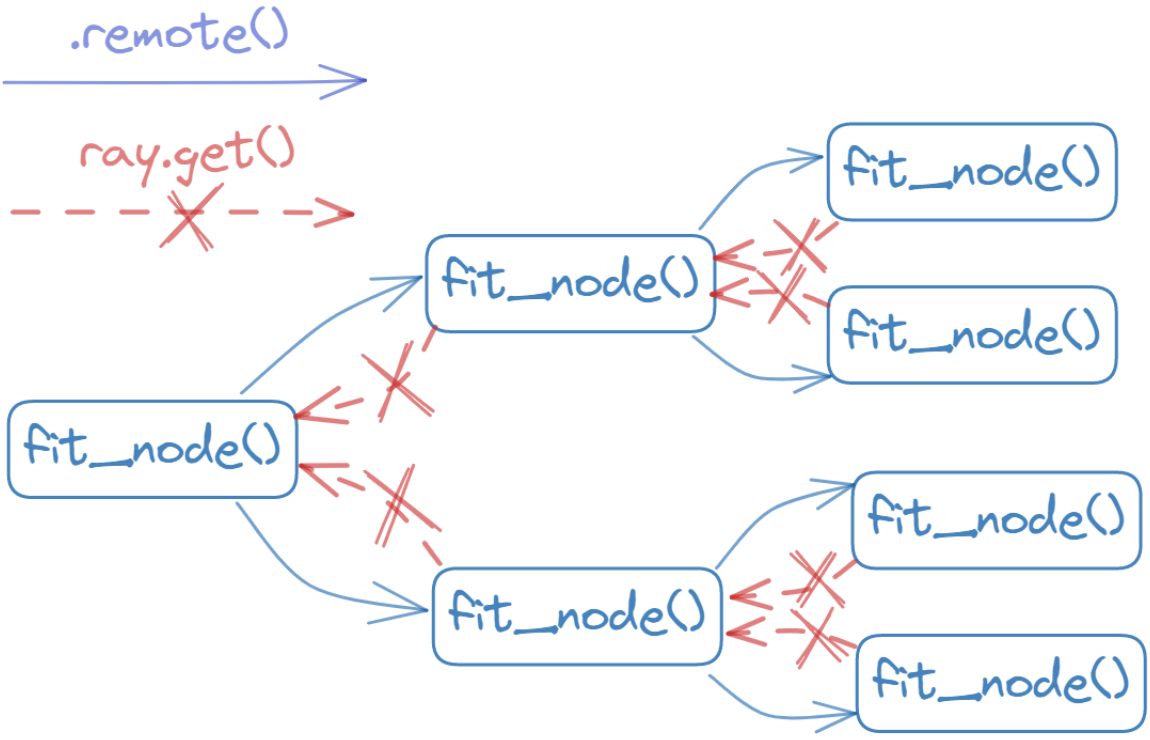


Figure 3.5.1.3: Ray parallel recursive call graph structure with tasks not waiting on child task returns. Each task performs its computations, invokes the child tasks and then ends without waiting.

It was then necessary to resolve the generated tree of references. To best do so, a recursive asynchronous function was implemented. This function took the reference of the head node and for each child node, awaited asynchronously on the results. The use of an asynchronous function here allowed for deeper levels of the node to be resolved concurrently even while

other parts of the tree were still waiting for results to arrive. To achieve this, the `asyncio` Python standard library [43] was used.

`Asyncio` works through the use of an event loop where new tasks are queued. The await operation allows the code to suspend the execution of the current function while some otherwise blocking operation is taking place, the continuation to be scheduled on the loop once it is ready, and instead proceed with the execution of other events present on the queue. Figure 3.5.1.4 shows a graphical representation of the async execution model.

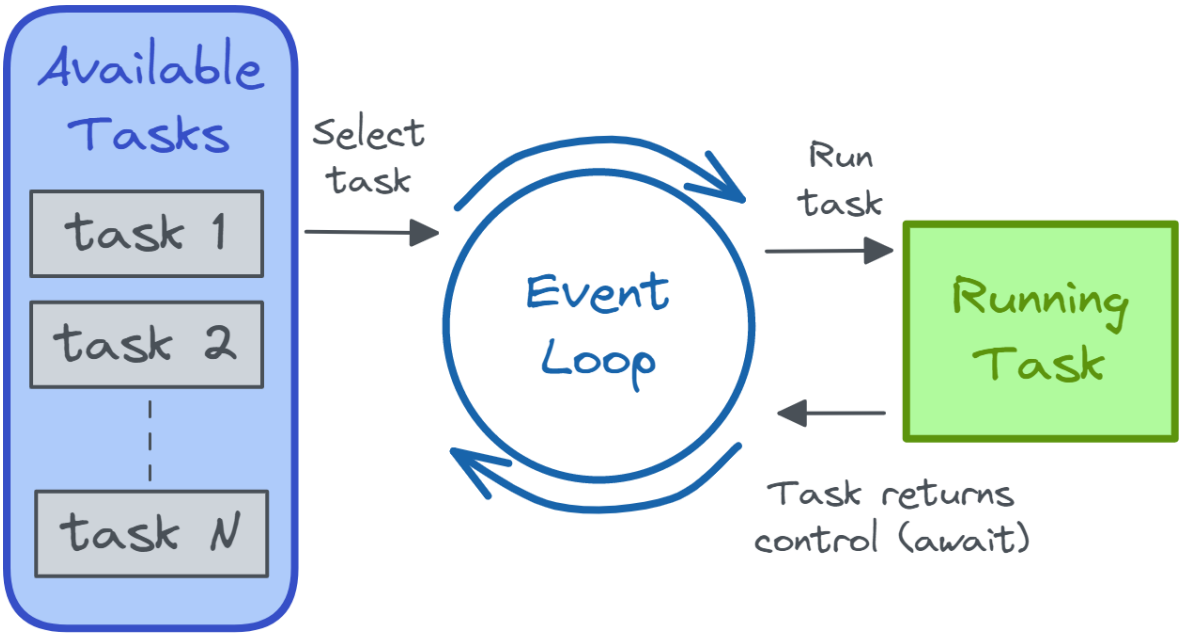


Figure 3.5.1.4: Execution model of async tasks through event loop.

In this case the otherwise blocking operation is the `ray.get()` operation, and the events that get enqueued are the results of the `ray.get()` calls from the various nodes in the binary tree. This way even if one of the branches is still stuck in computation, the asynchronous function can still resolve other parts of the tree while it waits.

With this new solution, each task generated two child tasks (except in the case of a leaf node) and then concluded, avoiding the overcrowding of workers and an `async` function is used to resolve the tree of remote references efficiently and concurrently. The maximum amount of parallelism obtainable depends on the depth of the tree and how balanced it is.

The more the tree is balanced and the more it is deep, the higher the obtained parallelism. The depth might not matter though, if the tree ends up being highly unbalanced, such that one child node always ends up requiring much more computation time than the other, the effective parallelism can be greatly reduced, or even nullified.

3.5.2 Parallel split-point calculation along domain dimensions

Considering the potential low parallelism offered by the parallelization of the recursion, we changed our focus on the parallelization of the computation of the best split-point along each of the domain dimensions. The level of parallelism achieved corresponds exactly to the number of variables of the domain, so to the number of features of the examined function/model. This was simpler in concept and execution compared to the parallelization of the recursion.

The implementation didn't require many changes apart from annotating the function responsible for the split-point computation for each dimension with the `ray.remote()` decorator and changing the loop where it was invoked.

The original version:

```
for id_var in range(num_variables):
        # get score and splitpoint for the best partition on the given
          variable
        splitpoint, score = get_best_splitpoint(id_var, X, y, minsplit, beta,
                              method=method, cov_matrix=J,
                              aggregation_function=aggregation_function)
        splitpoints[id_var] = splitpoint
        scores[id_var] = score
```

The new version:

```
    # for each variable compute the best splitpoint in parallel
    result_refs = [get_best_splitpoint.remote(id_var, X, y, minsplit, beta,
                   method=method, cov_matrix=J,
                   aggregation_function=aggregation_function) for id_var in
                   range(num_variables)]

    # collect the results [list of tuples(splitpoint, score)]
    result_list = ray.get(result_refs)

    # remove references to remote data to allow garbage collection
    result_refs = None

    for id_var in range(num_variables):
        splitpoints[id_var] = result_list[id_var][0]
        scores[id_var] = result_list[id_var][1]
```

The new version uses a list comprehension to submit the parallel `get_best_splitpoint` remote calls and so obtain a list of references. The references are resolved with a `ray.get()` call, which waits for all the computations to be complete before extracting the actual split-points and scores.

This solution though had an issue with memory usage, as for each domain dimension, a copy of the entire `X` and `y` arrays was created and placed on the Ray shared object storage. To avoid this problem, two calls to `ray.put()` were used to preemptively place the two large arrays on the storage and obtain a reference to them. The references were then passed to the remote functions to avoid duplicating the array.

```
# preemptively send data on the cluster to avoid sending it for each
  variable
X_ref = ray.put(X)
y_ref = ray.put(y)

# for each variable compute the best splitpoint in parallel
result_refs = [get_best_splitpoint.remote(id_var, X_ref, y_ref, minsplit,
               beta, method=method, cov_matrix=J,
               aggregation_function=aggregation_function) for id_var in
               range(num_variables)]
```

This solution is simple and efficient, but the level of parallelism can be limited by a low number of features, especially considering larger clusters, where the number of CPUs or parallel workers can be very high. Effectively leaving resources unused.

To obtain the highest level of possible parallelism, an attempt to combine the two solutions was made.

3.5.3 Combined nested parallel solution

This version was implemented by merging the code changes already done for the previous two versions and resolving any conflicts. The idea was to generate a new task to handle the computation of each child tree node, each of which would then itself spawn further tasks to compute the best split-point in parallel.

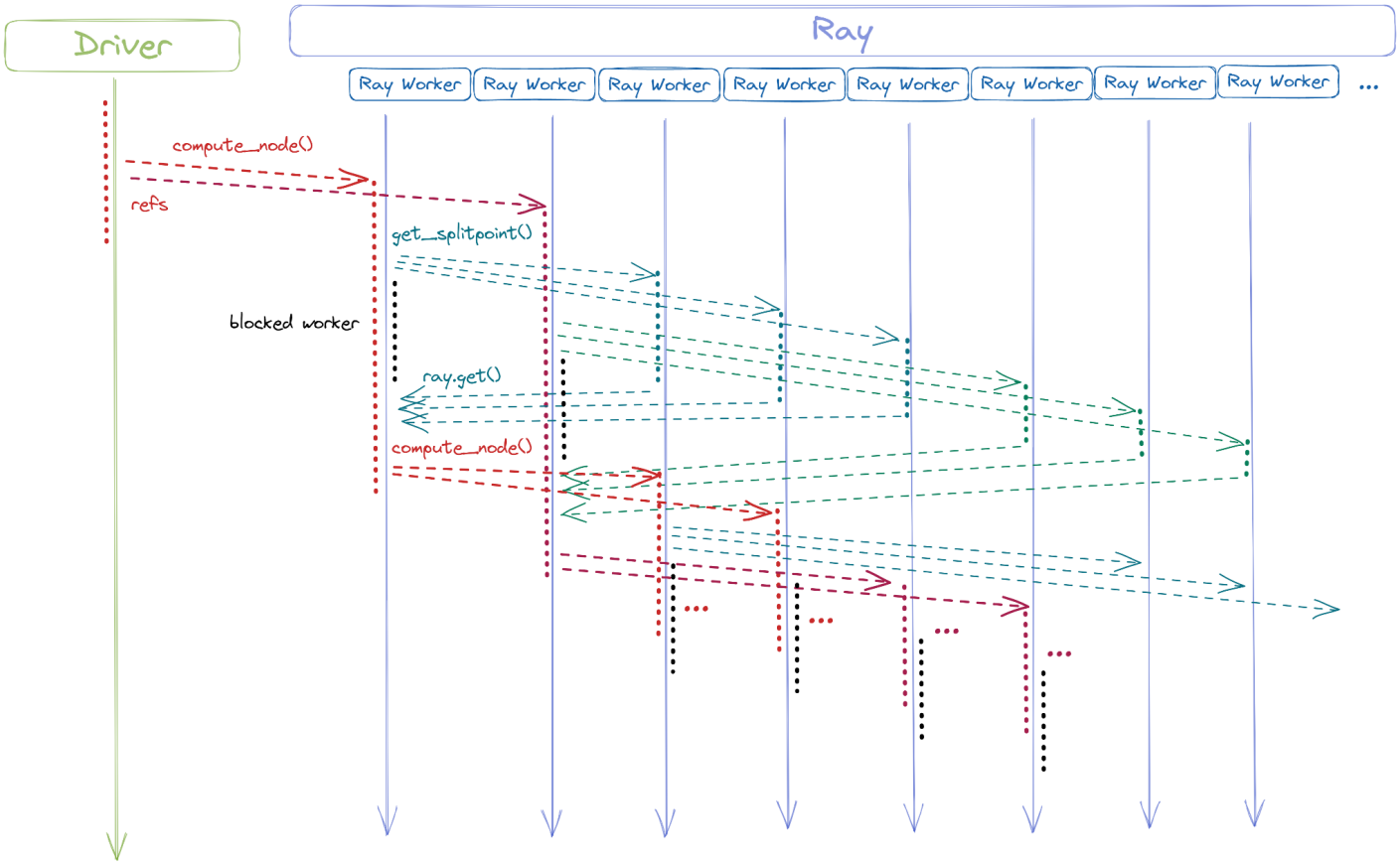


Figure 3.5.3.1: Execution graph of Ray combined parallel solution of both binary tree and split-point computation. In red tasks computing nodes of the recursive tree, while in green tasks computing the best split-points along the various domain dimensions.

In Figure 3.5.3.1 it is possible to observe the rough behaviour of the solution in the case of domain dimension of 3, so with 3 different variables along which different split-points need to be computed and compared. The computation of each node is a task that itself splits into 3 further tasks and blocks on the results with `ray.get()`. After getting the results, it further splits into two other tasks, one for each child node. This time though the child tasks are not waited on, as the remote reference to the results is instead stored in the tree. An `async` function will later resolve the tree from all the distributed references.

While this solution allows to combine the potential parallelism of both previous solutions, the nested nature of the resulting parallelism is problematic as it results in blocked workers waiting on results, which we know from our previous tests to be an undesirable solution as it results in high resource usage.

Testing this solution with problems with bigger input sizes, and so bigger memory and resource usage, we were able to observe several instances of the computation failing to complete due to timeouts, out of memory errors and other similar issues related to overuse of

resources. The performance improvements are present as will be illustrated in the next chapter, but at the cost of stability and resource overuse.

The problem in this case is the fact that to proceed with the computation of the node, the task needs to have the result of the parallel split-point computation to compare them and select the best one and then divide the remaining domain into the two child domains before generating the child tasks to compute the two children. This means that the `compute_node` tasks will block during the parallel computation of the split-points, and there isn't a workaround this time to avoid waiting on the results like what was done in the recursion parallelization. The result is needed immediately.

Looking at Ray examples to see whether any solution was proposed for this kind of problem and one of the options that could work was the use of chained remote functions. Remote functions can accept both normal objects and object references as inputs. If a remote reference object that isn't yet available is passed to a remote function, that function will only be scheduled once the remote reference is ready. With this method subsequent operations can be queued without the need for `ray.get()` calls, which we were trying to avoid.

The problem with this approach is that to completely avoid any `ray.get()` call, the entire code that followed the parallel split-point computation would need to be converted into a remote function, a sort of continuation. Converting the code though posed many challenges, specifically due to control flow. The split-point computation is located a few functions deeper than the top level node computation task level. Each of those functions has potential for early returns depending on preconditions and postconditions checks.

The conversion of this complex control flow into two top level functions, where the first one started the remote computation and the second one worked as a continuation, basically required rewriting the entire algorithm. To correctly handle all the stop cases (where the current node would be classified as a leaf node), the structure of the algorithm would need to change a lot. A lot of data would also need to be moved around between the various tasks.

The return value of the `compute_node()` tasks is a node reference. In the case of a continuation task, it would require access to the node to be able to change its attributes, that would then require the continuation to also have the node as the return value to allow the original caller to have a reference to the correctly updated node object. Since we can't use `ray.get()` after the continuation function, what actually gets returned by the `compute_node` task is a reference to a remote reference, two levels of indirections.

The solution could have been implemented but it would have required major rewriting of the existing algorithm structure to accommodate the requirements of the parallel framework. Something that we wanted to avoid. The fewer changes required to the algorithm structure the better to avoid ending up with complex code structure that makes the understanding of the actual algorithm function harder.

The problem could be observed better using the Ray `timeline` command, which generates a json file containing the execution information of the job. This file can be examined using the *chrome://tracing* utility.

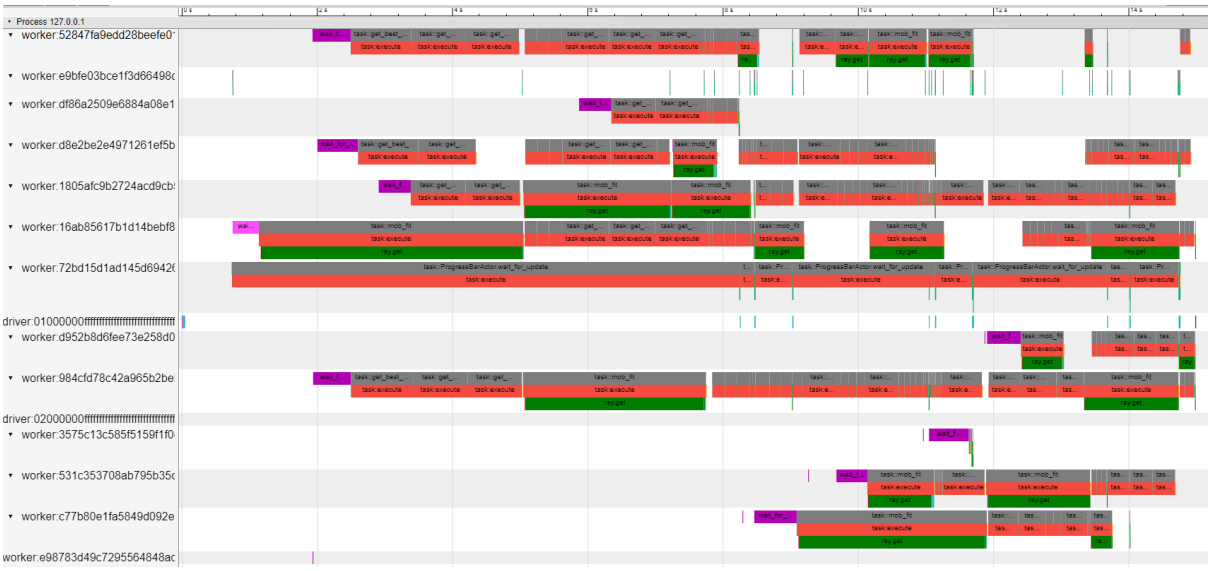


Figure 3.5.3.2: Ray timeline of the combined parallel solution executing on a 4 core limited cluster. Each horizontal line indicates a worker and the coloured sections indicate task executions. In purple is the wait_for_function initialization overhead.

While from Figure 3.5.3.2 it might be difficult to see, analysis of the graph allowed us to observe how, even though the Ray cluster was created with the number of cpus set to 4 on a 8 core 16 threads machine, the execution still instantiated and executed more workers than that. Some are valid, in particular the driver and actor processes, which don't count towards the supposed worker limit. This still leaves us with 10 workers used instead of 4.

Another interesting fact that can be observed is the presence of pretty significant initial delay before the first utilisation of each worker, shown in purple in the graph. This delay is tagged by Ray as `wait_for_function`, and could take from 200ms to 600ms of execution time. Investigation in the Ray code revealed that this tag indicates the setup operation on the worker to obtain the code and dependencies (files, modules, etc) required to run said code on the new process. This only needs to be executed once per worker process, as successive executions of the same tasks on the worker would already have the necessary code and dependencies.

The presence of this additional delay, makes the `ray.get()` problem worse, as each new worker that is instantiated to make up for a blocked one needs to first get the code and dependencies, adding overhead to the total execution time. From the figure it is possible to see how some workers were instantiated after the halfway mark of the total execution time and still needed to `wait_for_funciton` to be executed. This makes for a graph featuring

many holes and empty spaces. The available resources are not utilised to their full potential and instead computation resources are wasted.

3.5.4 Considered optimizations: Numba and others

The `wait_for_function` overhead that was initially registered was actually higher and could vary from 600ms to 1.5s or even more. These kinds of added delays seemed unjustified when the execution time of the task could be much shorter than that. This overhead warranted an investigation which revealed as mentioned before that the `wait_for_funciton` label indicates the time necessary for a worker to obtain everything necessary to execute the code of the remote task that was scheduled on it.

While transferring the code isn't usually that slow or expensive, a discussion with a Ray team member [44] revealed that the most expensive part is actually the initialization of all the modules that the code requires. If a remote function is defined in a file which contains a lot of imports, all those modules will need to be reimported and thus reinitialized on the worker.

To minimise the `wait_for_function` overhead, the best solution for now seems to be to reduce the number of imports mentioned in the code files containing the definitions of the remote functions. The Ray team mentioned being in the progress of developing some improvements on this front [44], to reduce worker initialization overhead for version 3.0. Some restructuring and cleanup of the code allowed to reduce the overhead to a more manageable 400~500ms.

To further improve performance, the use of Numba [15] was attempted in conjunction with the parallel solutions. The idea was to compile the most computationally intensive function, and have the remote workers utilise the compiled version for reduced overall execution time. This function is the one that performs all the maths matrix and array operations that allow the computation of the best split-point along a given variable (domain dimension).

The easiest way to achieve this was by using the `numba.jit` decorator over the function. This required some adjustments since Numba is only able to compile a subset of python and numpy operations. So the most internal part of the computation was extracted into a separate function and the Numba just-in-time decorator was added to it. The first time each worker executed the function, the Numba compiler would invoke the just-in-time compiler to obtain the native version and execute that one on subsequent task executions.

This resulted in some slight performance improvements with higher loads, but with small problem sizes the added overhead actually made the Numba version slower than the normal parallel version. The reason being that the compilation step added more overhead than the time saved by the shorter execution time of subsequent calls. An important difference of using Numba in a distributed environment is the fact that the function needs to be recompiled multiple times, one for each worker that is tasked to execute the graph. In a single threaded case in comparison, the compilation only needs to happen once.

To try to remove the overhead, different approaches were tested. The first idea was to make use of Numba eager compilation. This approach was supposed to potentially offer higher performance by avoiding the on-line overhead of the just-in-time compilation and by using more typing information to generate a more optimised result. Some more changes were required as Numba needs to have accurate and complete typing information for the eagerly compiled function. After all the input parameters and return values types were defined in detail, and some changes were made to fix some issues with read-only arrays and how those arrays where stored in memory.

The hope with the eager version was to perform the compilation only once, and have all the workers use the already compiled version, but this didn't work as desired in practice. The function was unavoidably recompiled on each worker when the module containing the function code was initialised, removing any advantage over the previous test. In this scenario, the initial `wait_for_function` overhead on worker initialization could exceed 5 seconds of execution time.

Another attempt was made by manually compiling the function using Numba tools and passing the compiled function object as a parameter to the remote tasks, to avoid the recompilation. This resulted in even worse performance, as each remote function call suffered from severe parameter serialisation and deserialisation overhead, as the compiled function was passed around.

Overall, Numba could offer some performance benefits with large enough problem sizes, but in general it introduced big enough overhead to be detrimental. Furthermore, the changes required for optimal performance through eager compilation negated the ease of use advantages initially considered.

In the end, the use of Numba was discarded to avoid excessive overhead on small problems and reduce complexity.

Another focus for optimization was memory usage. Due to the high resource usage observed with the combined parallel solution, attempts were made to reduce the total memory usage of the application and of each worker. Some of the memory improvements were indeed achieved as detailed previously in this chapter under the optimizations section. One change though that was considered was the distribution of the entire array of input points at the very start of the computation, so that each worker would have access to it from any machine.

The goal of this idea was to cut down on the data that would need to be passed around between tasks, and reduce the overall memory usage by avoiding copying the data and instead using references or slices of the single distributed array.

Some investigation revealed the idea infeasible due to the need of sorting the input points along the different dimensions for the computation of the split-points as described in the

`get_best_splitpoint()` pseudocode in Chapter 1.3. There is no way to efficiently handle resorted references to an array. Each time the array would need to be accessed a sorted copy would either need to be reallocated or several layers of indirection would need to be passed, negating any performance or memory benefits.

The final version used a single subset copy of the input array for each node, which would be put on the shared object store before the parallel tasks computing the best split points along the various domain dimensions were submitted. Each of those tasks would then make a local copy, sorted along a specific variable, and use that for all the internal computations.

Let's consider the memory usage on a single machine cluster with an input problem featuring an `X` array of 1GB, containing sample points with 8 variables. On the second iteration, so at the computation of the first two children, the driver process would have 1GB allocated for the original array and another 1GB divided in subset copies, one for each tree node currently being computed. At the moment of the parallel split-point computation, each parallel task needs to make a copy of the subset to sort along one of the 8 dimensions, so 8 copies for each subset. Given that the sum of the subsets would amount to 1GB, 8 copies of each subset would also amount to a total of 8GB. This example is illustrated in Figure 3.5.4.1.

So considering both the driver process memory usage of 2GB and the worker processes total memory usage of 8GB, the overall memory usage on the machine would amount to 10GB, which would decrease as computation progressed, as leaf nodes would be encountered and subsets of the samples would be excluded from further computations. This isn't ideal but it is also unavoidable without sacrificing any performance gained through parallelism in the first place.

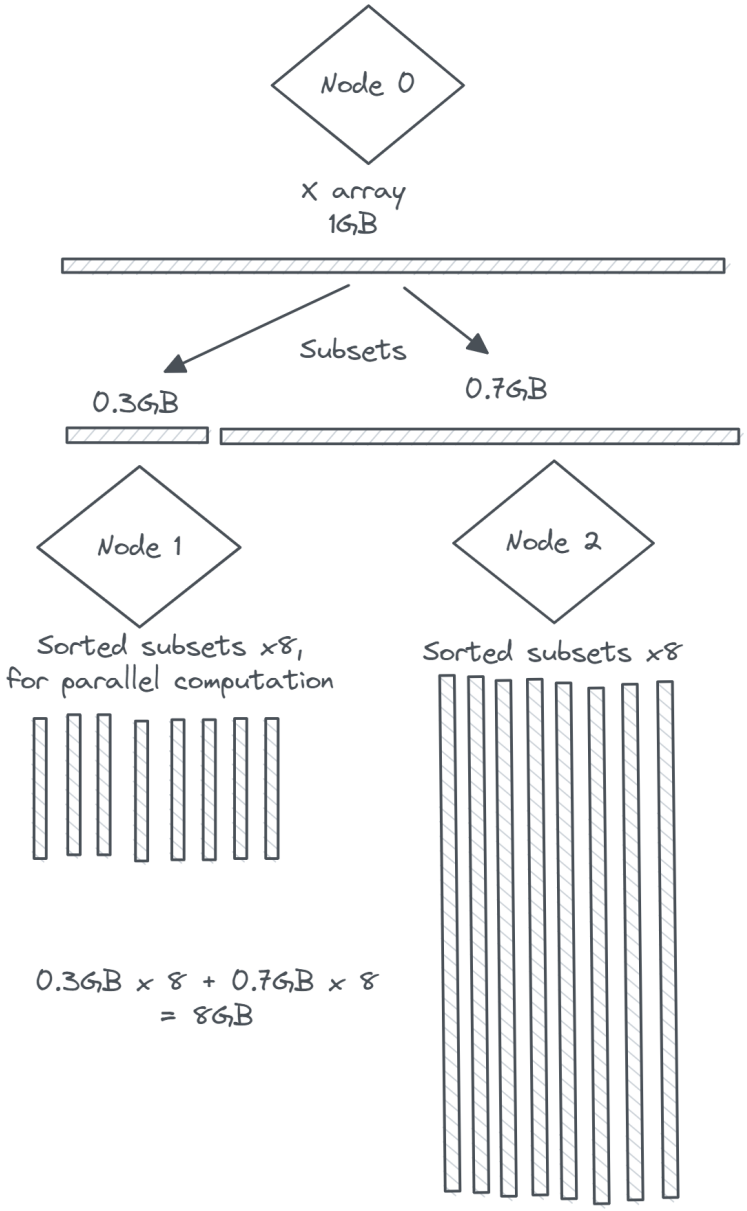


Figure 3.5.4.1: Example of memory utilisation of combined parallel solution on a single machine at second iteration, with 1GB ***X*** *array and 8 domain variables. The subsets are replicated 8 times for each tree node, so that each parallel task can sort its copy along one of the 8 dimensions. Total memory usage is 10GB.*

3.5.5 Asynchronous concurrency for recursive tree computation

As the results obtained from the combined parallel implementation of both the recursion and the split-point computation weren't satisfactory either from resource usage or stability point of view, further solutions were researched.

The main issue to be resolved was the blocking nature of the nested parallelism, which resulted in the spawn of multiple workers. The best thing would be to somehow avoid blocking the workers, and instead reuse them to execute the tasks that are blocking the current one. Sadly this isn't currently possible. While Ray took great care to be asynchronous

in its remote calls, that alone doesn't solve the problem, especially since there is no way for a developer to have a worker expressively handle other tasks while it is waiting on something.

With the building blocks offered by Ray, it should be possible to create a framework on top of it that could offer the wanted behaviour, but the time needed to implement it and the performance hit derived from the use of Python would make the effort not worth it. More discussions about the ideal solutions are better outlined in Chapter 4.3.

Returning to our particular use case, one observation that we could make was that the recursive parallel version wasn't often able to exceed 3 or 4 parallel threads of execution. This meant that, considering the stability and resource usage issues, forgoing the recursion parallelisation and only relying on the parallel split-point computation was an option to be regarded.

To not lose out on the performance to be had on the recursive tree side though, the option to use concurrency instead of parallelism was considered. In this case, the two terms are used with the distinction that parallelism implies multiple tasks that are actually running at the same time, while concurrency is a more general term that only means that multiple tasks are executed during the same time period, with no particular order. Concurrency thus encompasses also cases of multiple tasks executing on the same core, by alternating access (time sharing). Figure 3.5.5.1 shows the difference between the two concepts with an example.

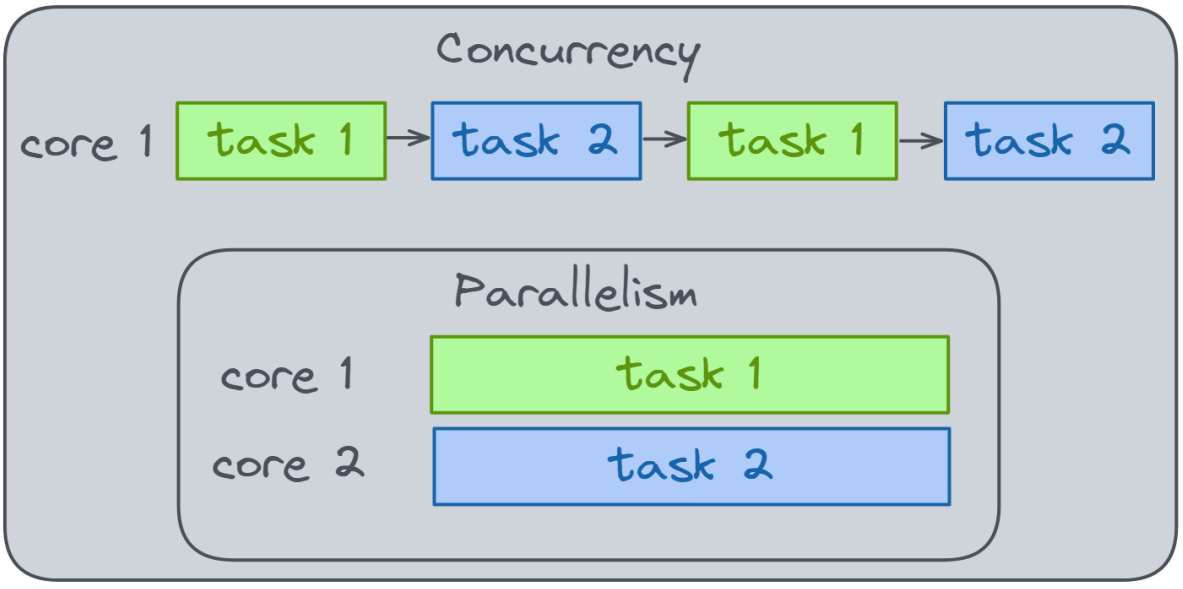


Figure 3.5.5.1: Concurrency vs Parallelism.

A notable example of successful and beneficial concurrency contrasted to parallelism is the use of async execution inside Node JS [45] based web servers instead of the more traditional thread based ones. In the latter cases, each incoming request would spawn or use a thread from a pool to be handled. Each thread could get blocked waiting on external requests or IO

access. In case of asynchronous execution instead, new requests are managed by an event loop, where each request handler could interrupt its execution when having to wait on non-compute operations, leaving the CPU free to handle other requests at the same time. CPU utilisation with this method was much improved when compared with the thread counterpart.

The asynchronous programming paradigm has become very popular in recent years, with many programming languages adding tools and libraries to support it or even being completely built around the concept. Python is not excluded and it offers tools for asynchronous programming through the already mentioned `asyncio` standard library [43].

With `asyncio` it is possible to set up an event loop and define functions as `async`, so that they can be suspended and resumed. Such functions are also referred to as coroutines in other programming languages. The idea is to avoid blocking by instead continuously handling events which in turn are either short or would need to block. Traditionally async computation is more suited for IO bound operations, and not CPU bound computation. The reason is that a long computation task would hold the CPU occupied for a long time without giving it the opportunity to handle other events or tasks in the loop, hurting their latency.

Our use case is practically exclusively computation, with no IO or external requests. This is without considering the remote tasks themselves. If a remote task is considered as an external request over which we would like to avoid blocking, then `asyncio` could still be very beneficial. As mentioned before, the ideal would be for tasks to be able to be non blocking themselves when waiting on a `ray.get()`, but while it is possible to set up an event loop inside a worker, there is no way to access the event loop from the outside to feed it with the tasks that need to be completed to resume the initial task execution. This could instead be done with an async actor, which allows methods to behave like events that are placed on the loop.

An async actor could have an async method that computes a tree node, the method would be recursive and would avoid blocking on either the parallel split-point computation or the recursive calls. With this system, when the actor is handling the first child nodes, it could submit all the remote parallel tasks to compute the best split-point for the left child, and while awaiting the results, submit the parallel tasks for the computation of the split-point of the right child node.

Initially this was the solution that was devised, with the idea that multiple instances of the actor could be deployed to alleviate traffic towards it or load of operations. The problem that was quickly encountered was that there was no efficient or effective way to subdivide or balance the work between the actors.

Instead of using an actor then, the idea was changed and simplified to use the driver process as the async agent. The initial Python process where the script was first executed would be the one hosting the asynchronous event loop. This simplified the complexity of the proposed

solution. The implementation required the change of a series of functions into `async` ones and the substitution of all `ray.get()` calls with `asyncio` equivalents that are awaitable.

Luckily, all Ray remote object references extend the asyncio interfaces and are thus awaitable, which makes the entire thing possible in the first place. The functions that were modified were the recursive tree node computation one, and a series of deeper ones that contained the parallelization point where the best split-point along each domain dimension is computed separately by Ray tasks. Figure 3..5.5.2 illustrates the general execution model of the designed solution.

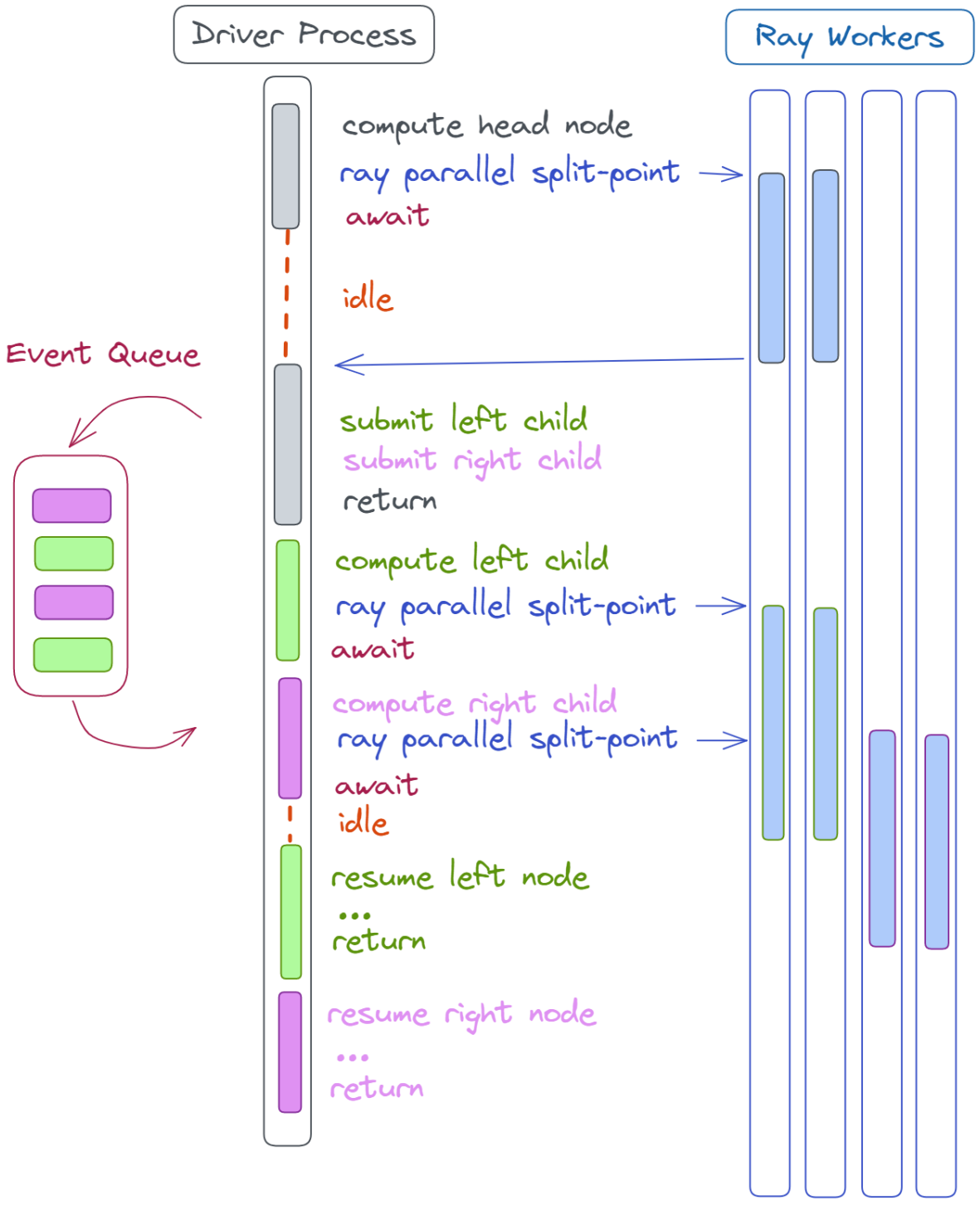


Figure 3.5.5.2: Async parallel solution execution model. Each filled rectangle represents an async task. The await operation puts the current task on the event queue and looks for another task that is ready to execute instead. If no ready tasks are available, the driver remains idle until a task becomes ready.

The only reason why this solution was contemplated was the observation that, comparatively, the execution time of the remote tasks computing the split-points was much longer than the execution time of all the other operations around it. This meant that the other operations,

while still being CPU bound, could basically be considered trivial in comparison. What the async system is actually enabling is the possibility of beginning the parallel computation of the best split-points for multiple nodes at the same time and efficiently waiting on the results.

Figures 3.5.5.3 shows the Ray timeline execution graph of the async solution on Ray cluster artificially limited to 4 cores. Figure 3.5.5.4 shows the timeline graph of the combined parallel solution.

Figure: 3.5.5.3: Ray timeline of async parallel solution executing on 4 cores limited machine. The number remains equal to 4. Each line appears to be almost completely filled, indicating high efficiency in the usage of those workers.

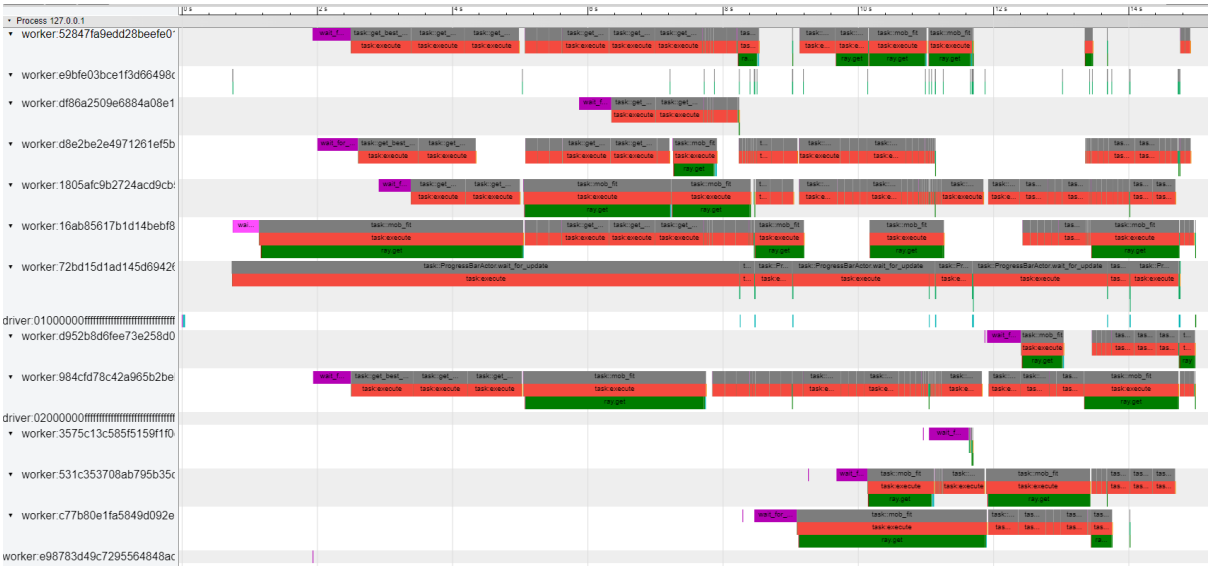


Figure: 3.5.5.4: Comparative Ray timeline of combined parallel solution executing the same workload on a 4 cores limited machine. Number of workers exceeds 4 and each worker exhibits many idle moments with no work being done. Much less resource efficient.

It can clearly be seen how the async version actually respects the resource limitations imposed and is much more efficient in the use of the computation resources. Only 4 workers are instantiated, using less memory and avoiding additional worker setup delays during the middle of the computation.

So in this solution, async concurrency through the `asyncio` library was used to handle the binary recursive tree, and Ray remote tasks were used to parallelize the computation of the

best split-point along different domain dimensions. The two methods allowed for better stability than the version with Ray parallelization for the recursion as well.

3.5.6 Parallelizing the concurrent Asynchronous code

One of the considerations that were made during the development of the async solution was that the driver process could become a bottleneck since the `async` functions did contain CPU bound computations around the invocations of remote tasks. The worry was that the event loop would introduce too much overhead because it would be busy with computing the results of a big operation and not be able to handle other events in the meantime.

To avoid this potential issue, two main options were considered. The first one was to try to move the rest of the CPU bound operations to remote tasks as well, simply to not keep the loop busy with them. The issue with this approach was identifying exactly which operations were worth moving to a Ray task. We know that remote tasks have inherent overhead and thus they are worth only for long enough workloads, otherwise the latency that derived from the async loop being busy would just be converted into Ray remote tasks latency.

Analysis of the code surrounding the existing remote invocations didn't highlight any sections with particularly computationally heavy code. At least not in cases with normal input sizes. The computation could potentially become longer when dealing with a larger array of sample points. This though could mean that, to avoid unnecessary overhead, the remote tasks would need to be dispatched only when the problem size was empirically big enough.

Implementing such a solution would not only add complexity to the code but also require empirical testing to identify ideal load values around which the use of dedicated remote tasks could be worth it. The possibility existed that the testing would reveal no advantage at all in using such an approach. To avoid wasting effort in solutions that would only potentially be useful in edge cases anyways, the idea was discarded.

The second option that was considered, was to instead parallelize the async loop itself. Instead of having a single thread of execution handling events, a pool of executors would handle the incoming events in parallel. Python offers an interface called `Executor` [46] through which it is possible to execute `async` tasks concurrently. The Python standard library offers two implementations: `ProcessPoolExecutor` and `ThreadPoolExecutor`.

The idea would be to execute the `async` functions inside one of these executors to allow multiple `async` functions to be computing at the same time, even in the case of longer CPU bound operations. The presence of a pool would allow new events to be handled even while others are still being processed.

To avoid incurring in the Python threads parallelism limitations, the use of `ProcessPoolExecutor` was first attempted in vain. The issue that was quickly encountered was the fact that each new process in the pool would need to initialise Ray, which devolved in

multiple Python processes trying to generate their own cluster or all trying to become the driver process in case of an existing multi machine cluster. A quick query on the Ray Forums showed that this is a known limitation of Ray at the moment and that the use of Python multiprocessing is advised against in conjunction with Ray [47].

The same solution was attempted with the `ThreadPoolExecutor` instead, but another issue appeared. The computation of the tree nodes is recursive, which means that each function handling the computation of one node could need to spawn two further child functions. The initial idea was to make each of these functions be executed inside the thread pool, but this revealed itself to be impossible. The `ThreadPoolExecutor` doesn't allow scheduling of `async` functions inside of it, rather it only allows normal functions to be run. This made it impossible or very hard to make the recursive structure work as intended.

A possible workaround could have been that of extracting the bulk of the operations inside the recursive function and offloading them to the threadpool, while the async recursion would solely remain on the main thread and event loop. This would have meant that the only parallelizable part would have been the extracted portion of the recursive function, and even then with the known hard limitation of the presence of the GIL. The threadpool executor was meant to be used for blocking IO operations, not CPU bound operations. The process pool executor would have been the tool best suited for that but wasn't viable in our case.

The proposed solution would have made for very confusing code as some parts would have been async, some multithreaded, and some remote Ray tasks, each with their own peculiarities and limitations. Some attempts at implementations were made but early testing showed that the performance obtained with this solution would have been strictly worse than what measured with async solution without multithreading. The absence of any perceivable benefit and the increased code complexity that the solution would have required, made us decide to discard it as an option in favour of the simpler async only based solution.

Chapter 4

Performance tests and analysis of results

Following their implementation, the parallel solutions were tested and compared both on a single machine and on a distributed cluster. In this chapter the results of those tests are presented and analysed.

4.1 Single Machine testing

4.1.1 System Specifications

One of the primary goals of the project was to improve performance of the algorithm even when executed on local single machines. To compare the results and verify performance improvements a desktop machine was used with the specifications outlined in Table 4.1.1.1.

CPU Model	AMD Ryzen 7 3700x
CPU Physical Cores	8
CPU Logical Cores / Threads	16
CPU Frequency	3.6GHz Base - 4.4GHz Boost
Memory	32GB 3200MHz DDR4
Storage	Samsung SSD 970 EVO Plus 500GB
Operating System	Windows 11
Python Version	3.10.9

Table 4.1.1.1: Single machine system specifications.

4.1.2 Benchmark Specifications

For the purpose of measuring performance of the various solutions, a benchmark with various problem sizes was used. The changing variables were `m` and `dim`. The `m` variable is the exponent of 2 that indicates the number of sample points in the `X` array; `dim` on the other hand indicates the number of dimensions of the domain, so in practice the number of dimensions of each of the points contained in the `X` array.

As mentioned before, different sizes of the input array and different numbers of dimensions have a great impact on the execution profile of the algorithm, thus multiple configurations of them were tested.

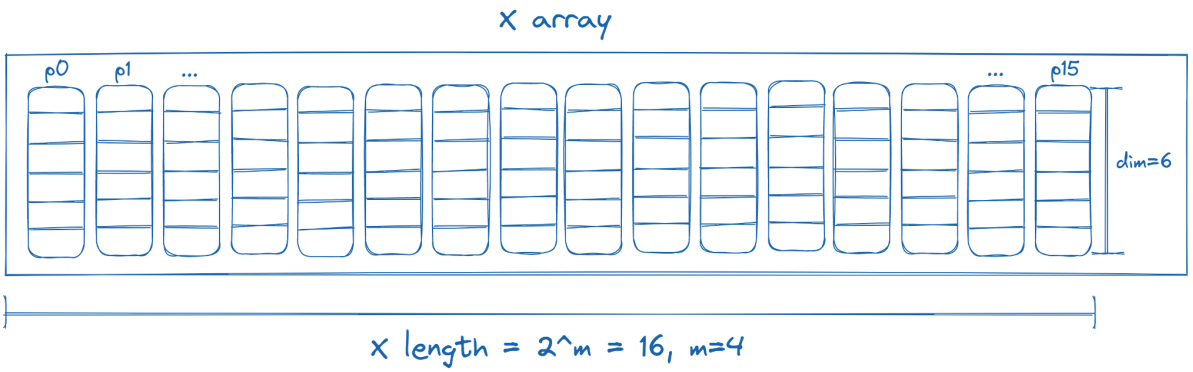


Figure 4.1.2.1: Array of samples `X`*, with* `dim=6` *and* `m=4`.

The total size of the array in bytes depends on both m and dim and can be obtained with the formula: $dim * 2^m * float\_size$. Some graphs will show the input size on the x axis, which is calculated with the formula above without the inclusion of the float size. Figure 4.1.2.1 shows a graphical representation of the `X` array. For the benchmarks, the values of 14, 18, 20 and 22 were used for `m` and the values of 5, 10, 20, 50, 100, 200, 400 were used for `dim`.

`Dim` is a factor that depends on the problem that the user is trying to analyse, and represents the number of features, or variables in that problem. Most problems that are of interest for this algorithm will not have a very high number of features, most likely between 5 and 30. Higher counts of `dim` are included mostly to observe scaling at higher levels of parallelism, but should be considered more as extreme stress test cases rather than realistic usage conditions.

`M` on the other hand is a parameter that can be changed by the user. Higher values of `m` offer higher result granularity and accuracy, at the cost of higher resource utilisation and execution time, while lower values offer faster results at the cost of accuracy. The length, being a power of two, offers optimality for one of the sampling strategies, see Vincenzo's thesis [1] for more information. The length of the array grows exponentially with `m`, so, for example, the size increases by 16 times between `m=18` and `m=22`. Realistic usage values for m are between 15 and 20.

The Firedman1 problem [38][39] with fixed seed was used as the model from which to obtain the sample X points and y results over which the MOB tree was to be fitted. For more information refer to Chapter 3, section 3.3 Measuring performance.

4.1.3 Optimization performance results

Before looking at the parallel solutions, some time was dedicated to measuring the improvements obtained through the profiling and optimising of the sequential code. The optimizations were adopted in all parallel implementations and were used as the baseline for the performance comparisons and speedups.

As mentioned already in Chapter 3.4, the sequential unoptimized version used here actually contains some of the very first optimizations that were implemented, as they were included during refactoring of the code. Using the older version, before the latest changes and refactoring would have been impractical. The total speedup reported in the graphs is consequently a lower estimate of the actual speedup offered by the optimised version.

Due to the code execution being sequential, some of these tests resulted in very long execution times, for some input configurations, prohibitive. For this reason some of the larger combinations were excluded from the testing. Tables 4.1.3.1 and 4.1.3.2 present the testing result data.

m	dim	Input size	timedelta	seconds
14	5	81920	00:00:12	11.986
14	10	163840	00:00:23	23.175
14	20	327680	00:00:47	47.452
18	5	1310720	00:03:13	193.024
18	10	2621440	00:06:20	380.102
18	20	5242880	00:13:06	785.841
20	5	5242880	00:12:30	750.324
20	10	10485760	00:25:09	1508.643
20	20	20971520	00:51:25	3085.144

Table 4.1.3.1: Result data for sequential version without optimizations.

m	dim	timedelta	seconds	speedup	improvement
14	5	00:00:04	4.35	2.76	176%
14	10	00:00:09	8.572	2.7	170%
14	20	00:00:18	17.747	2.67	167%
18	5	00:01:09	69.196	2.79	179%
18	10	00:02:22	141.909	2.68	168%
18	20	00:05:06	305.597	2.57	157%
20	5	00:04:38	278.015	2.7	170%
20	10	00:09:28	568.429	2.65	165%
20	20	00:20:09	1209.057	2.55	155%

Table 4.1.3.2: Result data for sequential optimized version.

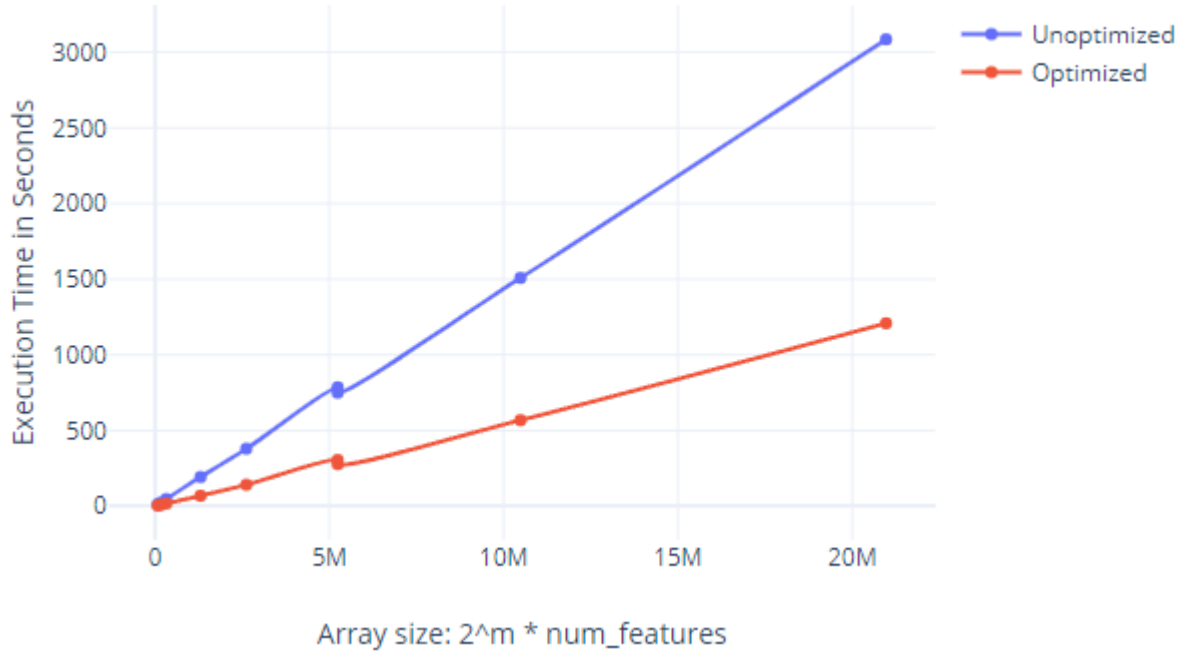


Figure 4.1.3.1: Optimized vs Unoptimized execution time comparison. On the x axis is the size of the input samples array `X`*. Single 8 core / 16 threads machine.*

This graph in Figure 4.1.3.1 uses the total size of the array on the x axis and shows the execution time in seconds. It can be observed that both the optimized and unoptimized sequential versions' execution time appear to have a linear relationship with the array size. The optimised version clearly performs better than the original one. Many of the data points however can't be distinguished very well as the scale compresses them all at the start due to the presence of much larger data points (due to the exponential 2^m). The actual value of the used `m` and `dim` is also obscured.

To better observe the actual relative improvement, a bar graph of the speedup can be used instead. On the x axis the actual `m` and `dim` inputs values are listed. The speedup is defined as the ratio of the original execution time over the optimised one. This gives a normalised account of the relative improvement of the enhanced version, where the 1x marker indicates the performance of the unoptimised version.

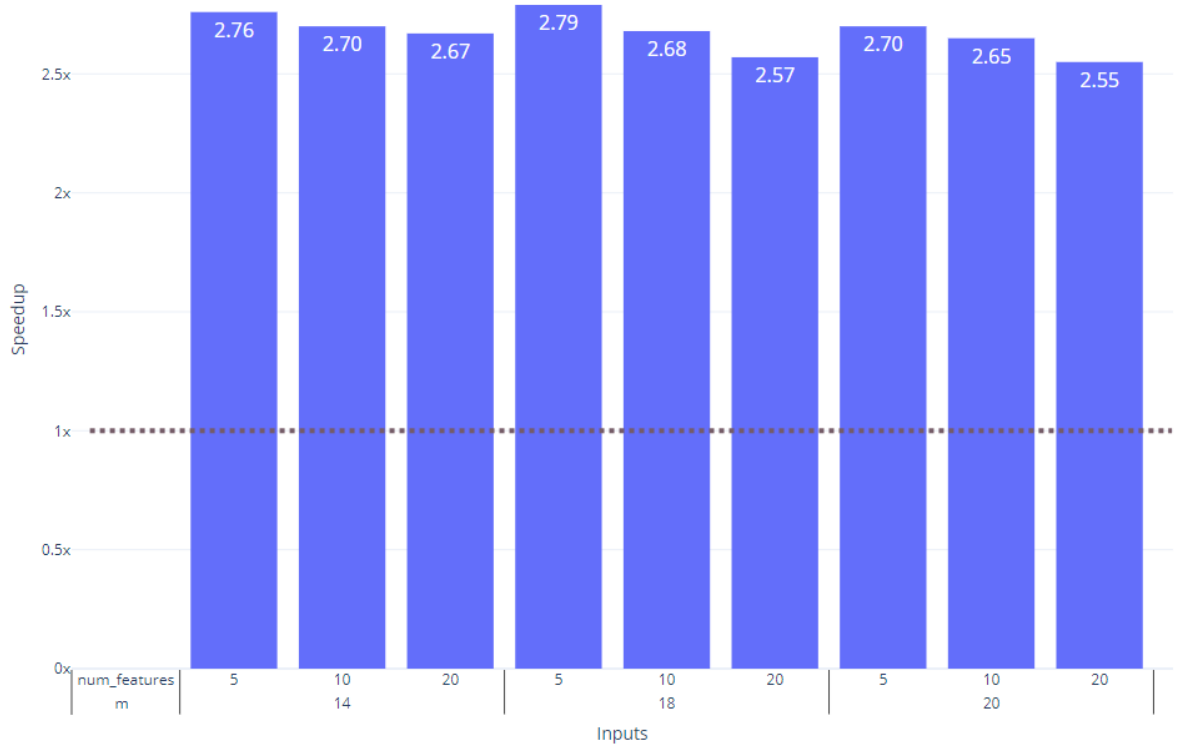


Figure 4.1.3.2: Optimized version speedup over unoptimized version in 9 different dim and m input scenarios. Single, 8 core / 16 threads machine.

The graph of Figure 4.1.3.2 shows how the improved version seems to offer consistent 2.5x speedups or more over the original version, for an average of 2.68x. A slight trend of the speedup being lower with higher dim counts can be observed, though nothing majorly outstanding.

As mentioned before, these results show the comparison with an already partially optimised version. Some artificial tests with older code showed a 3.3x total improvement. The most recent version of the base code was used as it also included bug fixes and other structural changes.

These results are very satisfactory and attest to the usefulness of profiling and optimisation before attempting parallelization.

4.1.4 Parallel versions results

Following the tests of the sequential versions, the optimised one was selected as the new baseline for all the parallel comparisons. All the parallel versions include the same performance improvements, so the differences stem only from the effectiveness of the parallelization method.

To follow are the tables with the raw result data (4.1.4.1, 4.1.4.2, 4.1.4.3, 4.1.4.4).

m	dim	timedelta	seconds	speedup	improvement	efficiency
14	5	00:00:03	3.263	1.33	33%	17%
14	10	00:00:06	6.378	1.34	34%	17%
14	20	00:00:14	13.648	1.3	30%	16%
18	5	00:00:51	50.606	1.37	37%	17%
18	10	00:01:47	106.764	1.33	33%	17%
18	20	00:03:48	227.608	1.34	34%	17%
20	5	00:03:24	204.292	1.36	36%	17%
20	10	00:06:57	416.679	1.36	36%	17%
20	20	00:14:59	898.888	1.35	35%	17%

Table 4.1.4.1: Parallel recursion only results.

m	dim	timedelta	seconds	speedup	improvement	efficiency
14	5	00:00:01	1.464	2.97	197%	37%
14	10	00:00:02	2.465	3.48	248%	43%
14	20	00:00:04	4.104	4.32	332%	54%
18	5	00:00:18	17.503	3.95	295%	49%
18	10	00:00:25	24.888	5.7	470%	71%
18	20	00:00:57	56.711	5.39	439%	67%
20	5	00:01:04	64.35	4.32	332%	54%
20	10	00:01:31	90.708	6.27	527%	78%
20	20	00:03:30	210.378	5.75	475%	72%

Table 4.1.4.2: Parallel split-point computation for each dimension results.

m	dim	timedelta	seconds	speedup	improvement	efficiency
14	5	00:00:04	4.417	0.98	-2%	12%
14	10	00:00:05	4.558	1.88	88%	24%
14	20	00:00:05	5.413	3.28	228%	41%
18	5	00:00:15	14.527	4.76	376%	60%
18	10	00:00:24	24.288	5.84	484%	73%
18	20	00:00:51	50.832	6.01	501%	75%
20	5	00:00:55	54.625	5.09	409%	64%
20	10	00:01:29	89.311	6.36	536%	80%
20	20	00:03:10	190.405	6.35	535%	79%

Table 4.1.4.3: Parallel combined split-point and recursion computation results.

m	dim	timedelta	seconds	speedup	improvement	efficiency
14	5	00:00:01	0.982	4.43	343%	55%
14	10	00:00:01	1.47	5.83	483%	73%
14	20	00:00:03	2.833	6.26	526%	78%
18	5	00:00:13	13.11	5.28	428%	66%
18	10	00:00:22	21.656	6.55	555%	82%
18	20	00:00:46	46.435	6.58	558%	82%
20	5	00:00:55	55.327	5.02	402%	63%
20	10	00:01:26	85.954	6.61	561%	83%
20	20	00:03:04	184.144	6.57	557%	82%

Table 4.1.4.4: Asynchronous recursion, parallel split-point results.

The following graphs will better show the data in context. The first one (Figure 4.1.4.1) shows the general behaviour of the solutions when considering the combined size of the input array obtained by multiplying the sample points dimensions by 2^m. While a lot of the points are too close to offer good insight, it is possible to notice how the sequential and parallel recursive solutions appear to be considerably slower than the other solutions. The graph also hides information regarding the actual `m` and `dim` values used.

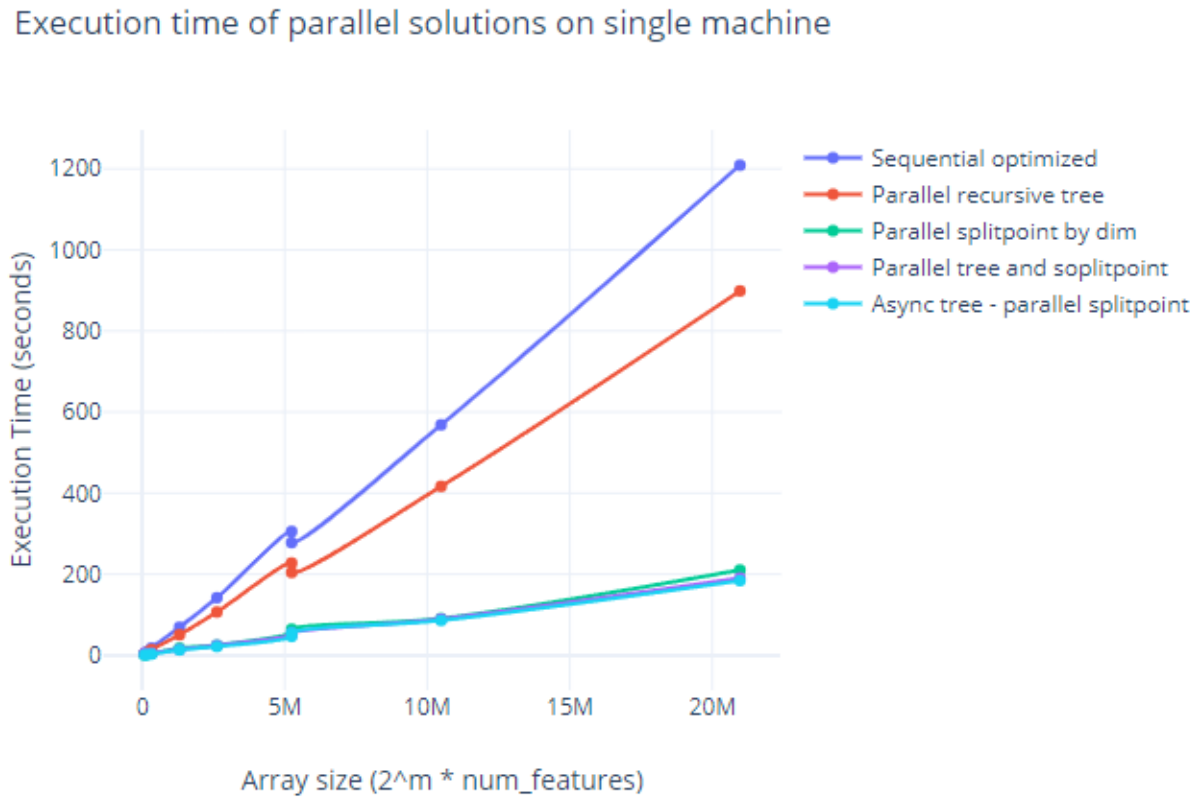


Figure 4.1.4.1: Execution time of the four parallel versions and the sequential optimized version on a single, 8 core / 16 threads machine. On the x axis is the total size of the samples array `X` as different problem sizes were tested.

The following bar graph in Figure 4.1.4.2 gives a more detailed overview of the speedup of each solution over the sequential optimised one, with the specific input values used for `dim` and `m` on the x axis.

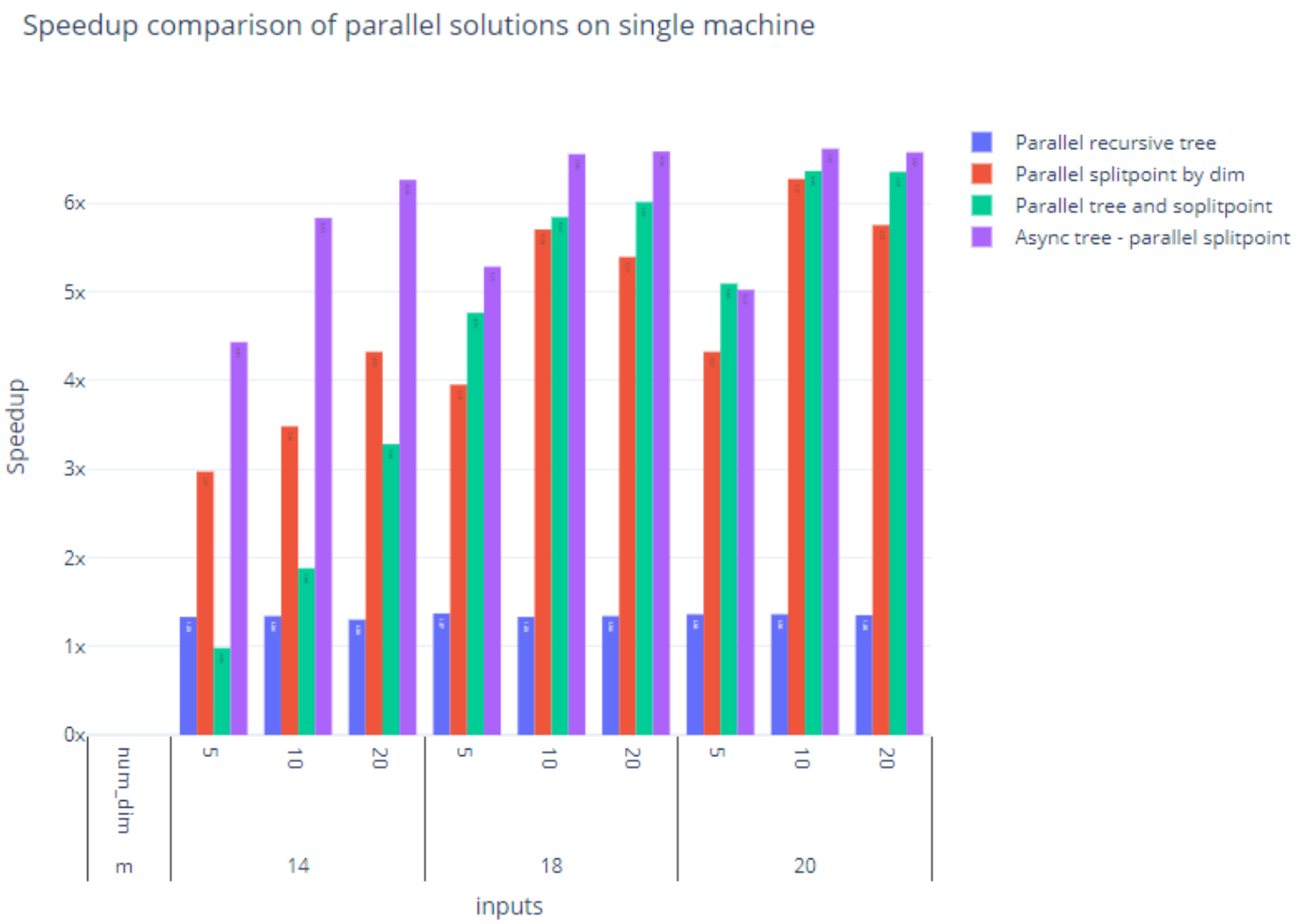


Figure 4.1.4.2: Speedup of the four parallel versions over the sequential optimized one on a single, 8 core / 16 threads machine. 9 different m and dim input combinations.

The graph is able to show how the solution offering the best improvement overall appears to be the asynchronous one. Some of the other solutions, in particular the parallel split-point computation one and the combined parallel recursion and split-point computation one are also competitive in some cases but not in all of them.

The parallel split-point computation solution in red consistently offers less performance than the async solution, due to it not being able to start computation of multiple tree nodes at once. The combined parallel split-point and recursive tree solution in green is perhaps the most competitive one, as it is also able to compute multiple tree nodes at the same time like the async solution, but it shows some big losses in the cases with smaller `m` values.

These particularly bad results are explained by our observations of high worker initialization overhead. The combined parallel solution uses nested Ray parallelism with `ray.get()` blocking operations which require the creation of new workers to avoid deadlocking. The overhead of each worker initialization appears to be fixed, so it stands to reason that it would appear less and less prominently as the total execution time increased. The total number of new workers instantiated could still be higher, and thus the total overhead greater, but the ratio of overhead to task execution time would still be lower.

All the solutions except for the parallel tree version show better speedups with higher number of domain dimensions. This is consistent with the fact that the `dim` value directly defines the amount of parallelism available to the solution. Higher values of `dim` imply higher number of parallel tasks, thus better utilisation of the multi core system.

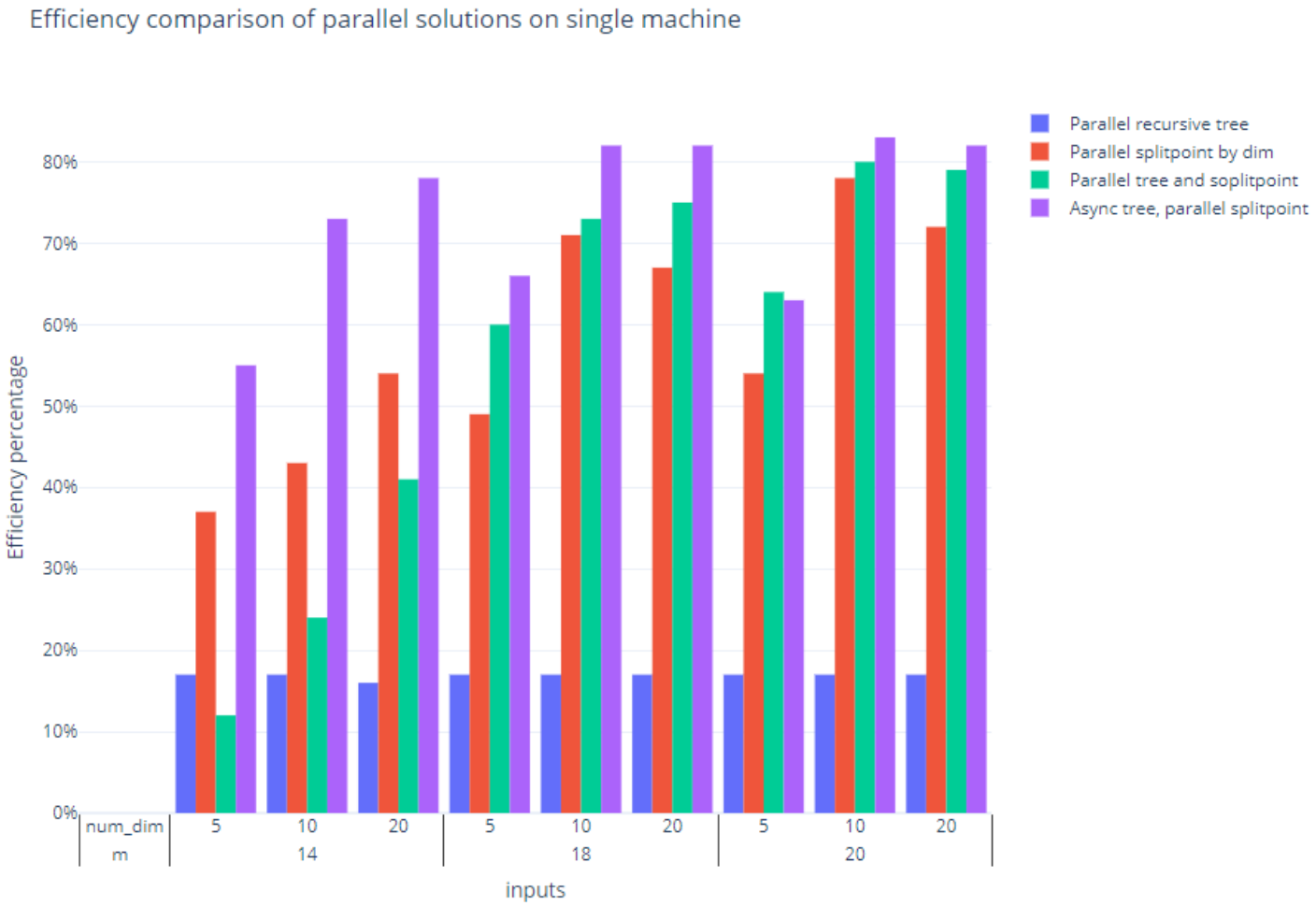


Figure 4.1.4.3: Efficiency of the four parallel versions on a single, 8 core / 16 threads machine. 9 different m and dim input combinations.

The graph of Figure 4.1.4.3 in this case shows the efficiency instead of the speedup. Efficiency is a metric that is obtained by dividing the speedup by the number of cores available on the system and represents how effectively the solution was able to utilise the

parallel architecture. The assumption is that if a workload was perfectly parallelizable, and the parallel implementation didn't introduce any overhead it would result in an efficiency of 1, or 100% as the speedup would match the number of parallel cores the program was run on. If the parallel solution introduces overhead or the problem complexity doesn't offer a level of parallelism greater than the number of execution units, then the efficiency will be lower.

In the case of the implemented solutions executed on a single machine with 8 cores, the maximum level of efficiency, 83%, was reached with the async solution. In general the async solution manages to maintain high levels of efficiency throughout. In comparison the combined parallel solution in green has some very low results with 12% and 24% which are lower than the red solution which is the one where only the split-point computation is parallelised.

It is possible to see how the maximum improvement of the parallel tree solution only reaches 37%. This means that that solution is only able to reduce the execution time by around one third of the original time. This in turn indicates that either the overhead is high enough to make the parallelism not worthwhile, or, more likely, the total parallelism obtainable through this method isn't particularly high. The level of parallelism obtainable from this solution depends on the structure of the binary tree. The wider and the more balanced the tree becomes, the higher the level of parallelism. What is more likely to be happening here is that the tree isn't balanced and it never reaches situations where it is particularly wide.

Some monitoring of resource usage during computation confirmed that the total number of cores used during the execution rarely exceeded 3, even with larger `m` and `dim` values. The same monitoring showed that the other solutions, which made use of the parallel computation of the split-point for each dimension, had much higher utilisation of the cores, with the async one being the one showing less utilisation dips.

In general these results seem to indicate that the best parallel implementation, at least in a local single machine environment, is the concurrent async solution. The parallel tree solution could be excluded from further testing on the grounds of offering extremely low performance improvements when compared to the other solutions, but in the end it was kept for further tests due to being deemed a good estimate of the amount of parallelism that the recursive tree could yield.

4.2 Distributed Cluster testing

Following experimentation on a single machine, the distributed nature of the solutions needed to be tested on a cluster, to not only to identify the best implementation, but also verify that the behaviour in a distributed environment wouldn't be divergent from expectations.

4.2.1 Cluster environment details

For the tests, a cluster of 98 identical machines of the campus laboratory of University of Bologna were used. Their specification is as described in Table 4.2.1.1.

System Product	OptiPlex 7090
Number of machines	98
CPU Model	Intel(R) Core(TM) i3-10305 CPU
CPU Cores	4
CPU Logical Cores / Threads	8
CPU Frequency	3.8GHz Base - 4.5GHz Boost
Cluster CPU Threads	784
Memory	15.38GB DDR4
Storage	diskless, 1TB shared NFS SSD
Interconnect	1Gbps Ethernet
Operating System	Ubuntu 22.04 LTS
Python Version	3.9.15

Table 4.2.1.1: Cluster environment details.

The cluster was set up with a master machine, identical to the others, except for the presence of a dedicated disk. The other machines (slaves) would all be diskless and be composed of Ubuntu images that would be launched on the physical machines. These images were prepared to have all the dependencies required to run Python and the Ray worker demon.

Each slave machine had a small RAM disk allocated for temporary files and normal operations, but to make the python script and all its dependencies available this wasn't enough. An SSD-based shared NFS was mounted on all machines. This is where the code and all its dependencies were stored so that all machines could access them.

In addition to handling the existing data, the NFS had the added task of offering the storage space for the Ray shared Object Store spillover data. In the case of the RAM size being insufficient for the allocation of new data in the shared Object Store, some of the data in it would be spilled to disk. Given that the amount of RAM on each cluster machine was inferior to what was available on the single machine system, and the size of the tests much larger, the possibility of incurring in Out of Memory scenarios was likely.

The tests performed had the goals of checking whether the solutions were working correctly on a distributed system and identifying the best performing one. The total number of computation units was much larger than the maximum achievable parallelism for most of the tests, meaning that the environment allowed observing the behaviour of the solutions in a

situation with unbounded computation resources, thus identifying the one that would achieve the highest parallelism and performance.

Thought was given on testing the weak and strong scalability of the developed solution. Weak and strong scalability are terms commonly used in high performance computing and express the ability of hardware and software to deliver greater computational performance when the amount of resources is increased. Strong scalability is defined as how the solution time varies with the number of processors for a fixed problem size. Weak scalability is instead defined as how the solution time varies with the number of processors for fixed problem size per processor [48].

In our case, the nature of the workload did not allow for weak scalability testing, as there wasn't a reliable and consistent way to increase the size of the problem together with the number of processors, while maintaining a fixed workload per processor. Testing was instead conducted to measure the strong scalability of the solutions with fixed problem sizes by configuring the cluster to use a reduced number of machines and a reduced number of total available CPUs.

4.2.2 Distributed unbounded testing results

The unbounded testing was performed with all the available 784 logical cores included in the cluster. The four developed solutions plus the optimised sequential one were tested with up to twenty two different combinations of input parameters, where `m` is the power of two indicating the length of the sample array `X`, and `dim` is the number of dimension of each sample point contained in `X`, so the number of variables of the model.

To follow are the raw results of the various solutions (Tables 4.2.2.1, 4.2.2.2, 4.2.2.3, 4.2.2.4, 4.2.2.5). Some combinations were skipped for some solutions as they required too much time to complete. In particular some speedup and improvement data is missing for some combinations as measuring their sequential execution time would have required too much time (multiple hours).

m	dim	timedelta	seconds	speedup	improvement
14	5	00:00:03	3.426	1	0%
14	10	00:00:07	6.807	1	0%
14	20	00:00:15	14.577	1	0%
14	50	00:00:54	54.477	1	0%
14	100	00:03:44	223.827	1	0%
14	200	-	-	-	-
14	400	-	-	-	-
18	5	00:00:55	55.028	1	0%
18	10	00:01:56	116.087	1	0%

18	20	00:04:17	256.83	1	0%
18	50	00:16:46	1005.512	1	0%
18	100	01:04:11	3850.833	1	0%
18	200	-	-	-	-
18	400	-	-	-	-
20	5	00:03:56	235.836	1	0%
20	10	00:08:05	485.176	1	0%
20	20	00:18:04	1083.778	1	0%
20	50	01:07:57	4076.831	1	0%
20	100	04:17:15	15434.529	1	0%
22	5	00:16:04	964.032	1	0%
22	10	00:33:09	1989.084	1	0%
22	20	-	-	-	-

Table 4.2.2.1: Sequential optimized results, used as baseline for speedup and improvement metrics in following tables.

m	dim	timedelta	seconds	speedup	improvement
14	5	00:00:03	3.106	1.1	10%
14	10	00:00:06	5.942	1.15	15%
14	20	00:00:13	13.031	1.12	12%
14	50	00:00:45	44.989	1.21	21%
14	100	00:03:00	180.431	1.24	24%
14	200	-	-	-	-
14	400	-	-	-	-
18	5	00:00:47	47.061	1.17	17%
18	10	00:01:37	97.365	1.19	19%
18	20	00:03:37	216.624	1.19	19%
18	50	00:13:05	785.098	1.28	28%
18	100	00:52:05	3124.576	1.23	23%
18	200	-	-	-	-
18	400	-	-	-	-
20	5	00:03:10	190.475	1.24	24%
20	10	00:06:35	395.448	1.23	23%
20	20	00:14:28	868.047	1.25	25%
20	50	00:54:21	3260.941	1.25	25%
20	100	-	-	-	-
22	5	-	-	-	-
22	10	-	-	-	-

22	20	-	-	-	-

Table 4.2.2.2: Parallel recursive tree results.

m	dim	timedelta	seconds	speedup	improvement
14	5	00:00:02	1.9	1.8	80%
14	10	00:00:03	3.189	2.13	113%
14	20	00:00:08	8.391	1.74	74%
14	50	00:00:14	13.885	3.92	292%
14	100	00:00:15	14.575	15.36	1436%
14	200	00:01:03	62.641	-	-
14	400	00:06:37	397.444	-	-
18	5	00:00:22	22.226	2.48	148%
18	10	00:00:34	34.128	3.4	240%
18	20	00:00:39	38.865	6.61	561%
18	50	00:01:14	74.019	13.58	1258%
18	100	00:03:01	181.259	21.24	2024%
18	200	00:12:12	732.089	-	-
18	400	02:09:39	7778.903	-	-
20	5	00:01:28	87.569	2.69	169%
20	10	00:02:16	135.525	3.58	258%
20	20	00:02:35	154.989	6.99	599%
20	50	00:05:26	325.736	12.52	1152%
20	100	00:13:48	828.122	18.64	1764%
22	5	00:05:33	332.582	2.9	190%
22	10	00:09:07	546.663	3.64	264%
22	20	00:10:33	633.046	-	-

Table 4.2.2.3: Parallel split-point computation results.

m	dim	timedelta	seconds	speedup	improvement
14	5	00:00:05	4.986	0.69	-31%
14	10	00:00:06	5.824	1.17	17%
14	20	00:00:06	6.338	2.3	130%
14	50	00:00:08	8.126	6.7	570%
14	100	00:00:13	12.698	17.63	1663%
14	200	00:00:53	53.367	-	-
14	400	00:05:57	356.737	-	-
18	5	00:00:16	16.222	3.39	239%
18	10	00:00:19	19.169	6.06	506%

18	20	00:00:29	29.488	8.71	771%
18	50	00:01:11	71.108	14.14	1314%
18	100	00:06:01	361.064	10.67	967%
18	200	00:15:14	913.934	-	-
18	400	01:49:16	6556.44	-	-
20	5	00:01:00	59.585	3.96	296%
20	10	00:01:11	71.431	6.79	579%
20	20	00:02:05	124.79	8.68	768%
20	50	00:07:50	469.798	8.68	768%
20	100	00:20:15	1214.534	12.71	1171%
22	5	00:04:28	267.603	3.6	260%
22	10	00:05:26	325.66	6.11	511%
22	20	00:12:32	752.048	-	-

Table 4.2.2.4: Combined nested parallel computation results.

m	dim	timedelta	seconds	speedup	improvement
14	5	00:00:01	1.353	2.53	153%
14	10	00:00:04	3.964	1.72	72%
14	20	00:00:03	2.564	5.69	469%
14	50	00:00:04	4.311	12.64	1164%
14	100	00:00:12	11.742	19.06	1806%
14	200	00:00:48	47.715	-	-
14	400	00:05:49	349.316	-	-
18	5	00:00:19	18.728	2.94	194%
18	10	00:00:25	25.455	4.56	356%
18	20	00:00:30	29.705	8.65	765%
18	50	00:00:59	58.941	17.06	1606%
18	100	00:02:36	155.526	24.76	2376%
18	200	00:10:01	601.227	-	-
18	400	01:44:28	6268.097	-	-
20	5	00:01:15	74.624	3.16	216%
20	10	00:01:41	101.428	4.78	378%
20	20	00:01:57	117.16	9.25	825%
20	50	00:04:06	246.195	16.56	1556%
20	100	00:11:15	675.012	22.87	2187%
22	5	00:04:53	293.379	3.29	229%
22	10	00:06:49	408.753	4.87	387%

22	20	00:07:42	461.774	-	-

Table 4.2.2.5: Asynchronous recursive tree, parallel split-point computation results.

The following graph (Figure 4.2.2.1) shows the relative speedup of the distributed solutions over the sequential one, at each input combination point.

Figure 4.2.2.1: Speedup of the four parallel versions over the sequential optimized one, on a 98 machine cluster (4 cores / 8 threads). 17 different m and dim input combinations.

The speedup of parallel recursive solution in red remains consistent throughout the input combinations, at around 1.1 ~ 1.25x. This is an indication that the maximum level of parallelism achievable with that option doesn't increase with the inputs like the other solutions. All three other solutions contain parallelization of the split-point computation, a separate task for each domain dimension, and the graph shows this by demonstrating a general increase in speedup as `dim` increases.

Changes in `m` offer less predictable behaviour. The variable controls the power of two that indicates the length of the array of sample points `X`. From 14 to 18 a general speedup improvement can be noticed. The cause is likely due to `m=14` having tasks that are very short,

so the overhead of the distributed solution is harder to overcome. With `m=18`, the size of the tasks is more substantial, managing to hide the overheads better, while with `m=20` and `m=22`, the size starts to become big enough to cause a reduction in speedup due to higher memory related overheads. More data needs to be allocated, copied and transferred across machines. The resource limits of the machines are also getting in reach, potentially causing massive slowdowns in cases of RAM saturation.

Comparing the speedups offered by the distributed solutions, it can be observed that, as expected, the green (parallel split-point by dimension only) usually remains lower than the other two solutions that also handle the recursive tree in some manner. There are some outliers, mostly the combined parallel version in purple being slower on multiple occasions and the single 14-10 case where even the async version performed worse.

Logically purple should be higher than green, as it should reach a higher total parallelism, but, as also seen on the single machine tests, the nested parallelism solution with Ray can have issues with worker spam, added delays for new worker initialization and general resource usage problems. Machines with already 8 workers allocated could be spawning new ones, causing memory usage to exceed what is available on the single machine, causing big performance degradations.

When comparing the concurrent async solution, where the recursive tree is all handled on a single async process, with the nested parallelism one, some cases can be seen, especially with lower values of `dim`, where the nested solution offers better improvements. A better view of the differences can be seen with the following graph (Figure 4.2.2.2), where the speedup of the async version over the Nested parallelism one is shown. This is computed as the execution time of the combined parallel version over the execution time of the async solution.

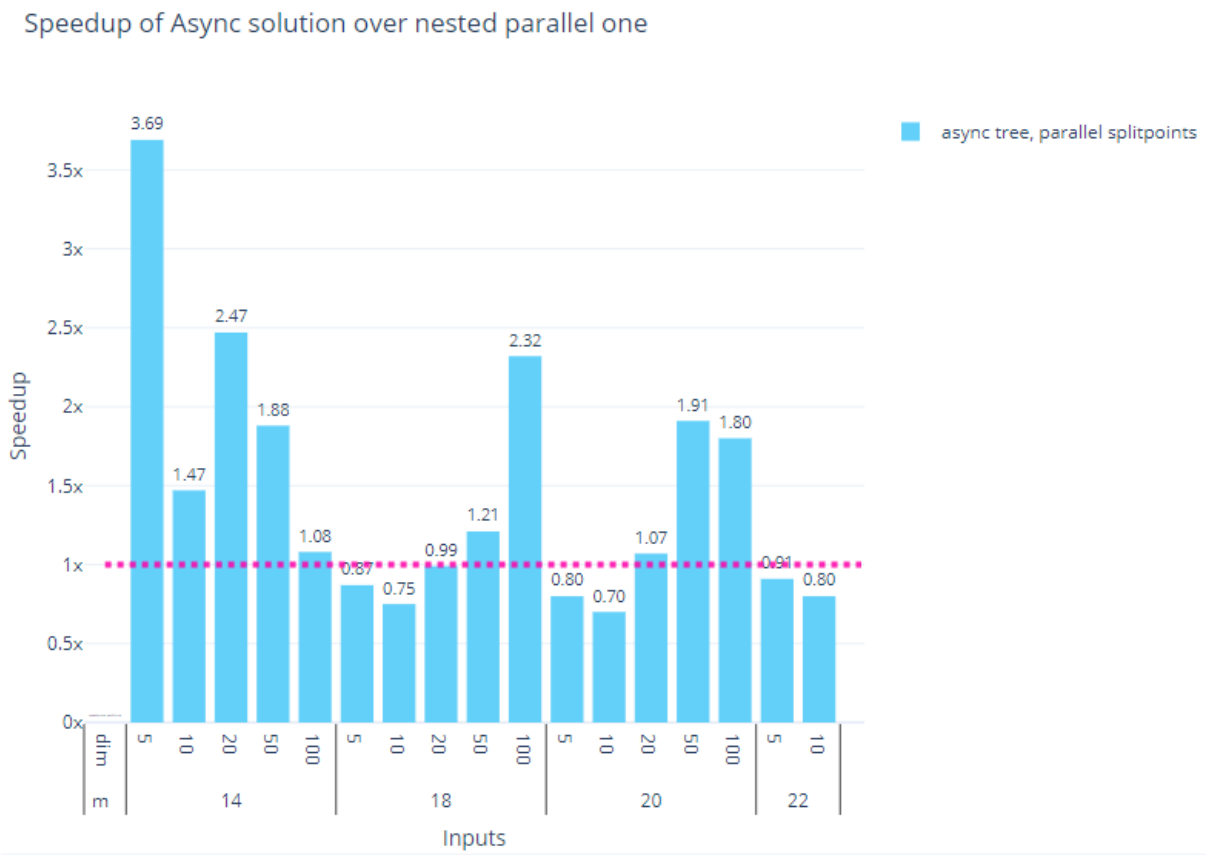


Figure 4.2.2.2: Relative speedup of async parallel version over the nested parallel one, on a 98 machine cluster (4 cores / 8 threads). 17 different m and dim input combinations.

Except in some cases with lower values of `dim`, the async version can offer great improvements over the alternative, especially at higher `dim` values. The relative better performance of the parallel nested solution could be attributed to having the advantage of actual parallel execution of the functions handling each tree node. The advantage could then be lost to higher resource usage as higher values of dim increase the total number of workers and machines used.

Looking at an execution timeline for the nested parallelism solution (Figure 3.5.5.4), it's possible to see how even though there are more machines and idle workers available, Ray still allocates more workers per machine than normal, due to workers being blocked on `ray.get()` calls. As machines start to get more saturated with a higher number of tasks to handle, the speedup over the sequential solution can reach a limit or even encounter regressions when compared with the version that only parallelizes the split-point computation.

The async solution appears to instead be able to scale much better with higher values of `dim`, while not losing performance on the smallest input sizes, where the nested solution struggles to overcome the overhead of using dedicated tasks. The advantage offered by the async solution in the favourable cases is very significant, being more than twice as fast. The main reason being that of having better resource efficiency.

In general, these results show how the nested parallelism is inconsistent especially when compared with the split-point parallel only version, which in turn is steadily outperformed by the async version. The latter appears to be the best solution from these tests.

The graphs used for the distributed scenario so far didn't feature execution time, nor the combined size of the input array on the x axis, computed as $2^m * dim$. The following graph (Figure 4.2.2.3) shows the execution time of the async solution over the combined input size and it is clear why these metrics weren't used for the other graphs.

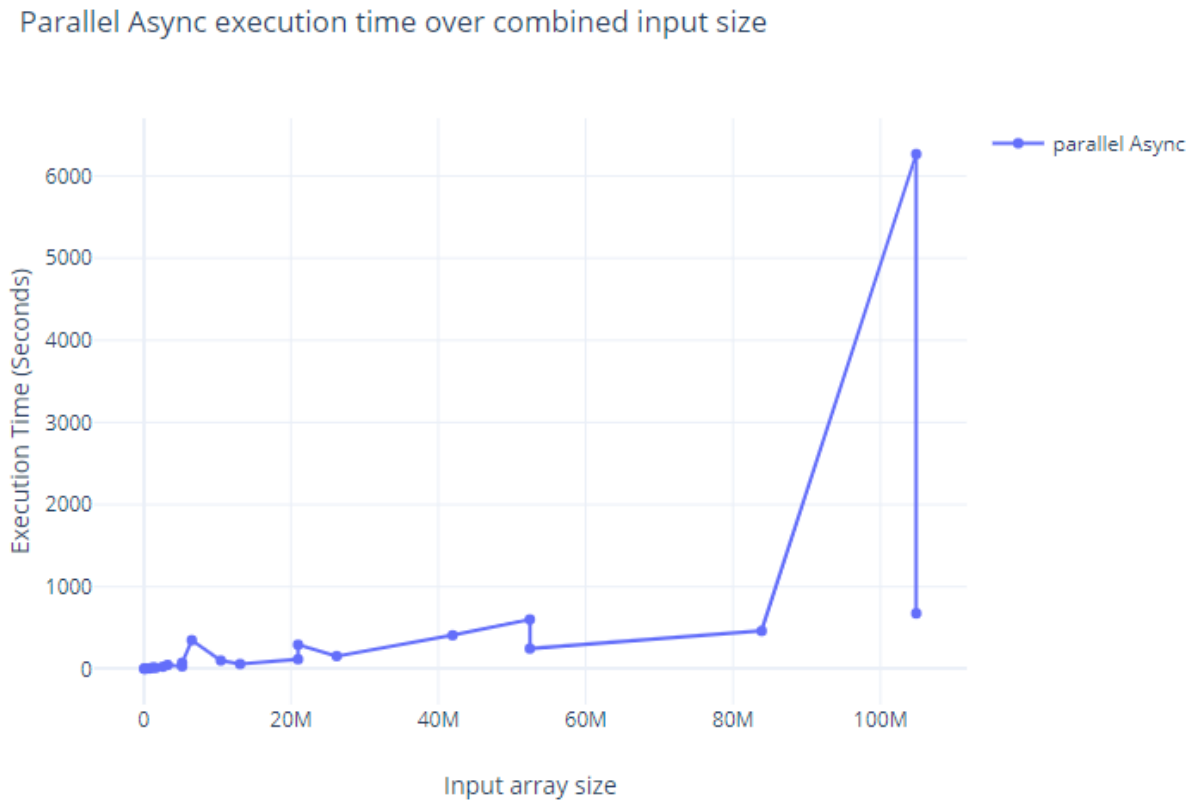


Figure 4.2.2.3: Execution time of async parallel version on a 98 machine cluster (4 cores / 8 threads). On the x axis is the total size of the `X` *input samples array.*

The y axis scale makes distinguishing most results values hard, and the graph line is very erratic. It doesn't seem to be making any apparent sense. This is because the used values of `dim` and `m` are obscured, which actually have a large bearing on the program execution and performance. Higher values of `dim` allow for much higher parallelism but more computation, while higher values of `m` don't equate to higher parallelism, but also don't increase the workload as much either. In general such a graph doesn't offer much useful information and the formats used before were preferred instead.

Before moving on to the results of the scalability testing, another experiment was performed, where, instead of using all the available resources of each node, Ray was limited to just 3 cores per node, so only using physical cores and even leaving one for extra for other things

like operating system. This was done in the hope of increasing performance by reducing the workload that each machine was put under, avoiding going out of memory and causing bottlenecks or delays due to saturated machines. Some tests were performed with the async solution, to see whether the increased breathing room made any difference. The collected data is reported below (Table 4.2.2.6).

m	dim	timedelta 8	seconds 8	timedelta 3	seconds 3	3 cores speedup
14	5	00:00:01	1.353	00:00:03	3.429	0.39
14	10	00:00:04	3.964	00:00:03	3.498	1.13
14	20	00:00:03	2.564	00:00:03	2.964	0.87
14	50	00:00:04	4.311	00:00:06	5.858	0.74
14	100	00:00:12	11.742	00:00:12	11.534	1.02
14	200	00:00:48	47.715	00:00:39	39.416	1.21
14	400	00:05:49	349.316	00:03:51	231.478	1.51
18	5	00:00:19	18.728	00:00:11	11.351	1.65
18	10	00:00:25	25.455	00:00:15	14.808	1.72
18	20	00:00:30	29.705	00:00:25	24.639	1.21
18	50	00:00:59	58.941	00:01:02	62.258	0.95
18	100	00:02:36	155.526	00:03:13	193.17	0.81
18	200	00:10:01	601.227	00:10:24	623.928	0.96
18	400	01:44:28	6268.097	01:23:37	5017.497	1.25
20	5	00:01:15	74.624	00:00:45	45.17	1.65
20	10	00:01:41	101.428	00:00:57	57.149	1.77
20	20	00:01:57	117.16	00:01:34	94.015	1.25
20	50	00:04:06	246.195	00:05:35	335.447	0.73
20	100	00:11:15	675.012	00:15:13	912.873	0.74
20	200	-	-	00:44:21	2660.999	-
22	5	00:04:53	293.379	00:02:58	178.183	1.65
22	10	00:06:49	408.753	00:04:01	241.482	1.69
22	20	00:07:42	461.774	00:07:34	454.309	1.02
22	50	-	-	00:21:19	1279.353	-

Table 4.2.2.6: Async parallel solution with 8 and 3 worker per node results.

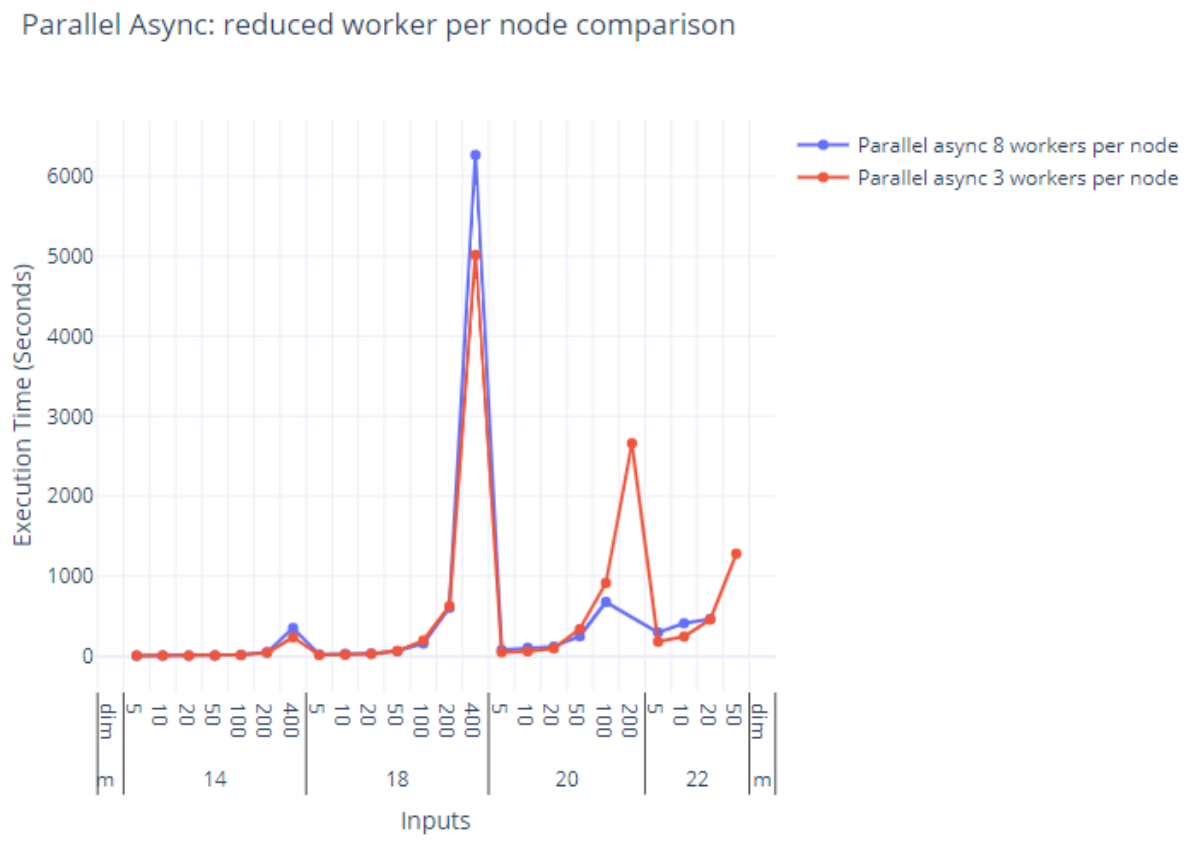


Figure 4.2.2.4: Execution time of async parallel solution on two different cluster configurations: 98 machines with 8 workers per node (784 cores) in blue; 98 machines with 3 workers per node (294 cpus) in red. 24 different `m` *and* `dim` *input combinations, some cases missing from the 8 workers data.*

In Figure 4.2.2.4 again the use of the execution time isn't ideal, but it shows that the 8 workers per node solution performed worse especially on cases where the execution time was already slow. Some results (20/200, 22/50) are then missing from the 8 workers per node scenario as they were incurring in out-of-memory (OOM) errors and thus unable to complete, while they completed successfully on the 3 workers per node cluster setup.

The following graph (Figure 4.2.2.5) gives a better overview of the relative performance of the 3 worker per node configuration, with respect to the normal 8 workers variant. The speedup is computed as the execution time of the 8 workers variant over the execution time of the 3 worker configuration.

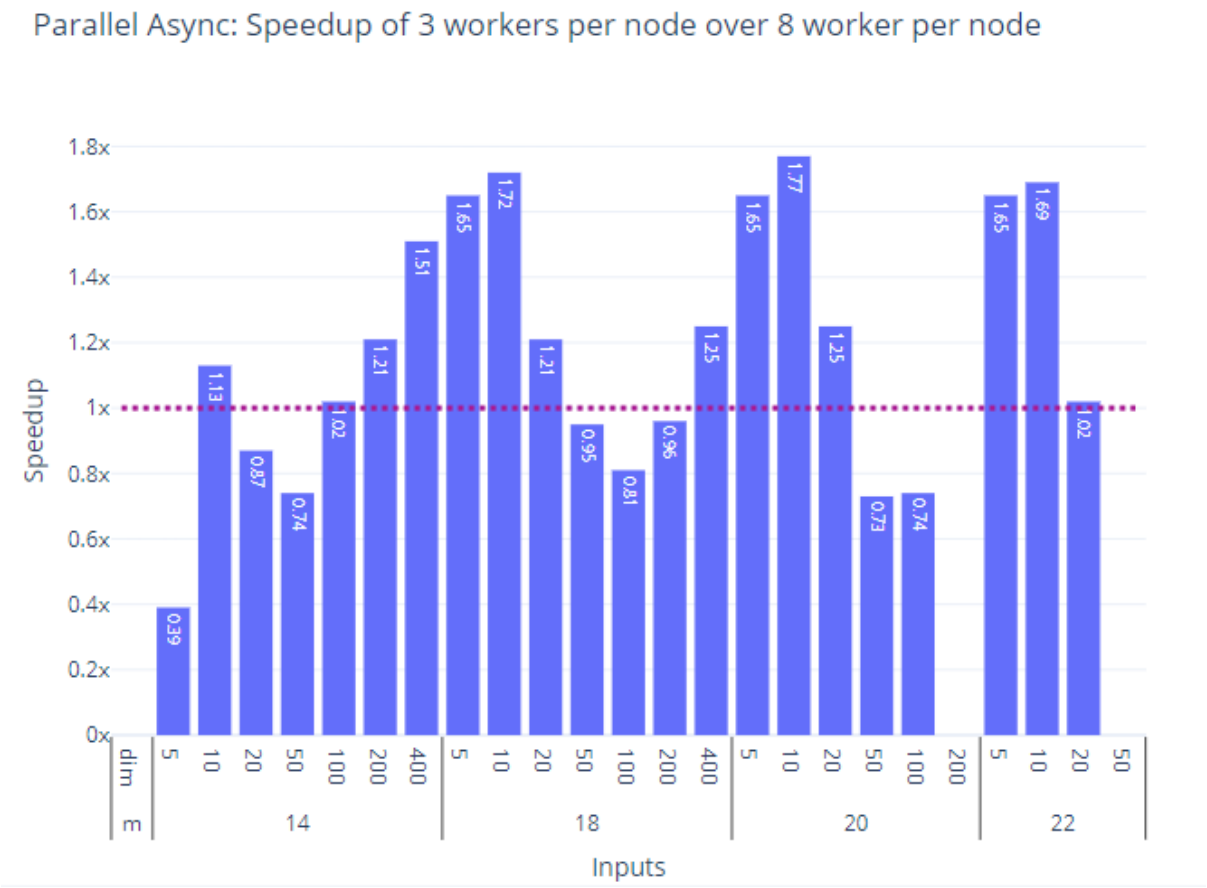


Figure 4.2.2.5: async parallel version speedup of 3 worker per node configuration over 8 worker per node configuration. 22 different m and dim input combinations.

The purple line indicates the performance level of the 8 workers per node solution. The results are mixed, with some better results when the system is really stressed and worse results when the workload is lower. This makes sense since limiting the number of workers per node would require the use of more machines to complete the same amount of work, potentially adding more communication overhead. The limiting of Ray workers per node can be an effective strategy to reduce the resource usage per machine and thus avoiding scenarios such as out-of-memory errors when dealing with large problem sizes.

4.2.3 Strong scalability testing on limited cluster sizes

To further compare the solutions, some more rigorous testing was performed to observe the behaviour of the solution when executing on limited clusters with different sizes. In particular the experiments are meant to emphasise the scalability of the different versions when the parallelism of the underlying system increases.

Since the cluster configuration is the test variable in these experiments, the input values need to be fixed. To not risk picking a particularly unfavourable combination, three samples were chosen as indicated in Table 4.2.3.1.

m	dim
16	20
18	20
18	50

Table 4.2.3.1: Input parameters combinations.

`m=16` offers good performance without sacrificing accuracy, while 18 is the more extreme case. `dim=20` is the average number of features of a model that the algorithm is expected to handle. The additional 18-50 combination was included to have a look at a different `dim` size and look at the behaviour in a more taxing scenario.

The following cluster configurations were used for the experiments (Table 4.2.3.2).

num CPUs	num machines
8	2
16	4
32	8
64	16

Table 4.2.3.2: Cluster configurations.

To set up the cluster, the Ray demon was only started on a number of machines equal to the numbers on the table, including the master. Additionally the demon was limited to only use 4 CPUs per machine, so one for each physical core.

To follow in Table 4.2.3.3 is the data obtained from the experiments:

solution	m	dim	timedelta	seconds	speedup	efficiency	num_cpus
sequential	16	20	01:04.8	64.828	-	-	1
sequential	18	20	04:25.7	265.661	-	-	1
sequential	18	50	15:52.8	952.806	-	-	1
par tree	16	20	00:50.1	50.118	1.294	16%	8
par tree	18	20	03:23.9	203.869	1.303	16%	8
par tree	18	50	12:12.0	732.012	1.302	16%	8
par tree	16	20	00:50.7	50.744	1.278	8%	16
par tree	18	20	03:27.3	207.254	1.282	8%	16
par tree	18	50	12:24.2	744.241	1.28	8%	16

par splitpoint	16	20	00:12.7	12.712	5.1	64%	8
par splitpoint	18	20	00:47.5	47.526	5.59	70%	8
par splitpoint	18	50	02:31.2	151.241	6.3	79%	8
par splitpoint	16	20	00:09.6	9.583	6.765	42%	16
par splitpoint	18	20	00:36.9	36.934	7.193	45%	16
par splitpoint	18	50	01:39.4	99.356	9.59	60%	16
par splitpoint	16	20	00:09.9	9.923	6.533	20%	32
par splitpoint	18	20	00:28.5	28.48	9.328	29%	32
par splitpoint	18	50	01:13.3	73.337	12.992	41%	32
par splitpoint	16	20	00:10.7	10.726	6.044	9%	64
par splitpoint	18	20	00:28.3	28.281	9.394	15%	64
par splitpoint	18	50	01:04.4	64.391	14.797	23%	64
par nested	16	20	00:13.2	13.205	4.909	61%	8
par nested	18	20	00:44.6	44.581	5.959	74%	8
par nested	18	50	02:28.0	148.042	6.436	80%	8
par nested	16	20	00:10.0	9.971	6.502	41%	16
par nested	18	20	00:33.6	33.608	7.905	49%	16
par nested	18	50	01:28.2	88.21	10.802	68%	16
par nested	16	20	00:10.4	10.431	6.215	19%	32
par nested	18	20	00:27.0	26.99	9.843	31%	32
par nested	18	50	01:24.4	84.361	11.294	35%	32
par nested	16	20	00:08.8	8.811	7.358	11%	64
par nested	18	20	00:26.9	26.947	9.859	15%	64
par nested	18	50	01:25.3	85.272	11.174	17%	64
par async	16	20	00:11.8	11.808	5.49	69%	8
par async	18	20	00:44.1	44.059	6.03	75%	8
par async	18	50	02:25.6	145.579	6.545	82%	8
par async	16	20	00:07.7	7.738	8.378	52%	16
par async	18	20	00:27.9	27.86	9.536	60%	16
par async	18	50	01:27.2	87.208	10.926	68%	16
par async	16	20	00:08.0	8.016	8.087	25%	32
par async	18	20	00:21.8	21.803	12.185	38%	32
par async	18	50	00:59.4	59.363	16.051	50%	32
par async	16	20	00:07.8	7.8	8.311	13%	64
par async	18	20	00:20.5	20.546	12.93	20%	64
par async	18	50	00:55.3	55.302	17.229	27%	64

Table 4.2.3.3: CPU scaling result data.

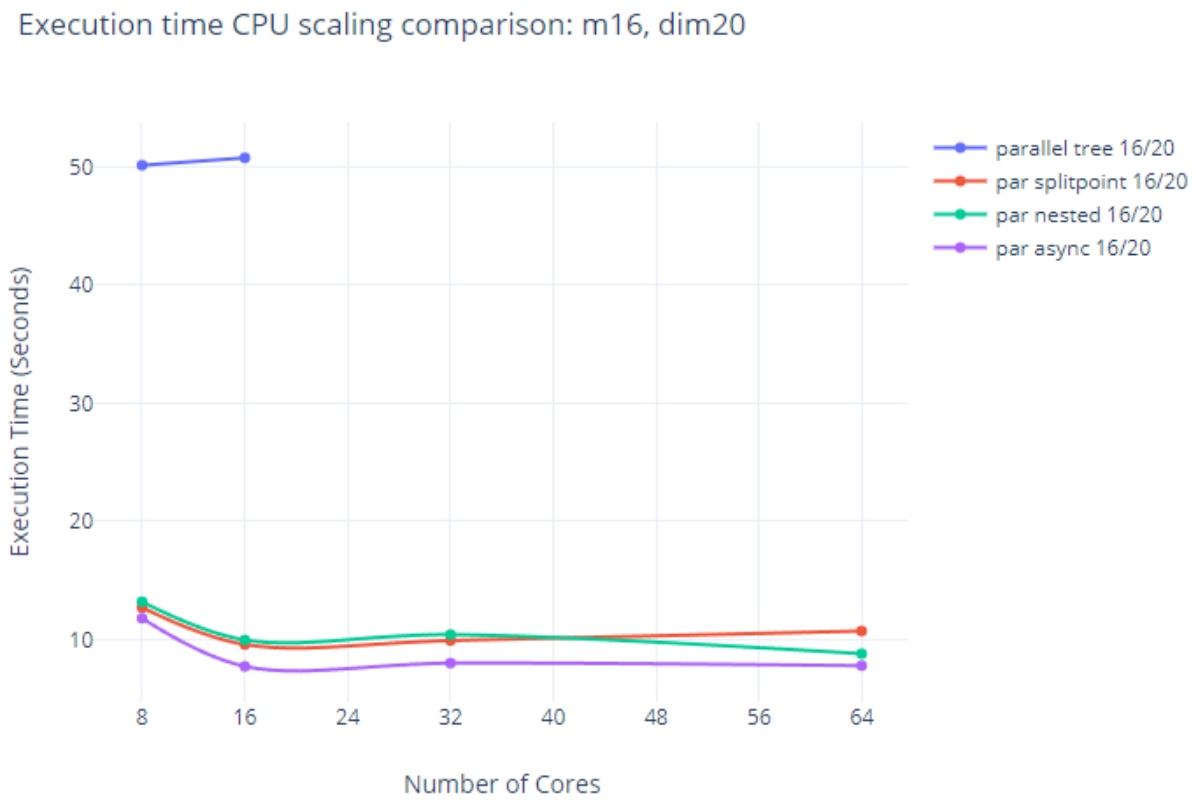


Figure 4.2.3.1: Execution time of the four parallel versions, with fixed input size of: `m=16`*,* `dim=20`*. Tested on 8, 16, 32, 64 cores cluster configurations (4 cores per machine).*

This first graph (Figure 4.2.3.1) shows the execution time of all the parallel solutions with inputs 16/20. By far the slowest, the parallel recursion version wasn't tested on all configurations as it clearly didn't demonstrate any further improvement as more CPUs were added. A slight decrease in performance was actually registered. Likely due to increased cluster management overhead by Ray.

To have a better view of the other results, the first solution was excluded from the graph.

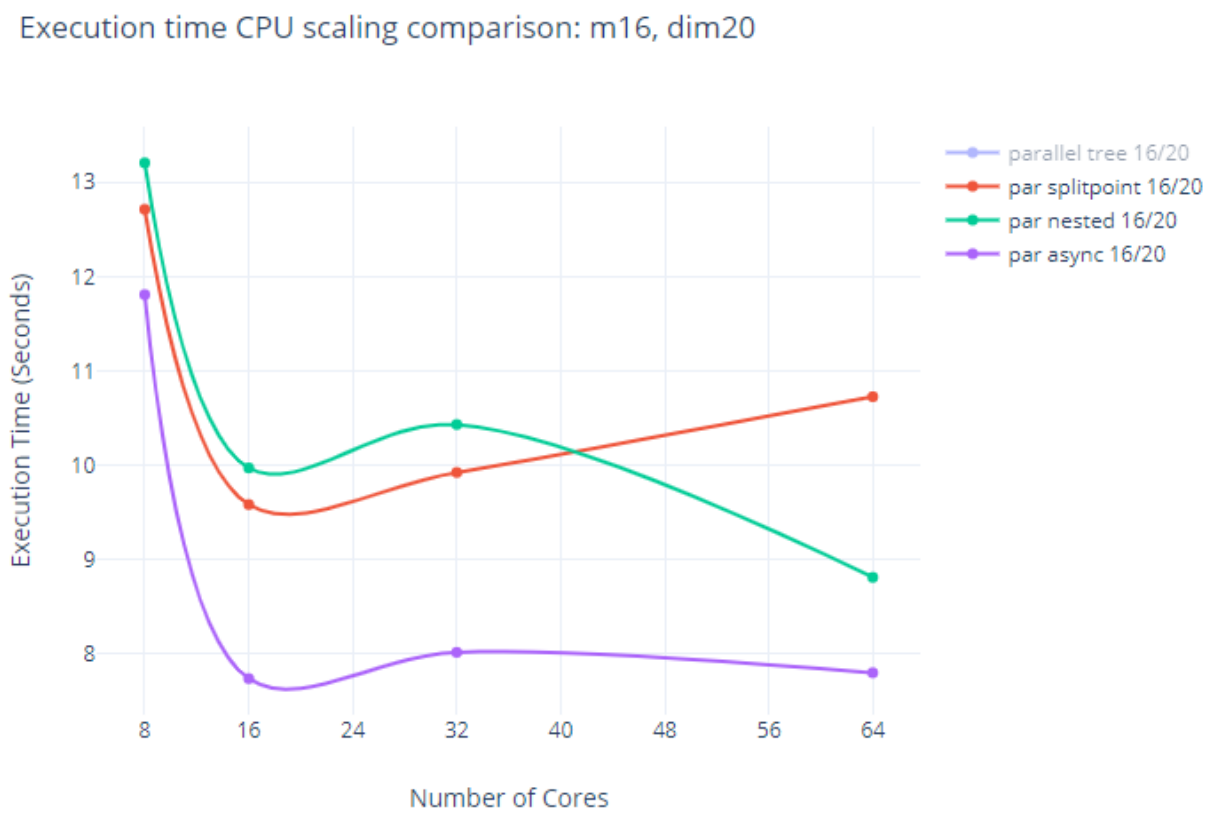


Figure 4.2.3.2: Execution time of three parallel versions, with fixed input size of: `m=16`, `dim=20`. *Tested on 8, 16, 32, 64 cores cluster configurations (4 cores per machine). Parallelized tree version was excluded.*

In this version of the graph (Figure 4.2.3.2) it's easier to see the differences between the various solutions. All of them seem to perform worse when limited to 8 cores and see better results on 16 cores. Further than that a regression seems to appear at 32 cores. It gets worse at 64 cores for the version with the parallel split-point computation, while the other two solutions manifest better behaviour. The async version still performs worse on 64 cores than on 16, but it is still the best performing solution of all of them.

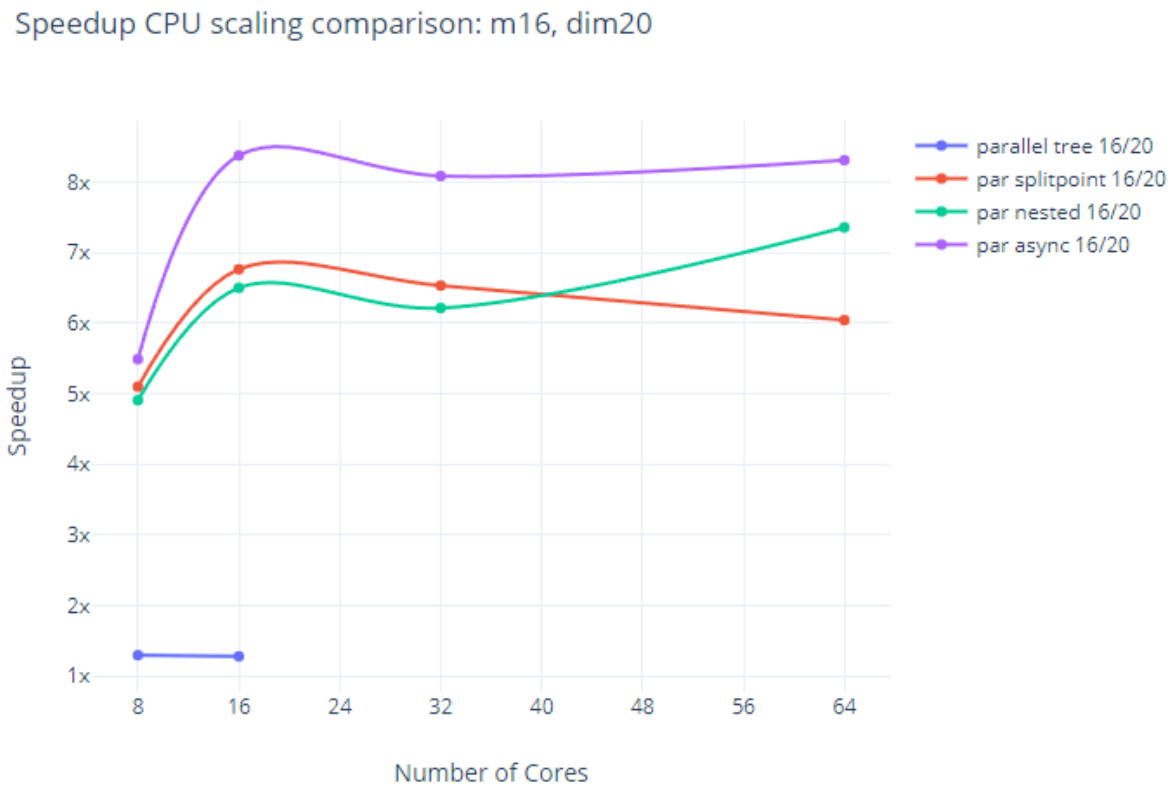


Figure 4.2.3.3: Speedup of the four parallel versions over the sequential optimized one, with fixed input size of: `m=16`, `dim=20`. *Tested on 8, 16, 32, 64 cores cluster configurations (4 cores per machine).*

This graph (Figure 4.2.3.3) refers to the same experiment but showing the speedup values instead of the execution time. The speedup seems to increase when going from 8 to 16 cores, but after that it doesn't seem to be able to gain much more when adding further cores. The `dim` value determines the parallelism of the solution; in this case the results seem to reflect the fact by showing that the best performance can be obtained when the number of CPUs approaches the value of `dim`.

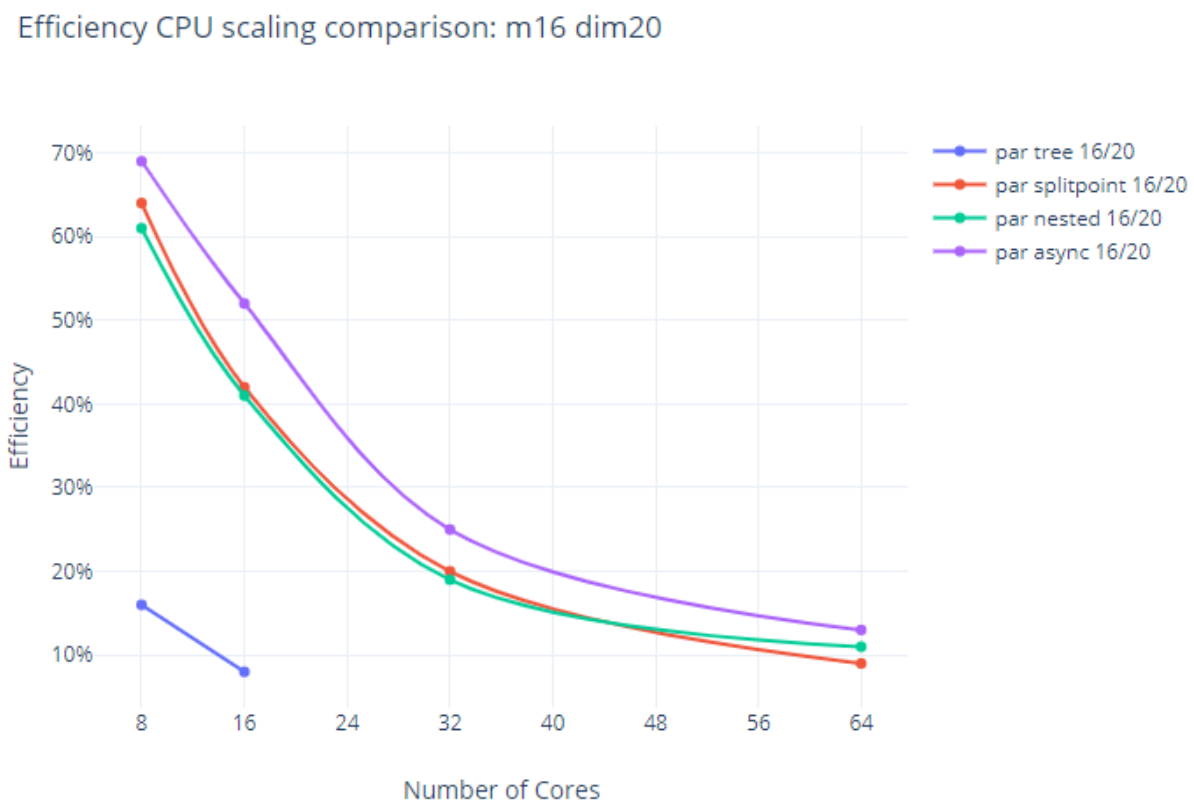


Figure 4.2.3.4: Efficiency of the four parallel versions, with fixed input size of: `m=16`*,* `dim=20`*. Tested on 8, 16, 32, 64 cores cluster configurations (4 cores per machine).*

Of more interest is the efficiency curve of the solutions (Figure 4.2.3.4), computed as the speedup over the number of available cores. The parallelization of the recursive tree isn't able to utilise the available resources at all, resulting in low percentages. The other solutions behave overall very similarly, with the async one being the most efficient one.

Moving to the `m=18`, `dim=20` test, we get a more expected pattern. Like before the results of the parallel recursive tree solutions were excluded as they were skewing the y axis (Figure 4.2.3.5). The performance this time seems to continue to improve for all solutions as more CPUs are added, with diminishing returns at 64 cores.

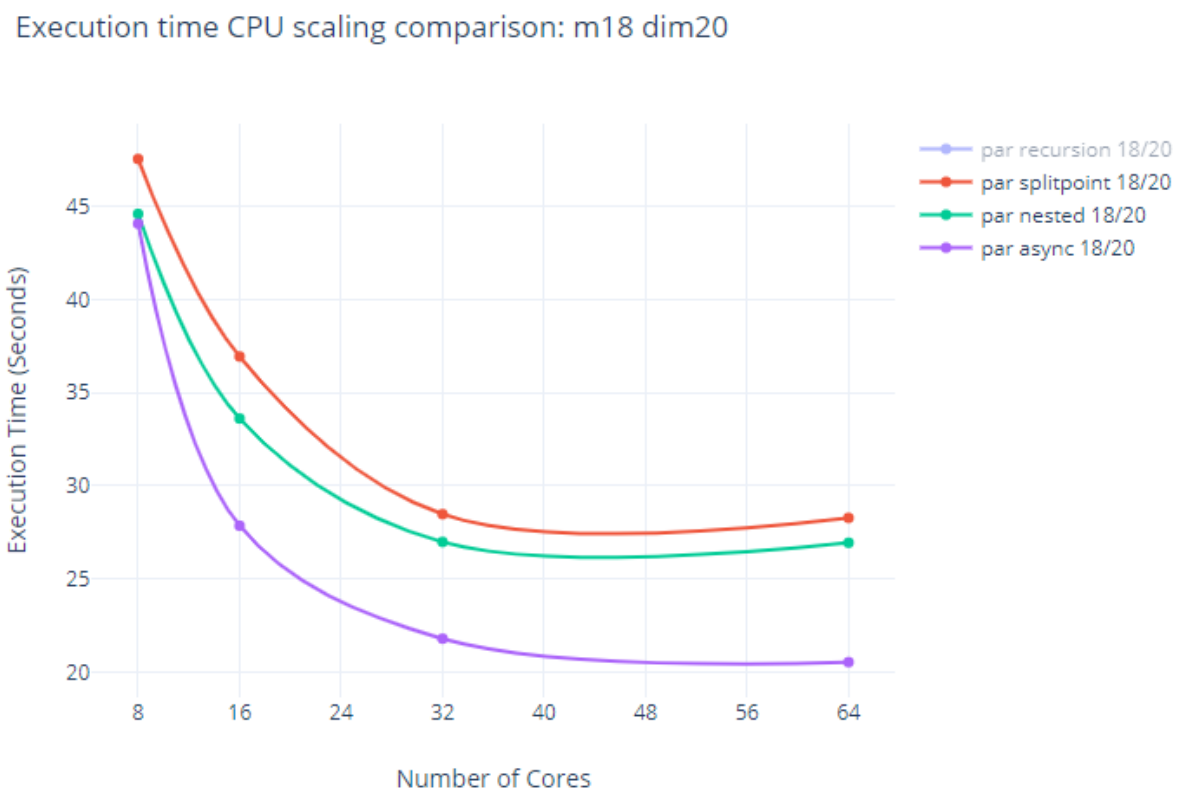


Figure 4.2.3.5: Execution time of three parallel versions, with fixed input size of: `m=18`, `dim=20`. Tested on 8, 16, 32, 64 cores cluster configurations (4 cores per machine).

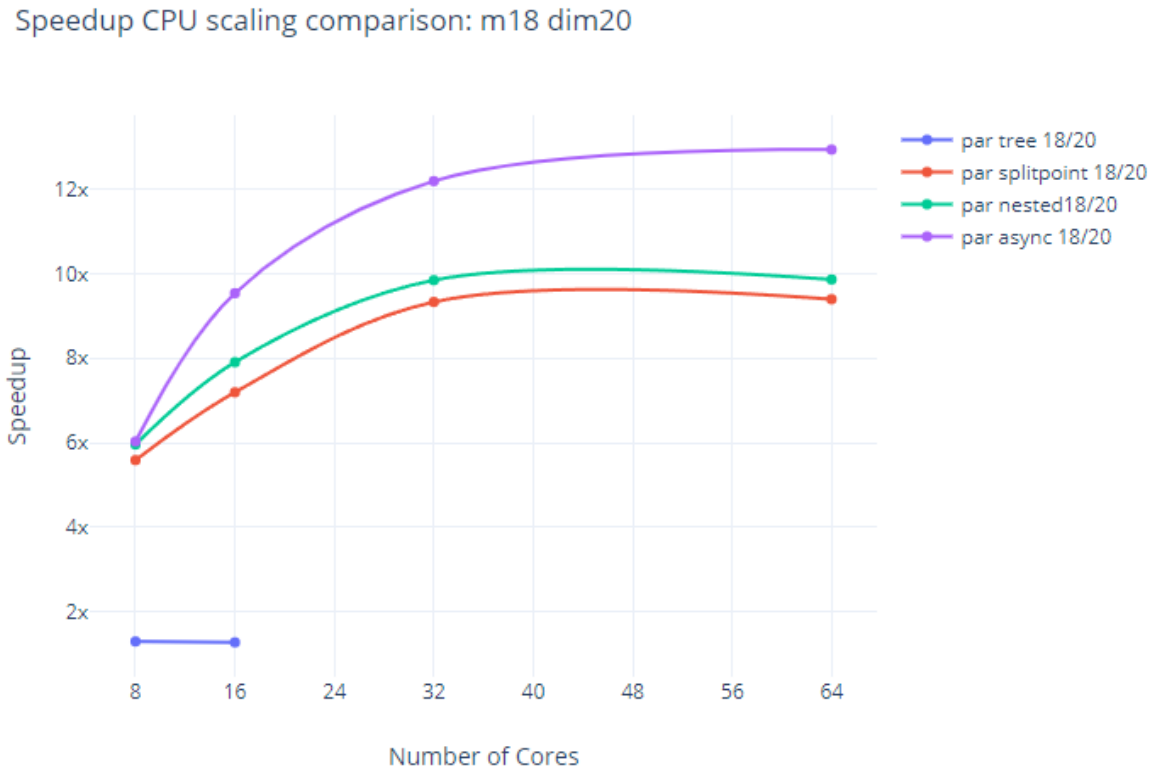


Figure 4.2.3.6: Speedup of the four parallel versions over the optimized sequential one, with fixed input size of: `m=18`, `dim=20`. Tested on 8, 16, 32, 64 cores cluster configurations (4 cores per machine).

Compared with the experiments with 16/20 inputs, the speedup (Figure 4.2.3.6) appears to be slightly higher for all the solutions. The greater load brought by the larger sized array seems to come with a larger execution time percentage in the parallelizable section of the code, making the overall speedup over the sequential version greater.

When looking at the efficiency curves in Figure 4.2.3.7, the async solution is still the best one.

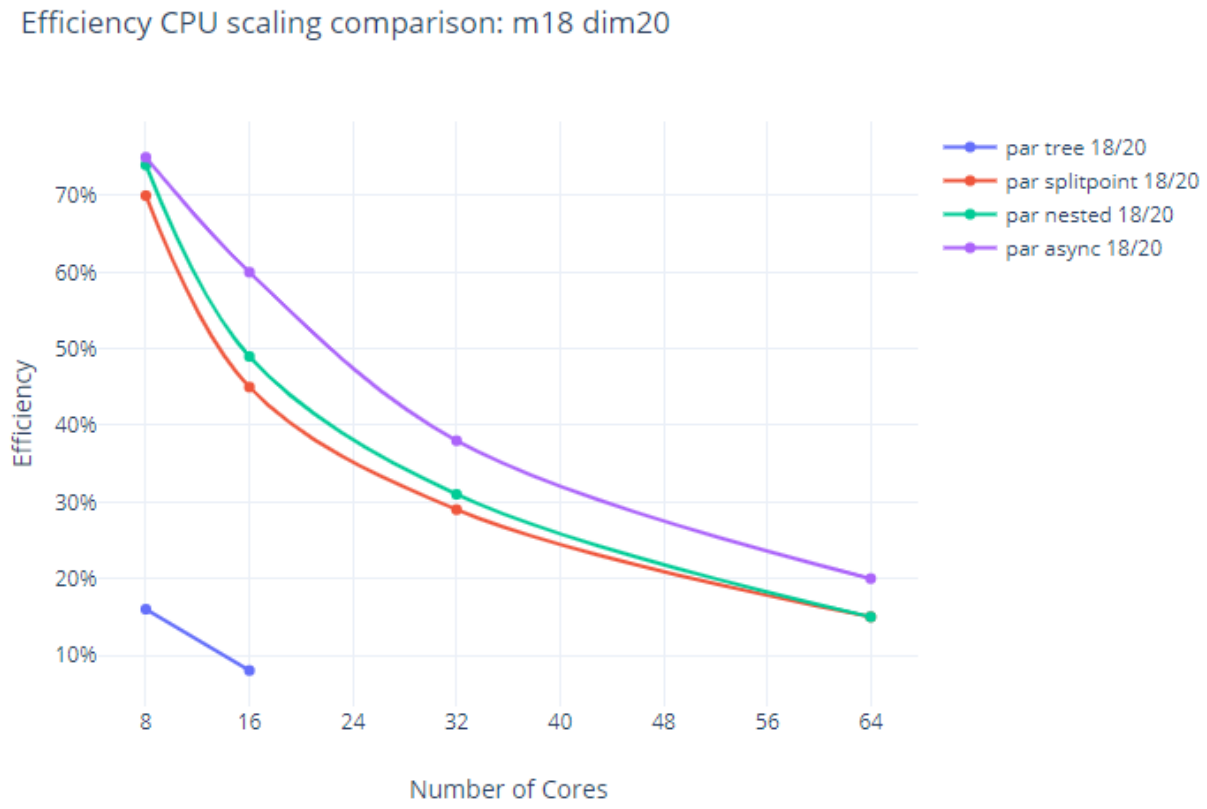


Figure 4.2.3.7: Efficiency of the four parallel versions, with fixed input size of: `m=18`, `dim=20`. Tested on 8, 16, 32, 64 cores cluster configurations (4 cores per machine).

Moving on to the last input combination of `m=18` and `dim=50`, we can observe in Figure 4.2.3.8 how the combined parallel solution, where both the recursion and the split-point computations are parallelized using Ray tasks, seems to be incurring into an early performance limit after 16 cores, where the other solutions seem to instead be able to still improve when provided with more resources, even if with diminishing returns.

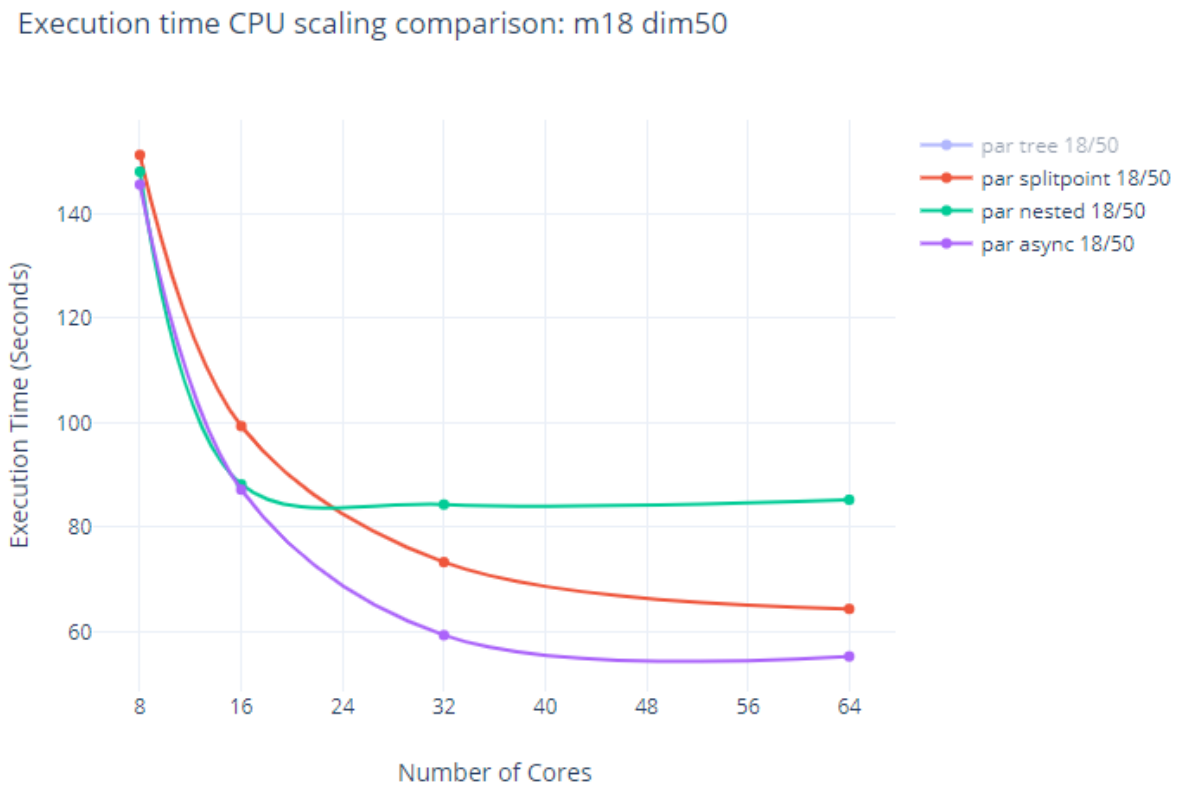


Figure 4.2.3.8: Execution time of three parallel versions, with fixed input size of: `m=18`, `dim=50`. Tested on 8, 16, 32, 64 cores cluster configurations (4 cores per machine).

The speedup graph in Figure 4.2.3.9 allows to gleam the same observation. More interesting are the values of the speedup. The total reached with these inputs is much higher than what previously observed with `dim=20`. The higher parallelism of the workload allows much greater gains.

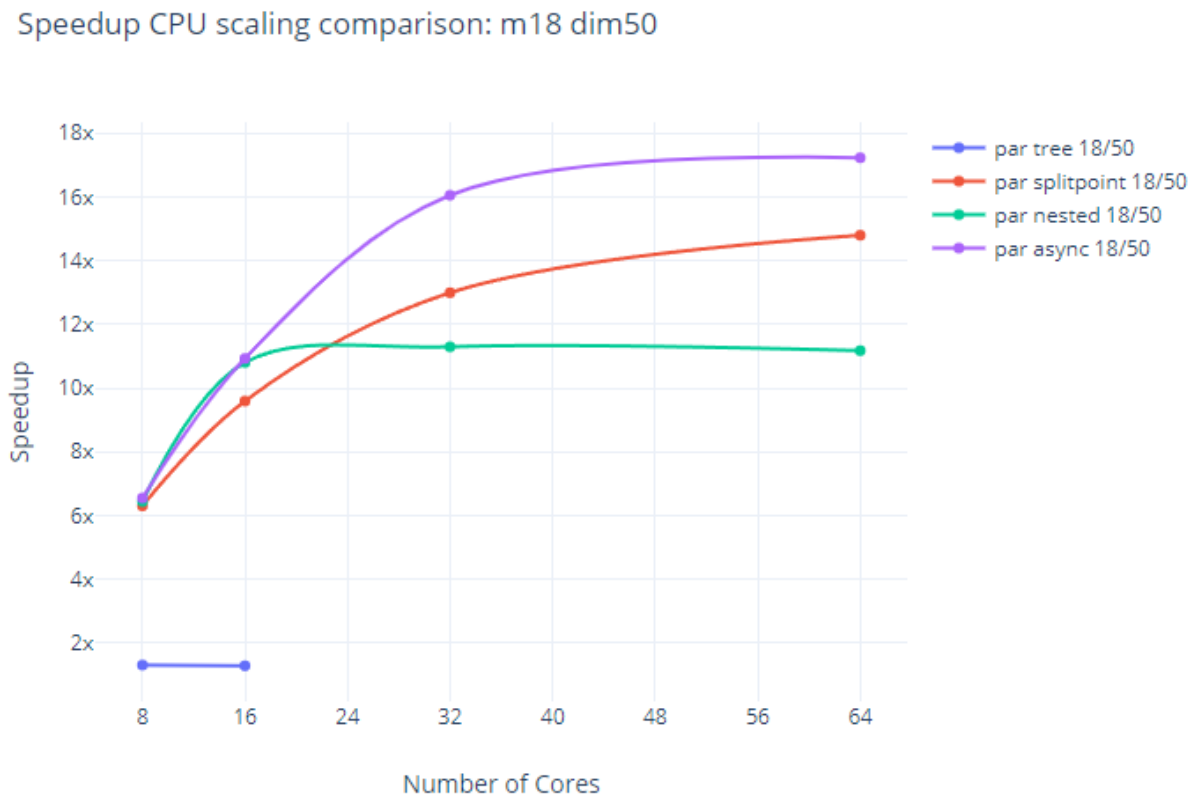


Figure 4.2.3.9: Speedup of the four parallel versions over the optimized sequential one, with fixed input size of: `m=18`, `dim=50`. Tested on 8, 16, 32, 64 cores cluster configurations (4 cores per machine).

The efficiency curves in Figure 4.2.3.10 show a less slanted behaviour from the 18/20 experiment, maintaining overall higher efficiency. In these experiments where the input size doesn't change as the number of cores is increased, the efficiency is expected to decrease as only the parallelism of the system is increased and not the parallelism of the workload. The slope indicates how much the workload can benefit from more resources.

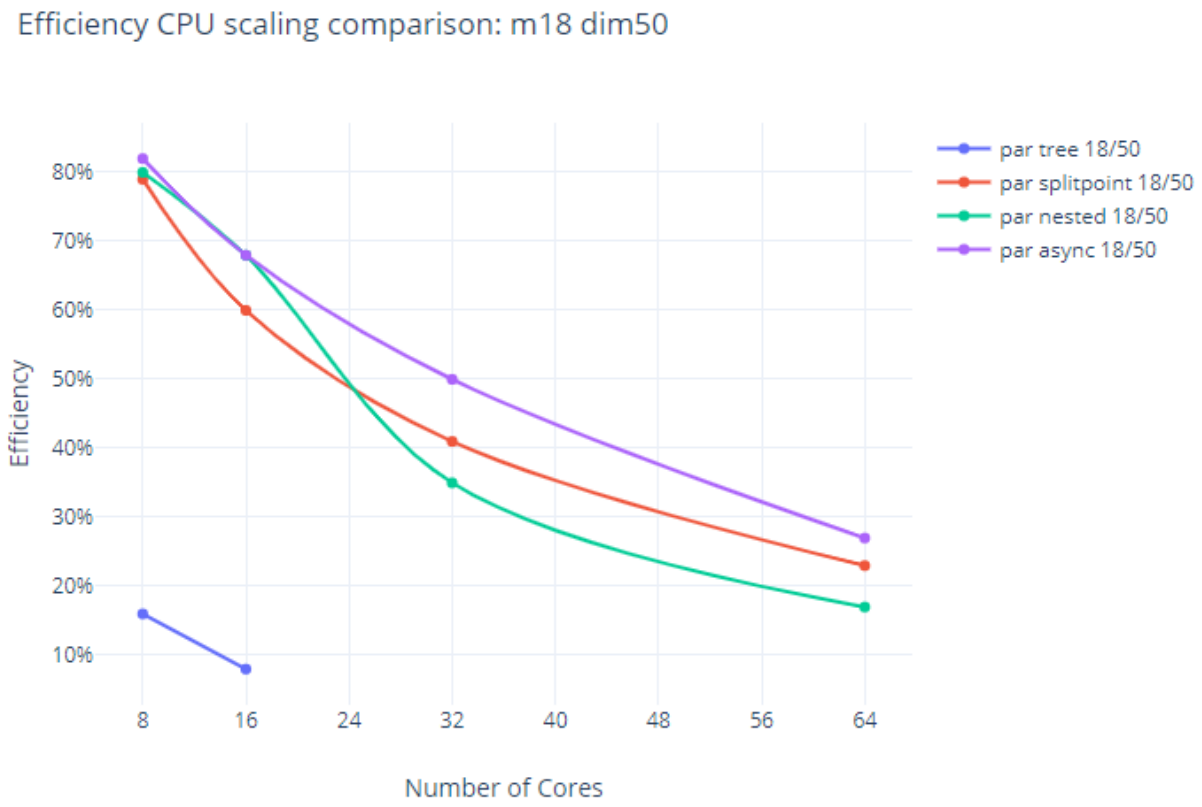


Figure 4.2.3.10: Efficiency of the four parallel versions, with fixed input size of: `m=18`, `dim=50`. Tested on 8, 16, 32, 64 cores cluster configurations (4 cores per machine).

In all the experiments, even though the maximum parallelism of the solutions could exceed the number of cores, the efficiency decreased with the addition of more CPUs. This shows that in none of the solutions the workload is perfectly parallelizable and that all of them incur overhead or have sequential sections that limit the overall efficiency.

The async version distinguished itself as not only the one that was able to scale the best, but also as the solution that performed the best in all experiments. The parallel tree quickly showed itself as inadequate, while the combined nested parallelism solution exposed some worrying scaling behaviours which suggest problems with the solution itself. Parallel split-point solution didn't show any strange behaviour or problem, it simply was invariably outperformed by the async one.

4.3 Considerations on the results

4.3.1 Limitations and shortcomings of results

The results of the solution where both the recursive tree and the split-point computation were parallelized through nested Ray tasks were very disappointing and demonstrated Ray's shortcoming in handling nested parallel tasks.

The mechanism Ray has in place to avoid deadlocks, where workers waiting on remote results are removed from the worker pool and replaced with newly instantiated worker processes, is problematic and inefficient. It introduces delays as new workers are initialised and can quickly lead to unrestricted and unpredictable resource usage as more and more worker processes are generated.

The concurrent asynchronous version was only developed as a workaround to these shortcomings. The final results obtained with it were more than satisfactory and the objective of the project was thoroughly achieved, but it as well has some limitations. In particular, the recursive tree isn't truly parallel.

All the computations, apart from the split-point calculations, are done on a single thread through the async event loop. In cases where multiple events are ready at the same time, one would need to execute before the other, causing a delay. There were some very few examples in the results where the nested combined parallel version outperformed the async one. This could suggest that, if the nested parallel version didn't suffer from such overhead, it could generally perform better than the async solution by virtue of using parallelism instead of concurrency for the binary tree.

`Asyncio` isn't meant to be used for CPU intensive tasks, because the execution of such a task would block the event loop until completed, delaying the execution of all other operations waiting in the queue. In our use case fortunately, the split-point computation measures a very high ratio of the execution time, making the rest of the operations short enough to be executed as async tasks. Larger values of `dim` and `m` could cause async tasks to become longer, but the relative increase of the split-point computation would be much bigger, making any relative overhead smaller.

If the algorithm to be parallelized didn't have the characteristics that it has and instead exhibited a different distribution of workloads, the current solution would not have been viable. If in the case of additional heavy computations outside the split-point one, a solution could be that of moving it to a remote task, to remove it from the async loop. This would be at the cost of data transfers of the parameters and results to and from the remote worker executing the task. In some cases the extraction of the computation could reveal itself to be too complex or even impossible.

So, while our use case was fortunate enough to allow for the asynchronous workaround, it could easily have required a fully nested structure, in which case all the considered solutions found in Python have shown themselves to be disappointing. Ray advertises nested parallelism as a supported feature [27], and while it does work, the way it actually works makes it pretty limited in usefulness.

The main problem is how workers that are blocked waiting on results of child workers are handled. Generating new workers is not a scalable solution. The idea of blocking the worker in the first place should be questioned instead. The solution that is sought here is one that not

only offers performance, but also doesn't require the developers to rewrite the code and its structure to adopt it. Ray actually deserves a lot of merit for embracing async support, allowing the creation of non blocking actors and driver processes.

This support allowed the creation of a non blocking driver process in our async solution, where the process didn't block waiting on results and instead handled other tree nodes or other results through the event loop while remote operations were being performed. What is sorely missed is the ability to do the same thing from within a remote task, allowing workers themselves to become non blocking.

4.3.2 Potential solution: Fully Asynchronous workers

The main difficulty with this approach, and likely the reason why it wasn't already implemented, is not making the worker non blocking, as that is actually possible already, but instead providing the worker with more work from the Ray scheduler. It is already possible to make a worker async, but after spawning child tasks and awaiting on their results, the worker doesn't actually have anything else to do in its event loop. In an ideal solution, when a worker awaits on child tasks, it would be able to obtain new tasks to execute from the Ray scheduler. This would allow the worker to continue to be used for the computation of new tasks, thus avoiding the risk of deadlock and removing the need for the creation of new substitute workers.

There could be issues with this approach though, in particular when a task is interrupted to await a result, the worker hosting it could then be scheduled to handle another task, which could take a long time to complete, potentially not finishing before the results for the original task are ready. The continuation of the original could then suffer a long delay due to the worker being busy processing an unrelated workload, introducing unwanted overhead. This issue can be avoided if the continuation of the task is not required to be executed on the same worker as the one where the task started. This would allow the continuation to be scheduled on other workers, the most likely ones being the ones that have the final results of the awaited operation. Making a continuation of a coroutine mobile in a distributed environment could be challenging.

Research in this space didn't yield many results, even in other languages, but some example of such systems exist for single machine environments: .NET Task Parallel Library [49], Tokio async runtime for Rust [50], Vienna Game Job System for C++20 [51]. These systems use a limited pool of threads to execute async tasks that can themselves generate new tasks, creating nested structures. The tasks are non blocking allowing threads to continue computing other tasks. A frequent characteristic of these systems is the use of a technique called work stealing, to improve the performance of the system.

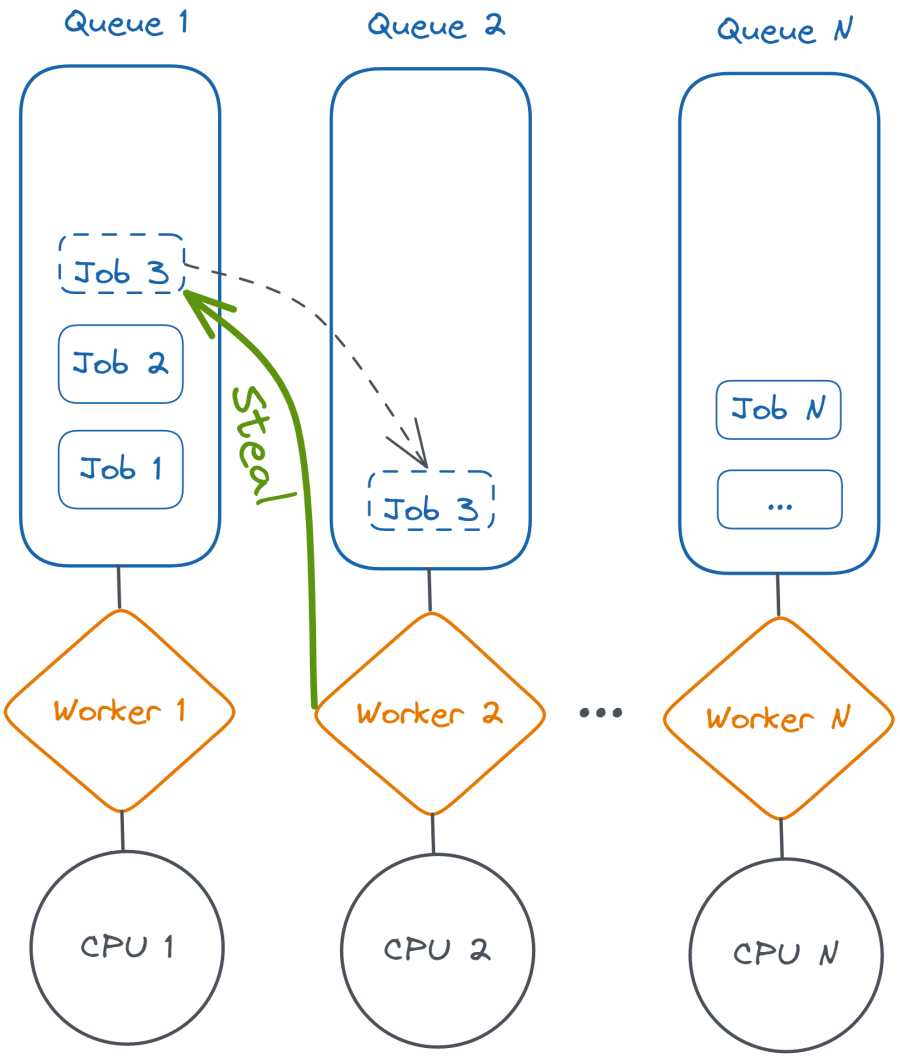


Figure 4.3.2.1: Graphical representation of work stealing in action.

Work stealing is an optimization technique for task scheduling on a pool of workers [54]. Instead of using a centralised scheduler with a single task queue for the entire pool, each worker has its own queue, which is public to the others. A worker will take and compute jobs from one end of its personal queue, for example from the bottom. If a worker runs out of jobs in its queue it will actively search jobs to compute from the queues of other workers that still have a populated queue, and they will steal the jobs from the opposite end of the queue of the victim workers, as shown in Figure 4.3.2.1.

This method avoids single scheduler bottlenecks and single queue contentions permitting efficient handling of unbalanced workloads, since empty workers are able to quickly obtain work from more taxed ones. The method works especially well with nested tasks, as a worker does not need to submit new child tasks to a central scheduler but is instead able to simply place them on its local queue.

While these systems exist, they were not developed for long running CPU intensive tasks. Their main objective is usually the speedup of traditional async workloads, so primarily IO

bound operations. The Tokio async runtime for Rust for example has its main focus on network communication and is commonly used to power web servers. It explicitly mentions in the documentation that it is not advised to be used for CPU bound operations [50]. One could argue that the Vienna Job System is actually meant to be used for CPU bound operations, but it being built in the context of a game engine, means that the majority of the tasks are meant to be completed during a single game frame, so usually less than 16 milliseconds.

In the data science field tasks can be much bigger, as demonstrated from our own use case, where the computation of the best split-point for a single dimension can take several seconds. The usage of these systems in these scenarios is usually discouraged due to the presence of long running tasks potentially reducing responsiveness of smaller tasks, which is usually a critical feature of these systems. In cases where the responsiveness of intermediate tasks isn't critical, and the only interest is the overall execution time, the system should theoretically actually offer good performance. This theory should be appropriately tested and could be part of future developments.

The main advantage of such systems is the ability to write parallel code naturally and simply use async await syntax to express the parallelism without any need for restructuring of the original sequential code.

4.3.3 Potential solution: Rayon library parallelism approach

Another interesting alternative that was identified for handling nested tasks is the system implemented in the *Rayon* Rust library [52]. This framework, contrary to the Tokio runtime [50], is meant to be used to parallelize CPU bound workloads. Its most accessible feature is the ability to easily parallelize for loops, but the more interesting part is its core primitive `join()`.

`Join()` is a blocking primitive that takes two tasks and potentially runs them in parallel [55]. The first one is put on a local queue, while the second one is executed immediately on the current thread. The queued task could be stolen and executed by another worker, or in case all other threads are also busy, be handled by the original thread, sequentially with the other task.

The way `join()` is usually used is through recursion, where a joined task is further split into smaller subtasks. In the example depicted in Figure 4.3.3.1, the sum of the elements of this 16 values array is potentially computed in parallel through the use of `join` operations which split the array into two chunks until there are only two elements to sum. The other chunks of the array could be computed on the same worker, or their task could be stolen by another thread that didn't have other tasks to execute [53].

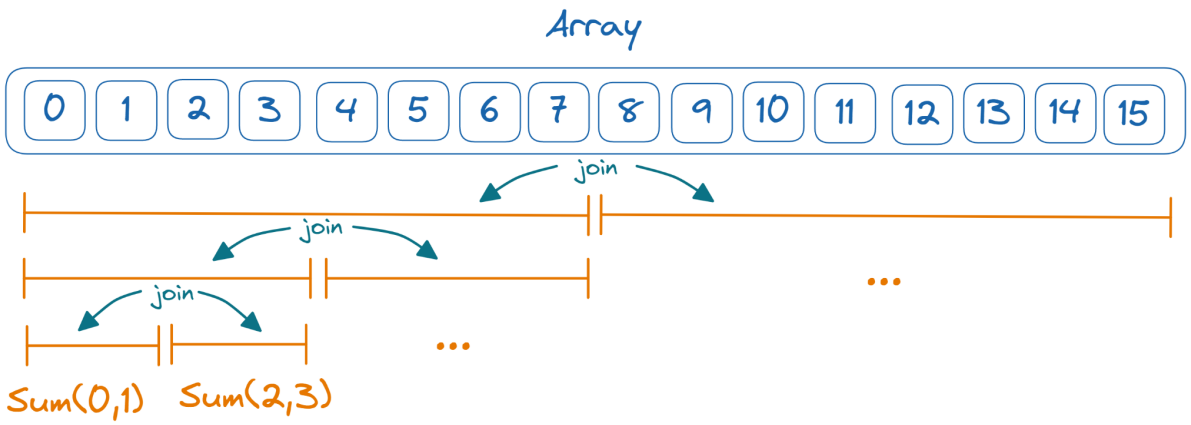


Figure 4.3.3.1: Use of the recursive `join()` primitive to parallelize the element sum of an array.

This recursive joining scheme is the core building block of the Rayon library. In contrast to the other solutions considered so far, this approach does not feature any kind of asynchronous execution as the `join()` primitive is blocking. If two operations are independent and should be executed in parallel if possible, the developer is required to explicitly define them as `join()` tasks. Depending on the structure of the original sequential code, this could in turn require changes and refactors.

In our particular use case though, each point of parallelization is characterised by a split in computation followed by either elaborations of results or no further computations, meaning that the code structure doesn't actually benefit from performing the parallel computations asynchronously. In the case of the parallel split-point computation for each domain dimension, the worker issuing the computation has nothing left to do but wait for the results after dispatching the tasks. The best usage of the worker would be that of contributing to the parallel computation it just issued, which the Rayon library enables.

Furthermore, the natively recursive nature of `join` would enable the seamless parallelization of the recursive tree structure of our use case. When a node of the tree further splits its domain into two child domains, join would allow the elaboration of the child nodes to be parallelizable, but in contrast with an async approach like Ray, part of the computation could continue to be executed on the current worker instead of having to be performed in remote. Figures 4.3.3.2 and 4.3.3.3 illustrate the different execution behaviours of Rayon and Ray libraries.

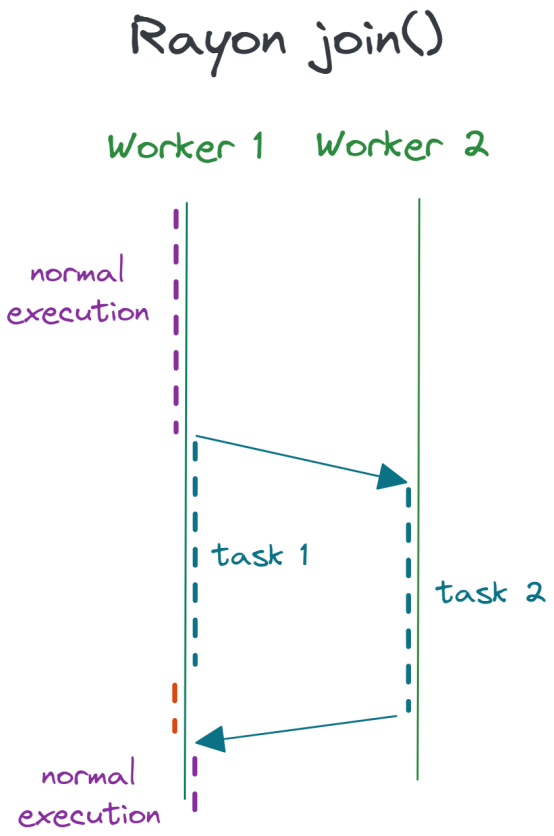


Figure 4.3.3.2: Rayon approach to task scheduling. The parallel task execution is synchronous and the current worker contributes to the parallel computation.

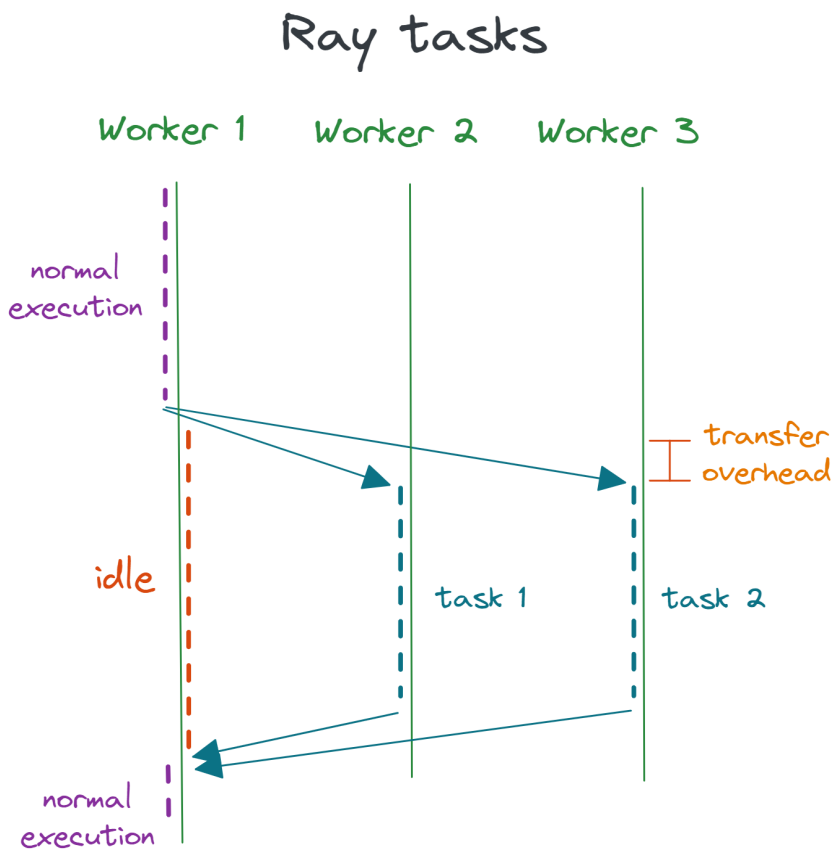


Figure 4.3.3.3: Ray approach to task scheduling. The parallel task execution is asynchronous and the current worker is free to do something else, but it is excluded from participating in the parallel computation.

The Rayon approach is more efficient as it continues to use the original thread of execution to perform computations instead of requiring two other idle workers. The local half of the computation can begin execution immediately, without incurring any data transfer overhead or having to wait for other workers to be free.

In the case of Ray, the task creation is asynchronous, and code following the remote task invocation could be executed on the original execution thread while the parallel tasks are being handled by other workers. But in many use cases, such as ours, there is often no significant other work to be done in addition to the already defined parallel tasks. If the starting point was a sequential script, it is very likely that the code following a computation will require the results to continue. This leaves the original worker with nothing to do but block on the results.

In the Ray example above, if all other workers were to be busy, the two child tasks would not be scheduled, even if the original worker had nothing left to do. The user can express the fact that there is nothing more to be done through the use of the blocking `ray.get()` call. Ray then spawns a new worker to replace the blocked one, which can in turn be used to schedule the remote tasks that were still pending.

So in the case of a saturated system where all other workers are assumed to be busy, if a Ray task generates two child tasks and blocks on the results, the execution of the tasks will only happen after a new worker is spawned and initialised to replace the one blocked on `ray.get`. With the Rayon approach on the other hand, the two tasks can be executed immediately on the original worker with no additional overhead. The problem with Ray is that all the task creations are inherently asynchronous, and there is no way to spawn tasks in a blocking fashion, which would instead allow Ray to use the starting thread as part of the worker pool able to execute the generated tasks.

The performance results obtained with the Ray nested parallelism solution that was implemented for this project, demonstrate how undesirable it is to block Ray tasks with `ray.get` operations, as the generation of new workers has both heavy resource and overhead penalties. To avoid both the generation of new workers and the exclusion of workers from the parallel computation that they generate, it could be considered to split the computation manually so that the original thread could handle a part of it, as to avoid leaving it idle.

This idea has the heavy downside of not allowing the original worker to participate in any kind of dynamic work balancing with the rest of the system. If the work kept on the starting worker is too small, the worker will become idle before the rest of the system and remain so for the rest of the computation. Worse still, if the workload taken is too high, the rest of the parallel computation will complete early and the remaining work will be computed sequentially without making use of the remaining idle workers.

The only downside to the Rayon approach as a tool to parallelize offline data science algorithms, seems to be the missing support for asynchronous operations. The same parallelism result as an async remote task can be achieved by explicitly listing the continuation of the code as another `join()` task, defining explicitly all parallel operations through the `join` primitive. This could require some code restructuring, but in the case of a sequential code starting point, this eventuality isn't actually that high. The case where code requires the results of the previous operations is much more frequent.

Using the Ray building blocks, it could theoretically be possible to build a framework on top of it using asynchronous actors that is able to behave as either a completely async system with non blocking tasks, or a Rayon like one, with primitives that share the same semantics as `join()`. The framework would likely be very complex, incur severe overhead and be incompatible with the normal Ray tasks execution model. It would become an alternative to Ray core without the performance benefits of being implemented directly over a supporting native layer of C++.

An implementation of such a framework would require a large amount of time and any performance benefits that could be gained from using an improved execution model would

almost certainly be negated by the many layers of indirection on top of Ray that would be required to make the new framework function

The issue of Ray's shortcomings in this context was raised with the Ray team, with Rayon's approach as an offered example of an alternative more desiderable system, but the changes that Ray would need to undergo to resolve the problems appear to be too great in scope to be likely to happen. The change of Ray workers to be non asynchronous and non blocking explained before was also suggested, but in this case there could be technical problems limiting the viability of the solution, in particular the possibility of splitting and transferring a continuation of a function coroutine on another machine.

The investigation on the current state of nested parallelism solutions proved to be very interesting and the field still open to many advancements.

4.4 Future Developments

With the results obtained with the testing the async parallel solution was selected as the best parallel distributed implementation and it should be adopted as the default implementation of the GLEAMS algorithm when it is released.

Since the changes are effectively a parallel implementation of the MOB algorithm that is being used by GLEAMS, there are plans to extract the parallel MOB implementation and bundle it as a separate Python package and so make the optimised parallel MOB implementation available for general use. At the time of writing, there are no other Python implementations of the MOB algorithm available.

Another future development would be the parallelization of the `predict()` function of the GLEAMS submodel. Given an array of input points, the function aims to use the pregenerated GLEAMS binary tree of submodels to generate the corresponding result values that the original model would have returned for those same inputs. Currently the function isn't implemented very efficiently and the plan would be to consider the use of parallelization to improve the performance when dealing with a large number of inputs.

For improvements on the solution itself, the biggest one would probably be the support for larger than memory input sizes, and general out of memory computation. The current solution can struggle or even never complete in cases where the executing system doesn't have enough RAM

Extending support for out of memory computation would probably require a lot of changes to switch the numpy arrays currently used to another type that is able to be only partially loaded in from disk. Such a change would also likely have a negative impact on performance. The most likely choices for the implementations would probably either be Ray Datasets [17] or Dask-On-Ray [23].

In another direction, to instead further improve the scalability of the solution and overcome the limited total parallelism achievable, the current workload would need to be further split into many more tasks that take overall less time to complete. Currently this is impossible due to the most computationally intensive part of the algorithm, the calculation of the best split-point, being inherently sequential. Another possible future improvement could be the research and development of an alternative splitting criterion approach that is instead inherently parallelizable.

In the same vein, some considerations could be made towards the use of GPU computing for the acceleration of the computations. While this was considered for the current version of the algorithm and splitting criterion, and was deemed not suitable, an alternative more parallel friendly criterion could benefit from it.

Conclusions

The goal of this work was the implementation of a distributed parallel version of the GLEAMS algorithm, to make it both more performant and able to utilise distributed infrastructure. Further objective was the testing and validation of the solution on both a local and distributed environment. An additional concern was to keep the complexity of the solution as low as practical, ideally maintaining as much of the original code as possible and limiting structural changes.

For those purposes, as part of the work on this project, the sequential algorithm was studied and analysed to identify parallelization opportunities. Many profiling methods were employed to examine the execution profile and determine the hot path and the slowest parts of the code. An important characteristic of the algorithm that was recognized was the presence of the recursive binary tree and its nested structure, which later revealed itself as a particularly difficult parallelization challenge.

Using the information obtained through profiling, it was possible to implement a series of optimizations that were able to significantly improve the execution performance of the sequential algorithm. As a measure to avoid the introduction of any new errors or bugs resulting from the parallelization or optimization efforts, a form of regression testing was put in place to validate the correctness of the changes. Additionally, a benchmark system was designed to efficiently and accurately measure and compare the execution performance of different versions of the algorithm.

For the parallelization, the choice to continue to use Python was justified and an in depth investigation in the space of Python frameworks for distributed parallel execution was performed to identify the best candidate for our specific use case. In particular, both the Dask [19] and Ray [17] frameworks were studied in detail and tests were performed to evaluate their support for nested task invocation. Ray was eventually selected for the superior control it provided for data sharing and distribution, as well as simpler nested invocation semantics.

Using Ray, multiple parallelization approaches were considered, ultimately four distinct solutions were designed and implemented. One which used recursive Ray tasks to only parallelize the binary tree structure computation, another which only parallelized the computation of the split-points along different domain dimensions, the next one merging the previous two and making use of nested tasks. The last one avoided the parallelization of the recursive tree, using an asynchronous approach to make tree node computations potentially concurrent instead of parallel. This last approach was developed due to poor results observed with nested parallel tasks and was only possible with our use case thanks to the majority of the execution time being almost completely concentrated in the computation of the split-point, making the remaining computation lightweight enough to not be blocking the async event loop.

All the developed solutions were subsequently tested and benchmarked with a large variety of problem sizes to inspect their behaviour and compare the performance. Testing was first performed on a single local machine and then on a 98 machine distributed cluster. Various configurations of the cluster were used to evaluate scaling across different numbers of cpu and memory resources. The data from almost 200 separate experiments was analysed and parsed into graphs which showed the best performing solution to be the asynchronous one in most cases.

The results of the parallel versions, where either only the binary tree or the split-point computations were parallelized, were great indicators of the level of parallelism that each of those points could offer. They confirmed the observations that the parallelization points only offered a relatively limited level of total parallelism. In particular the parallelism of the binary tree was noted to suffer specifically due to the tree itself being unbalanced and branches ending early in leaves, leading to low tree breadths and thus lower parallelism.

The limited level of maximum parallelism resulted in diminishing performance returns when scaling beyond certain numbers of available CPU resources. The issue only became apparent for very large problem sizes though, while for normal inputs the computation was quick enough to not need scaling beyond the diminishing returns. Very large problem sizes had other issues, namely running out of memory allocation space. The objective for this project was to improve the performance for the typical use case and not to support extreme cases such as larger than memory problems, which would have required a different approach than the one that was undertaken.

To achieve better scaling, the computation would need to be a lot more segmented, composed of a large number of tasks that take less time to complete. Unfortunately the most computationally intensive part of the algorithm, the calculation of the best split-point, is intrinsically sequential and can’t be parallelized. To better utilise large distributed clusters with many CPU resources, a different splitting criterion approach than the one currently in use would need to be developed. Ideally one that could be parallelized along the length of the input samples arrays, instead of along the dimensions of the sample points.

Testing also made very evident some issues with the way nested parallelism is handled in Ray. In particular the fact that a task waiting on results of further nested tasks would block the worker during the wait, preventing it from being able to handle other work. Ray avoids the risk of deadlock in this situation by instantiating a new worker process to replace the one being blocked, while otherwise the number of workers would remain fixed and equal to the number of CPU cores available. Testing showed how the creation of these new workers could not only add overhead delays in the computations, but also constitute a resource utilisation problem, as memory and other resources could quickly be depleted by hundreds or thousands of workers. The blocking nature of workers was particularly surprising, especially considering how asynchronous conscious Ray seemed to otherwise be.

While the developed asynchronous solution worked well for our use case and more than adequately satisfied the requirements that were set out, the current state of support for nested parallelism scenarios in Python was found to be disappointing. The other frameworks either had similar or worse problems in this regard. Some better theoretical approaches were thus proposed and discussed, inspired by some existing solutions present in other languages.

In particular, the idea of using non blocking workers was considered, inspired by parallel async job systems such as the .NET task parallel library [49], the Tokio async library [50] and the Vienna Game Job System [51]. Some potential downsides and possible technical limitations were raised with this approach, in particular the viability of moving the continuation of an async function to another machine was questioned.

Another proposed solution instead followed the Rayon library approach where the entire paradigm is based around a recursive parallelization primitive called `join()` [52]. Tasks are able to spawn other tasks recursively, each worker having a local task queue and work being balanced through the use of work-stealing. The approach was examined to identify the main differences and advantages over the one currently employed by Ray. The main ones were that the `join()` operation is not asynchronous and the fact that the current worker is included in the computation of the tasks that it spawns. This way there are no workers that are left idle waiting on remote results and any potentially parallel operation can be defined explicitly through the use of the `join()` primitive. Some potential disadvantages were considered, namely the loss of asynchronous operation expressivity, though the same execution behaviour could easily be achieved through the definition of the continuation as a `join()` task. The Rayon approach was considered the most likely to be the best option for a use case such as ours.

To conclude, the goal of this project, to implement a distributed parallel solution, was achieved with success through the use of an asynchronous approach in conjunction with Ray tasks. The total parallelism attainable can be limited depending on the properties of the black-box model considered, due to the most computationally intensive part of the algorithm being inherently sequential. Nevertheless, the performance obtained is more than satisfactory for all the normal expected use cases of GLEAMS.

List of Figures

List of Tables

Bibliography

[1] V. M. Stanzione (2022) “Developing a new approach for machine learning explainability combining local and global model-agnostic approaches”. [Master Thesis], University of Bologna, Computer Engineering [LM-DM270], http://amslaurea.unibo.it/25480/.

[2] M. T. Ribeiro, S. Singh, and C. Guestrin, “Why should I trust you?”: Explaining the predictions of any classifier, 2016. arXiv: 1602 . 04938 [cs.LG].

[3] F. Kuo and S. Joe, “Sobol sequence generator: Primitive polynomials and direction numbers”. [Online]. Available: https://web.maths.unsw.edu. au/~fkuo/sobol/.

[4] A. Zeileis, T. Hothorn, and K. Hornik, “Model-based recursive partitioning”, Journal of Computational and Graphical Statistics, vol. 17, no. 2, pp. 492–514, 2008. doi: 10.1198/106186008x319331.

[5] Sklearn Estimators, Documentation Reference. [Online]. Available: https://scikit-learn.org/stable/developers/develop.html#estimators.

[6] Python Speed, Documentation Reference. [Online]. Available: https://wiki.python.org/moin/PythonSpeed.

[7] Python GIL, Documentation Reference. [Online]. Available: https://docs.python.org/3/glossary.html#term-global-interpreter-lock.

[8] Python Extensions, Documentation Reference. [Online]. Available: https://docs.python.org/3/extending/index.html.

[9] Python C/C++ Extension Interface, Documentation Reference. [Online]. Available: https://docs.python.org/3/extending/extending.html.

[10] Rust Programming Language, Official Site. [Online]. Available: https://www.rust-lang.org/.

[11] Bjarne Stroustrup. 2013. The C++ Programming Language (4th. ed.). Addison-Wesley Professional.

[12] Rust Ownership Model, Documentation Reference. [Online]. Available: https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html.

[13] OpenMP, Documentation Reference. [Online]. Available: https://www.openmp.org/specifications/.

[14] Open MPI, Documentation Reference. [Online]. Available: https://www.open-mpi.org/.

[15] Numba, Documentation Reference. [Online]. Available: https://numba.pydata.org/.

[16] Spark, Documentation Reference. [Online]. Available: https://spark.apache.org/docs/latest/index.html.

[17] Ray, Documentation Reference. [Online]. Available: https://docs.ray.io/en/latest/.

[18] Ray Core, Documentation Reference. [Online]. Available: https://docs.ray.io/en/latest/ray-core/walkthrough.html.

[19] Dask, Documentation Reference. [Online]. Available: https://docs.dask.org/en/stable/.

[20] V. Sònia, "A Comparative Analysis of the Dask and Ray Libraries", Sep 2021. [Online]. Available: https://www.clearpeaks.com/a-comparative-analysis-of-the-dask-and-ray-libraries/.

[21] A. Puurula, "Benchmarking Python Distributed AI Backends with Wordbatch", Jun 2019. [Online]. Available: https://towardsdatascience.com/benchmarking-python-distributed-ai-backends-with-wordbatch-9872457b785c.

[22] S. Wang, "Analyzing memory management and performance in Dask-on-Ray", Jun 2021. [Online]. Available: https://www.anyscale.com/blog/analyzing-memory-management-and-performance-in-dask-on-ray.

[23] Dask-on-Ray, Documentation Reference. [Online]. Available: https://docs.ray.io/en/latest/data/dask-on-ray.html.

[24] Dask Futures, Documentation Reference. [Online]. Available: https://docs.dask.org/en/stable/futures.html.

[25] Python `concurrent.futures` API, Documentation Reference. [Online]. Available:

https://docs.python.org/3/library/concurrent.futures.html.

[26] Dask, “Submit Tasks from Tasks”, Documentation Reference. [Online]. Available: https://docs.dask.org/en/stable/futures.html#submit-tasks-from-tasks.

[27] Ray, “Pattern: Using nested tasks to achieve nested parallelism”, Documentation Reference. [Online]. Available: https://docs.ray.io/en/latest/ray-core/patterns/nested-tasks.html.

[28] Dask Data Locality, Documentation Reference. [Online]. Available: https://distributed.dask.org/en/stable/locality.html.

[29] Ray, “Anti-pattern: Passing the same large argument by value repeatedly harms performance”, Documentation Reference. [Online]. Available: https://docs.ray.io/en/latest/ray-core/patterns/pass-large-arg-by-value.html.

[30] Python, “The Python Profilers”, Documentation Reference. [Online]. Available: https://docs.python.org/3/library/profile.html.

[31] `gprof2dot,` Documentation Reference. [Online]. Available: https://github.com/jrfonseca/gprof2dot.

[32] `line_profiler`, Documentation Reference. [Online]. Available: https://github.com/pyutils/line_profiler.

[33] `pprofile`, Documentation Reference. [Online]. Available: https://github.com/vpelletier/pprofile.

[34] `austin`, Documentation Reference. [Online]. Available: https://github.com/P403n1x87/austin.

[35] Austin VS Code, Visual Studio Code Marketplace. [Online]. Available: https://marketplace.visualstudio.com/items?itemName=p403n1x87.austin-vscode.

[36] `memory_profiler`, Documentation Reference. [Online]. Available: https://github.com/pythonprofilers/memory_profiler.

[37] `debugpy`, Documentation Reference. [Online]. Available: https://github.com/microsoft/debugpy.

[38] J. Friedman, “Multivariate adaptive regression splines”, The Annals of Statistics 19 (1), pages 1-67, 1991.

[39] L. Breiman, “Bagging predictors”, Machine Learning 24, pages 123-140, 1996.

[40] `sklearn.datasets.make_friedman1`, Documentation Reference. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_friedman1.html.

[41] `tqdm`, Documentation Reference. [Online]. Available: https://tqdm.github.io/.

[42] G. Cousineau, M. Mauny. “The Functional Approach to Programming”. Cambridge, UK: Cambridge University Press, 1998.

[43] Python `asincio`, Documentation Reference. [Online]. Available: https://docs.python.org/3/library/asyncio.html.

[44] Ray, “High wait_for_function overhead for new tasks” issue discussion. [Online]. Available: https://discuss.ray.io/t/high-wait-for-function-overhead-for-new-tasks/7905/3.

[45] Node JS, Documentation Reference. [Online]. Available: https://nodejs.org/en/docs/.

[46] Python Executors Interface, Documentation Reference. [Online]. Available: https://docs.python.org/3/library/concurrent.futures.html#executor-objects.

[47] Ray, “Is it possible to use Ray in a subprocess created with multiprocessing.Process?”. [Online]. Available: https://discuss.ray.io/t/is-it-possible-to-use-ray-in-a-subprocess-created-with-multiprocessing-process/7966.

[48] Mark D. Hill. 1990. “What is scalability?”. SIGARCH Comput. Archit. News 18, 4 (Dec. 1990), 18–21. doi: 10.1145/121973.121975.

[49] .NET, Task Parallel Library (TPL), Documentation Reference. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/standard/parallel-programming/task-parallel-library-tpl.

[50] Rust, `Tokio`, Documentation Reference. [Online]. Available: https://tokio.rs/tokio/tutorial.

[51] C++, Vienna Game Job System (H. Hlavacs), Documentation Reference. [Online]. Available:

https://github.com/hlavacs/ViennaGameJobSystem.

[52] Rust, `Rayon` Documentation Reference. [Online]. Available: https://docs.rs/rayon/latest/rayon/.

[53] N. Matsakis, “Rayon: data parallelism in Rust”, Dec 2015. [Online]. Available: https://smallcultfollowing.com/babysteps/blog/2015/12/18/rayon-data-parallelism-in-rust/.

[54] R. Blumofe, C. Leiserson, (1999). "Scheduling multithreaded computations by work stealing". J ACM. 46 (5): 720–748. doi:10.1145/324133.324234. S2CID 5428476.

[55] Rayon, `join()`, Documentation Reference. [Online]. Available: https://docs.rs/rayon/latest/rayon/fn.join.html.

[56] Numpy library, Documentation Reference. [Online]. Available: https://numpy.org/doc/stable/.