POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Parallel Monte Carlo Tree Search for Index Optimization

Advanced Methods For Scientific Computing

Authors: **Kamil Hanna**

Professor: Luca Formaggia
Academic Year: 2024-25
Release Date: Feb 18^{th} 2025
Version: 1.0.0

Contents

1. Introduction

Monte Carlo Tree Search (MCTS) is a powerful algorithm for decision-making in complex environments [**kocsis2006bandit**]. It has been widely applied in various domains, including games, robotics and optimization problems [**browne2012survey**]. MCTS combines the precision of tree search with the generality of Monte Carlo simulations, enabling it to efficiently explore and evaluate potential actions without requiring a complete enumeration of the search space. The algorithm proceeds iteratively through four main stages:

- **Selection**: Traverse the tree from the root node to a leaf node using a selection policy (e.g., UCB1) that balances exploration and exploitation.
- **Expansion**: Expand the tree by adding one or more child nodes to the selected leaf node, representing possible actions or states.
- **Simulation**: Perform a Monte Carlo simulation from the newly added node(s) to estimate the potential outcome of the selected actions.
- **Backpropagation**: Update the tree with the results of the simulation, propagating the rewards back to the root node to refine future decisions.

Objective : In this paper, we present our application-specific parallelized and optimized implementation of the Monte Carlo Tree Search Algorithm (MCTS). The implemented algorithm is designed for index optimization (Portfolio optimization), specifically targeting the S&P 500 index which is use case demonstrated in our code.

2. Repository Structure

Our project's repository can be found at https://github.com/KamilHanna/MonteCarloTreeSearch In the main folder of the repository you can find a detailed README.md file that further explains concepts that we presented in this report. The code is organized into 3 sub folders:

2.1. Header Files

The project includes the following header files, each serving a specific purpose:

- `Asset.hpp` - Defines the `Asset` object for storing asset information.
- `Portfolio.hpp` - Defines the `Portfolio` object that manages assets.
- `Constants.hpp` - Defines constant values used across the portfolio optimization process, such as the risk-free rate, risk aversion, and exploration constant for MCTS (UCB1). These constants help control parameters like risk preference and exploration/exploitation balance during portfolio optimization.
- `Constraints.hpp` - Defines portfolio constraints, including sector weight limits, asset weight boundaries, and sector index boundaries for the S&P 500, to guide the portfolio optimization process.
- `Types.hpp` - Defines fundamental types used in the project.
- `Utils.hpp` - Implements utility functions for file reading, user input handling, and other helper tasks in portfolio optimization.
- `Node.hpp` - Defines the data structure for nodes in the MCTS algorithm.
- `MCTS.hpp` - Implements the core Monte Carlo Tree Search algorithm, including the methods for node selection, expansion, simulation, and backpropagation.
- `Logger.hpp` - Implements the Terminal I/O interface functionalities.

2.2. Source Files

The `src` directory contains the implementation of the header files described above. Each source file corresponds to its respective header file, providing the necessary functionality for the project.

2.3. Python Libraries

The `Python` directory includes Python libraries that generate the assets data. These libraries are responsible for creating and preprocessing the data used in the portfolio optimization process.

3. Data Preparation Using Python

The data used in this project is generated using Python libraries. In order to create a portfolio/index, we must gather assets and their data! In our project we focus on optimizing the S&P 500, therefore we must collect all the stocks included in the index with their related data.
Although many websites contain data on stocks or indices with a list of its stocks data, a Python interface was built to collect the stocks data. This was due to many reasons, first, all most of these free websites contain outdated data, while the ones with the newest data require you to pay for a subscription in order to access the information. Moreover, the built-in interface lets a user create his own test cases (creation of new indices) by retrieving data for a specific list of chosen stocks, which is very convenient. In the end, most of the "handy data" such as weights and other metrics used by indices makers are hidden from the public, so the best way is to really try to replicate them manually.

In particular our interface was built on top of the existing yfinance python API, due to the fact that yfinance API is free to use and has no limit on the number of requests to the Yahoo servers. Our interface consists of a library called `FinanceLib.py` which includes multiple functionalities responsible for querying for data. The most relevant functionalities are listed in the table below :

Function Name	**Description**
`initialize_stocks(filename, _period)`	Initializes stock data by invoking all the data querying functions and saves them into files.
`get_current_price(df)`	Retrieves the current price of each stock.
`get_weights(df)`	Calculates stock weights based on market capitalization for each stock.
`get_expected_returns_and_risk(df, _period)`	Calculates expected returns and risk (standard deviation) for each stock over a specified period.
`get_variance(returns, expreturn)`	Calculates the variance of a set of returns given the expected return.
`get_correlations(df_returns)`	Computes the correlation matrix for a given set of stock returns.

In the following paragraphs, a small overview of the most important functions is presented, in addition to other meaningful functionalities whom are invoked within these functions.

Expected Return and Risk The function `get_expected_returns_and_risk` calculates the expected return and risk for each stock. For each stock, historical price data is fetched using the `yfinance` library. The percentage returns are computed as:

$$\text{Returns} = \frac{\text{Close} - \text{Open}}{\text{Open}} \times 100.$$

The expected return is the mean of these returns, and the risk (standard deviation) is derived by first computing the variance of the returns and then taking its square root. The values are then stored in the Stocks dataframe.

Correlation Matrix The function `get_correlations` computes the correlation matrix for all pairs of stocks in the dataset. For each pair of stocks, their return series are extracted, and the covariance between them is calculated as:

$$\text{Covariance}(X, Y) = \frac{1}{n} \sum_{i=1}^{n} (X_i - \bar{X})(Y_i - \bar{Y}).$$

The correlation coefficient is then derived by dividing the covariance by the product of the standard deviations of the two return series:

$$\text{Correlation}(X, Y) = \frac{\text{Covariance}(X, Y)}{\sigma_X \cdot \sigma_Y}.$$

This value is stored in the correlation matrix dataframe.

Helper Functions The helper functions `get_variance`, `get_variance2`, and `get_covariance` support the main calculations. The variance of a return series is computed as:

$$\text{Variance} = \frac{1}{n} \sum_{i=1}^{n} (R_i - \bar{R})^2.$$

The covariance between two return series is calculated as:

$$\text{Covariance}(X, Y) = \frac{1}{n} \sum_{i=1}^{n} (X_i - \bar{X})(Y_i - \bar{Y}).$$

These functions ensure that the calculations for risk and correlations are accurate and efficient.

At the end of the initialization, we would obtain 2 files. The first file **Stocks.csv** which includes the stocks Symbol (or name), the sector, the expected return, the risk , the current price and finally the weight. The second file **correlations_matrix.csv** which includes the computed correlation matrix for all the stock returns, providing a comprehensive view of the relationships between different stocks in the portfolio.

4. Foundation Classes

This section describes the C++ implementation of our algorithm foundation classes. We begin, by discussing the core components of our implementation which include both the **Asset** and the **Portfolio** classes that handles storing the data with other related files that govern the optimization process.

4.1. Asset Class

The `Asset` class was designed to store and manage information about individual stocks from the S&P 500 index. Each instance of the `Asset` class represents a single stock and encapsulates its key attributes, including its name, current price, expected return, risk (standard deviation of returns), asset class, and correlations with other assets. This class serves as the building block for constructing and manipulating single stocks.

The `Asset` class includes the following private attributes:

- `name`: The name of the asset (e.g., the stock ticker symbol).
- `assetClass`: The category or sector to which the asset belongs (e.g., Technology, Healthcare).
- `currentPrice`: The current market price of the asset.
- `expectedReturn`: The expected return of the asset, calculated as the mean of historical returns.
- `risk`: The standard deviation of the asset's returns, representing its volatility.
- `correlations`: A vector storing the correlation coefficients between this asset and all other assets in the portfolio.
- `numberOfAssets`: A static counter tracking the total number of `Asset` instances created.

In practice, assets are assigned weights separately, then combined together in a bucket, which we call a Portfolio.

4.2. Portfolio Class

A `Portfolio` Class, was implemented in order to construct and store portfolios/indices (collection of assets/stocks). Our portfolio class encapsulates a set of assets (stocks) and their corresponding weights. Below, we describe the `Portfolio` class attributes :

- `assets`: A vector of `Asset` objects representing the stocks in the portfolio.
- `weights`: A vector of weights corresponding to each asset in the portfolio. The weights determine the proportion of each asset in the portfolio.

The `portfolio` class, is the main character in our algorithm. It includes all the metrics and formulas that will be used for taking optimization decisions, in addition to the possible optimization functionalities themselves.

4.2.1. Constructing a Portfolio

Since our `Portfolio` class plays a critical role in the Monte Carlo Tree Search (MCTS) algorithm, it was built with a focus on high efficiency, performance and memory management.

```
//Constructor
template <typename T>
Portfolio<T>::Portfolio( vector<Asset>&& assets, vector<T>&& weights)
    assets(move(assets)),weights(move(weights)){}
```

Figure 1: Portfolio Class Constructor

Constructing a portfolio involves moving large amounts of data, as each portfolio is represented by a collection of assets and their corresponding weights. For this reason we involve move semantics in its constructor, for handling efficient data movement. This approach avoids unnecessary copying of data, which is also very crucial in our MCTS algorithm implementation especially during the expansion phase. The expansion phase in the MCTS involves adding child nodes to the root, which implies that new portfolios are created. These move semantics help by ensuring our data is transferred efficiently without duplicating large vectors. This optimization here also enables scalability and the capability of using large datasets.

4.2.2. Portfolio metrics

In this section, we discuss the metrics computed in the portfolio class. These metrics are described in the table below.

Metric Methods	Description
`computePortfolioWAP()`	Computes the Weighted Average Price (WAP) of the portfolio. It multiplies each asset's current price by its weight and sums the results.
`computeExpectedReturn()`	Computes the expected return of the portfolio. It multiplies each asset's expected return by its weight and sums the results.
`computeVarianceRisk()`	Computes the portfolio's risk as the variance of returns. It accounts for the individual risks of assets and their correlations.
`computeVolatilityRisk()`	Computes the portfolio's volatility (standard deviation of returns) by taking the square root of the variance.
`computeSharpeRatio()`	Computes the Sharpe ratio of the portfolio, which measures the risk-adjusted return. It subtracts the risk-free rate from the expected return and divides by the portfolio's volatility.
`computeAnnualizedReturn()`	Computes the annualized return of the portfolio by converting the daily return to an annualized figure using the number of trading days.
`computeAnnualizedVolatility()`	Computes the annualized volatility of the portfolio by scaling the daily volatility by the square root of the number of trading days.

Table 1: Metric functionalities in the `Portfolio` Class

Most of these metrics are computed using data from the assets, which are generated by the python interface mentioned earlier. All of these functions are designed in an efficient way via the use of iterators and good handling of data. These metrics depend also on some application-specific user-defined variables from the both **Contants.hpp** and the **Constraints.hpp** files. These two files include important governing constants and constraints that guide our optimization process. Some examples from the **Contants.hpp** file include constants such as *tradingdays, riskaversion* and *riskfreerate*. Regarding the **Constraints.hpp** file, it divides our portfolio into sectors in a ghosted manner. The file includes important constraints for our optimization such as sector weights constraints, with their boundaries, Min/Max asset weights and Min/Max weight adjustment variables which will be discussed in details in the next section. It is of high importance also to mention the there are two objective behind these computed metrics. First, to set a compute the reward value for the nodes in the MCTS and second to guide the optimization functionalities which we call **actions**. These **actions** will be discussed in their deep details in the next section, as they are the core of the MCTS simulation phase.

4.2.3. Portfolio Action Space Functionalities

The `Portfolio` class includes three action functions (`Action1`, `Action2`, and `Action3`). Action's are basically functions that perform some adjustments to the portfolio asset weights using the other metrics related functionalities. These actions form the **Action Space**, which is used in the Monte Carlo Tree Search algorithm in the Simulation stage. Our portfolio class includes three types of different actions. The first two action represents weights modifications based on return and risk respectively, whereas the third one represents weights modifications based on correlations between the stocks. In our implementation later-on, we will see how the first two actions are introduced in the first phase of the Monte Carlo Tree Search, while the third one is introduced in the second phase, which I call fine-tuning. It is important to note that some of these actions perform sector based adjustments, while others perform overall portfolio adjustments. These design decisions are crucial for our MCTS implementation, since they are the ones that makes it highly efficient for Portfolio optimization. In the next paragraphs, a detailed explanation of each action is presented. Note, the actions rely on some of the Min/Max asset weight and adjustment constraints from our **Constraints.hpp** file.

Action Space Functionalities	**Description**
`Action1(adjustment_value, sector)`	Adjusts asset weights gradually based on the returns of the assets within a specific sector. Assets with higher returns are increased in weight, while those with lower returns are decreased.
`Action2(adjustment_value, sector)`	Adjusts asset weights gradually based on the risk (volatility) of the assets within a specific sector. Assets with lower risk are increased in weight, while those with higher risk are decreased.
`Action3(adjustment_value)`	Adjusts asset weights across the entire portfolio based on the correlations between assets. Assets with low correlations are increased in weight, while those with high correlations are decreased.

Table 2: Action Functions in the `Portfolio` Class

Action 1: Adjusting Weights Based on Returns The `Action1` function adjusts asset weights within a specific sector based on their expected returns. First, the average return of the sector is computed. For each asset in the sector, the distance between its return and the sector average is calculated. Assets with returns above the average are increased in weight, while those below the average are decreased. The adjustment is scaled proportionally to the distance from the average, ensuring a gradual and balanced change. After adjustments, the weights are normalized to respect the sector's total weight constraints.

Action 2: Adjusting Weights Based on Risk The `Action2` function adjusts asset weights within a specific sector based on their risk (volatility). The average risk of the sector is computed, and the distance between each asset's risk and the sector average is calculated. Assets with risk below the average are increased in weight, while those above the average are decreased. The adjustment is scaled proportionally to the distance from the average, ensuring a gradual and balanced change. After adjustments, the weights are normalized to respect the sector's total weight constraints.

Action 3: Adjusting Weights Based on Correlations The `Action3` function adjusts asset weights across the entire portfolio based on their correlations. For each pair of assets, the correlation coefficient is compared to a threshold (0.5). Assets with low correlations (below the threshold) are increased in weight, while those with high correlations (above the threshold) are decreased. The adjustment is scaled proportionally to the distance from the threshold, ensuring a gradual and balanced change. After adjustments, the weights are normalized to respect the sector's total weight constraints.

You might notice that a normalization layer is applied after every action, and this is very crucial to keep the weights sum equal to 1, in addition to sticking to sector constraints which are of significant importance in portfolio building. A small message to be mention here is that, after experiencing with the algorithm, we noticed that the first two actions were achieving good adjustments on the portfolio leading to good results, whereas the third action had negative effects on the portfolio's performance.

4.2.4. Portfolio Weights Functionalities

Our `Portfolio` class includes multiple functionalities on weights, also implemented efficiently using iterators and proper data management. Below we present these functionalities and their purpose :

Function	Description
`printWeights()`	Prints the weights of all assets in the portfolio to the console. Each weight is displayed sequentially, followed by a space.
`printWeightsToFile(filename)`	Writes the weights of all assets in the portfolio to a specified file. Each weight is written on a new line.
`initializeWeights()`	Initializes the weights of all assets in the portfolio to 1.0 and then normalizes them to respect sector-specific weight constraints. This ensures that the total weight of assets within each sector does not exceed the maximum allowed weight for that sector.
`normalizeWeights()`	Normalizes the weights of all assets in the portfolio so that they sum to 1. This ensures that the portfolio is properly scaled and ready for further analysis or optimization.

Table 3: Functions for Weight Management and Output in the `Portfolio` Class

IN our portfolio optimization application, weight initialization is very crucial, that is why we give the user two choices for weights initialization. The first choice is using the weights computed based on the market cap of each stock, in other words the ones available in the returned file from the python interface. The second choice is sector-equal initialization with overall sum equal to 1. The last, is of significance for us when it comes to testing the performance of our algorithm.

Finally, one more important file to mention is the **Types.hpp** which includes the data types used across the whole project implementation. These variable types can be modified by a user depending on the intended accuracy and the precision of the floating point of the data used in the project.

5. The Monte Carlo Tree Search

This section describes the C++ implementation of the Monte Carlo Tree Search for portfolio optimization. We describe all the foundations for the algorithm, the parallelization and optimizations done to the algorithm. We begin by describing the `Node` class, then move into the details of the MCTS with its optimizations and proper utilization of the algorithm and end it up with he parallelism.

5.1. Node Class

The `Node` class is a fundamental component of the Monte Carlo Tree Search (MCTS) algorithm. Each instance of the `Node` class represents a state in the search tree, encapsulating a portfolio, its associated metrics, and connections to child nodes. The class is designed to facilitate efficient tree traversal, reward accumulation, and decision-making during the MCTS process.

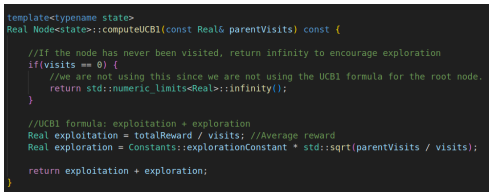
The `Node` class includes the following attributes:

- `portfolio`: Represents the state of the portfolio associated with the node. This is the primary data structure used for decision-making and evaluation.
- `visits`: Tracks the number of times the node has been visited during the MCTS process. This is used to balance exploration and exploitation.
- `totalReward`: Stores the cumulative reward obtained from simulations passing through this node. This is used to evaluate the node's performance.
- `children`: A vector of shared pointers to child nodes. Using `shared_ptr` ensures automatic memory management and proper ownership of child nodes.
- `numberOfNodes`: A static counter that tracks the total number of nodes created. This is shared across all instances of the `Node` class.

```
//Constructor
template <typename state>
Node<state>::Node(state &&portfolio) : portfolio(move(portfolio)) {
    numberOfNodes++;
}
```

Figure 2: Node Class Constructor

A small discussion about design choice here would be around the use of shared pointers for child nodes, this ensures automatic memory management and prevents memory leaks, it also allows multiple nodes to share ownership of the child nodes which is essential for the tree structure of the MCTS. The class also uses move semantics in its constructor for adding a portfolio and one of the important functions implemented here is the *computeUCB1()* function which is used in the Selection stage of the MCTS.



```
template<typename state>
Real Node<state>::computeUCB1(const Real& parentVisits) const {

    //If the node has never been visited, return infinity to encourage exploration
    if(visits == 0) {
        //we are not using this since we are not using the UCB1 formula for the root node.
        return std::numeric_limits<Real>::infinity();
    }

    //UCB1 formula: exploitation + exploration
    Real exploitation = totalReward / visits; //Average reward
    Real exploration = Constants::explorationConstant * std::sqrt(parentVisits / visits);

    return exploitation + exploration;
}
```

Figure 3: UCB1 Function computation

$$\text{UCB1} = \underbrace{\frac{\text{totalReward}}{\text{visits}}}_{\text{Exploitation}} + \underbrace{C \cdot \sqrt{\frac{\text{parentVisits}}{\text{visits}}}}_{\text{Exploration}}$$

Where:

- totalReward is the cumulative reward obtained from simulations passing through the node.
- visits is the number of times the node has been visited.
- parentVisits is the number of times the parent node has been visited.
- C is the exploration constant, which controls the balance between exploration and exploitation.

This function identifies most promising nodes for exploration. The visits and reward are updated during the simulation and backpropagation phases to reflect the outcomes of the simulation. Is it essential to mention here also the explorationConstant which comes from again the **Constants.hpp** file, mainly responsible for the trade-off between exploration and exploitation in the MCTS.

5.2. Pre-Algorithm setup

In this section, we discuss the initial steps in our project that prepares the environment to begin with the MCTS algorithm.

Initially, our data are generated from the python interface, which are then read by the *readPortfolioData* function from **Utils.hpp**. At this point, we would have an initial portfolio with all the needed data points. From here, we decide the weights initialization discussed in the previous sections, based on the user input. Following that, we create an initial Node (root node), by moving the created portfolio to it. From here, we also move again the Node object to the MCTS object and assign it's parameters, which are user-inserted via terminal. After that, the terminal prints the problem setup and the MCTS algorithm begins (invoked via the *MCTS_Setup()* function from the **Utils.hpp**). The I/O here is handled by the **Logger** Class. The algorithm implementation with all its small details will be discussed in the next section.

5.3. MCTS Class

The `MCTS` class implements the Monte Carlo Tree Search (MCTS) algorithm, which we are using for optimizing portfolios, by exploring different weights configurations. The class encapsulates the core components of MCTS, including the search tree, simulation parameters, and optimization strategies. It also provides the main functions for tree traversal, simulation, and backpropagation, as well as advanced features such as fine-tuning and early stopping.

The `MCTS` class includes the following attributes:

- `numberOfSimulations`: The total number of simulations to run during the first Phase of the MCTS.
- `treeWidth`: The minimum number of child nodes to generate during the expansion phase, (Number of sectors).
- `HorizontalScaling`: A parameter controlling the expansion of the treewidth based on the number of adjustment values we would like to test per sector (per child node) in the search tree.
- `HorizontalExpansion`: The number of child nodes to generate during the expansion phase.
- `finetuning`: A flag indicating whether fine-tuning is enabled.
- `FTiterations`: The number of iterations to perform during fine-tuning.
- `root`: The root node of the search tree, representing the initial portfolio node.
- `EarlyStopping`: A flag indicating whether early stopping is enabled.
- `early_stopping_return`: The target return value for early stopping.
- `early_stopping_risk`: The target risk value for early stopping.
- `TreeCut`: A parameter controlling the cut of the search tree.
- `TreeCutReductionValue`: The reduction factor applied in the obtained cuts.

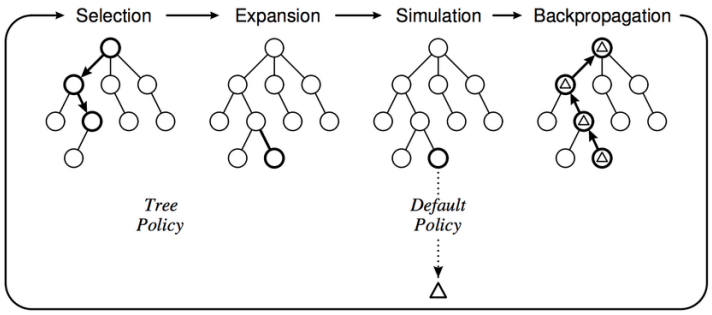


Figure 4: Original NON-optimized MCTS

The MCTS algorithm implemented, is divided into two phases. Phase I is basically where we use the actions 1 and 2 from the **Action Space** revealed in the previous sections, so technically weights adjustments based on returns and risk. Phase II which we call fine-tuning is where we use apply the third action for weights adjustments, which is adjustments based on correlations of returns between different stocks.

5.3.1. Phase I

The first phase of the MCTS is governed by multiple inputs, first of all the numberofsimulations. The number of simulations defines how many times are we going to iterate over the MCTS algorithm sequence <**SELECT-EXPAND-SIMULATE-BACKPROPAGATE**>. In the first step of the algorithm, Selection; if the root node is a leaf (terminal node), we select it and we expand from there. On the other hand, if the root node is not terminal, then the selection occurs by finding the node that maximizes the UCB1, which was explained earlier in the previous section, i.e. the node with the highest UCB1 is then selected and we expand from there. Now moving to the expansion, we have an important parameter is basically the horizontalexpansion, which is computed by multiplying the treewidth and the horizontalscaling. This parameter represents how many child nodes are we going to generate in the expansion step of the algorithm. The treewidth is a pre-defined variable from the **Constants.hpp** file, which equals the number of sectors in the portfolio. Whereas the horizontal scaling is inserted by the user, which is basically the number of weight adjustment values that will be used in a specific action. The catch here is that in our algorithm we are basically considering sector based actions on each expanded nodes, so for each sector action, we consider a child node, and when these actions use an adjustment value to adjust the weights, using the horizontalscaling we define how many adjustment values we want to generate. The function responsible for the generation of these values is *generate_adjustment_values(const int& N)*, which is part of the **Utils.hpp**. This function generates a number of adjustment values N (=horizontalscaling). The generated numbers are in ascending order, and their boundaries are defined in the **Constraints.hpp**, min_adjustment and max_adjustment variables. These adjustment values are used in the action functions in the simulation step of the MCTS, where we run Action1 (adjustment based on return) and Action2 (adjustment based on risk) sequentially on each child node. Finally, the reward for each child node is computed using the *simulePerformance()* function; which performs the following formula :

$$\text{Performance Score} = \underbrace{\text{Sharpe Ratio}}_{\text{Risk-Adjusted Return}} + \underbrace{(\text{Expected Return} - \lambda \cdot \text{Risk})}_{\text{Risk-Return Tradeoff}}$$

Where:

- Sharpe Ratio is the risk-adjusted return of the portfolio, calculated as:

$$\text{Sharpe Ratio} = \frac{\text{Expected Return} - \text{Risk-Free Rate}}{\text{Risk}}$$

- Expected Return is the average return of the portfolio.

- Risk is the volatility (standard deviation) of the portfolio's returns.
- λ is the risk aversion coefficient, which controls the tradeoff between risk and return.

The higher the result, the more important is the node; the higher the reward. Finally, after completion of the simulation step, the backpropagation step begins with updates the visits and rewards for the children and root node.

5.3.2. Phase I Innovative Optimizations

Several optimizations were done to make the MCTS algorithm as efficient as possible for this portfolio optimization task. The first optimization is "Merging the best nodes per sector" in the selection phase. In other words, since we are performing actions per sector; we choose the best child node that maximizes the UCB1 per sector (the adjustment that won performance wise over N adjustment values; (N sector based actions performed on N nodes)), and merge them into one portfolio and select it as the new root. This is done by taking the weights from the best nodes of each sector and combining them into a new weights vector. The function responsible for this task is the *getMergedPortfolioWeights(vector<vector<Real»& weights)* implemented in the **Utils.hpp**.

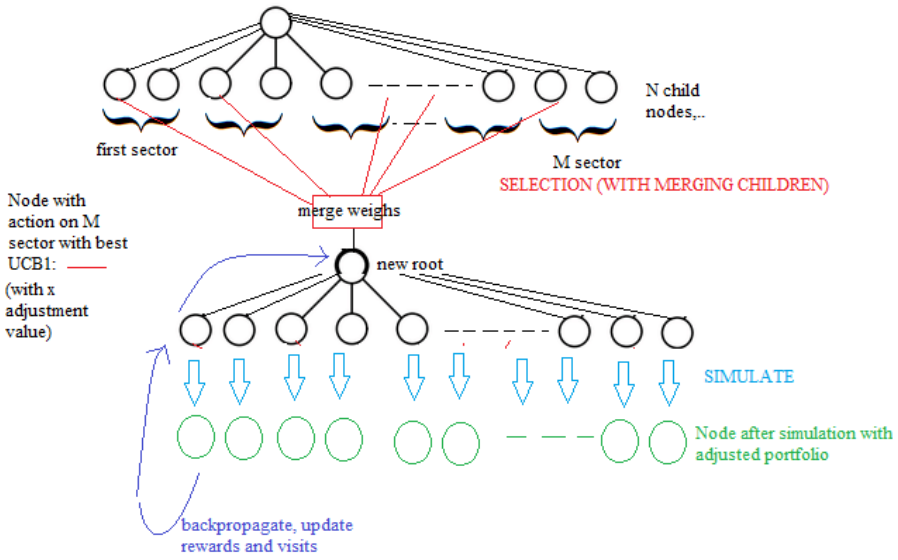


Figure 5: Optimized MCTS with Merging nodes illustration

The second optimization done to the algorithm is the tree-cutting, it is implemented in the simulation part. The treecut and TreeCutReductionValue are the two parameters responsible for enabling this optimization. What this does is it multiplies the adjustment value used in the actions by the reduction_factor computed below. You can imagine it as if we are cutting the dividing the tree simulations by some cut value (artificially), and we obtain multiple sequential blocks of simulations based on the cut value. In these blocks of simulations, the deeper we go , the lower the factor becomes and the lower the adjustments become. The formula below simplifies the explanation :

$$\text{Reduction Factor} = \text{TreeCutReductionValue}^{\left(\frac{\frac{\text{total\_simulations}-\text{remaining\_simulations}}{\text{total\_simulations}}}{\text{TreeCut}}\right)}$$

Where:

- TreeCutReductionValue is the base value used for reduction.
- total_simulations is the total number of simulations planned.
- remaining_simulations is the number of simulations remaining.
- TreeCut is a parameter controlling the rate of reduction.

The reduction factor obtained is multiplied by the adjustment value used in the actions for adjusting weights. As we go deeper in the tree, depending on the tree cut; this reduction value will reduce the adjustment value which leads to the smaller weight adjustments. These deep tree small weight adjustments can be highly beneficial to the portfolio to maximize the weight adjustments as much as possible, leaving no more room for optimization.

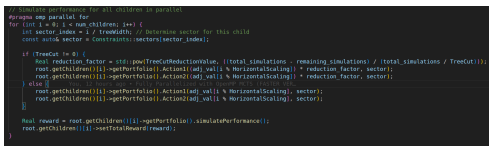


Figure 6: Tree-cut implementation example

It is important to note that the optimizations implemented in this work, such as the tree-cut and node merging are not found in any existing MCTS literature. These enhancements were developed independently as part of this project to address specific challenges in portfolio optimization and to improve the efficiency and effectiveness of the MCTS algorithm.

5.3.3. Phase II

After completing the first phase, it is optional to start a second phase of the MCTS, which is called finetuning. In this phase the main parameters are the finetuning flag (1 if you want to fine tune). Then we have the FTiterations, which is the number of finetuning iterations we would like to do. Also similarly to the first phase, here we iterate over the MCTS algorithm sequence of <**SELECT_FINETUNING-EXPAND_FINETUNING-SIMULATE_FINETUNING-BACKPROPAGATE**>. The only difference between this phase and previous one, is that no combining of child nodes optimization occurs here in the node selection. Moreover, in the expansion, we only expand a number of child nodes equal to the horizontalscaling. And finally, in the simulation step, we apply the third action (correlations-based adjustment) from the action space instead of the other two.

5.3.4. EarlyStopping

EarlyStopping is an optimization done to the MCTS, which covers both phases. If the EarlyStopping flag is set to 1, the user can set specific return and risk values of his own

preferences. What will happen is, whenever the algorithm reaches values equal or bigger to these return and risk user-defined parameters, it will stop iterating and exit. You can think of it as reaching a convergence set by a user, or reaching a persons optimal portfolio goals.

5.4. Parallelization

The Monte Carlo Tree Search (MCTS) algorithm is computationally intensive, especially when applied to large-scale problems like portfolio optimization. To improve performance, the MCTS implementation in this work leverages parallelization using OpenMP, a widely-used API for parallel programming in C++. Parallelization is applied to key phases of the MCTS algorithm, including selection, expansion, simulation, and backpropagation, to distribute the workload across multiple threads and reduce execution time.

5.4.1. Parallel Selection

The selection phase identifies the most promising child nodes to explore. In the parallel implementation, the computation of the Upper Confidence Bound (UCB1) values for each child node is distributed across multiple threads. Each thread calculates the UCB1 values for a subset of child nodes and identifies the best candidate within its subset. The results are then combined using a critical section to ensure thread-safe updates to the global best UCB1 values.

5.4.2. Parallel Expansion

The expansion phase generates new child nodes for the selected node. Parallelization is achieved by distributing the creation of child nodes across multiple threads. Each thread creates a subset of child nodes, and the results are combined using a critical section to ensure thread-safe updates to the list of children.

5.4.3. Parallel Simulation

The simulation phase evaluates the performance of each child node. Parallelization is applied by distributing the simulation of child nodes across multiple threads. Each thread simulates the performance of a subset of child nodes and updates their rewards independently.

5.4.4. Parallel Backpropagation

The backpropagation phase updates the tree with the results of the simulations. Parallelization is applied by distributing the updates of child nodes across multiple threads. A reduction operation is used to safely accumulate the total reward, and a critical section ensures thread-safe updates to the number of visits for each child node.

5.4.5. Benefits of Parallelization

The parallelization of the MCTS algorithm provides several benefits:

- **Improved Performance**: By distributing the workload across multiple threads, the algorithm can process more simulations in less time.
- **Scalability**: The parallel implementation scales well with the number of available CPU cores, making it suitable for large-scale problems.
- **Efficient Resource Utilization**: Parallelization ensures that computational resources are used efficiently, reducing idle time and maximizing throughput.

5.4.6. Challenges and Considerations

While parallelization improves performance, it also introduces challenges:

- **Thread Safety**: Care must be taken to ensure that shared data structures are updated safely using critical sections or atomic operations.
- **Load Balancing**: The workload should be evenly distributed across threads to avoid bottlenecks.
- **Overhead**: The overhead of managing threads and synchronizing data can reduce the benefits of parallelization, especially for small problem sizes.

5.5. Results

5.5.1. Efficient Frontier Analysis of MCTS-Generated Portfolios

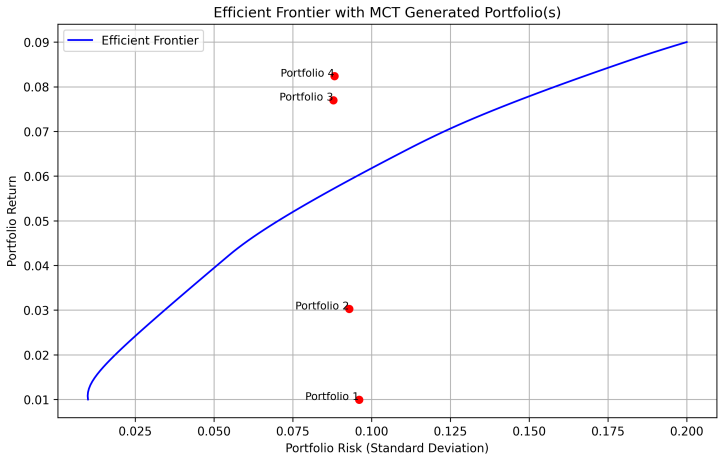


Figure 7: Efficient Frontier with MCTS-obtained portfolios

The portfolios generated by the Monte Carlo Tree Search (MCTS) algorithm were evaluated using the **Efficient Frontier Analysis**, a cornerstone of Modern Portfolio Theory (MPT). The efficient frontier represents the set of optimal portfolios that offer the highest expected return for a given level of risk or the lowest risk for a given level of return. Upon analysis, it was observed that two of the the MCTS-generated portfolios lie **inside the efficient frontier**. This indicates that while these portfolios are viable, they are **sub-optimal** in terms of the risk-return trade-off. Specifically, it is possible to achieve either a higher return for the same level of risk or a lower risk for the same level of return by reallocating assets. While the other two portfolios are outside the efficient frontier and above it, which means they are delivering higher returns for the same level of risk (or lower risk for the same level of return) compared to the theoretical optimum. This is highly unusual and suggests outperformance.Below, we can find a table representing the plotted portfolios. It is important to mention that portfolio's number 3 and 4 were ran with a TreeCut value of 5 and an treecutreductionvalue of 0.95.

Table 4: Summary of MCTS-Generated Portfolios

Portfolio #	**Return (%)**	**Risk (%)**	**Simulations**	**Scaling**	**Comments**
1	0.99	9.60	10	7	Low return, high risk
2	3.03	9.28	100	7	Moderate return, high risk
3	7.69	8.78	1000	7	High return, moderate risk
4	8.23	8.17	10000	7	High return, low risk

5.5.2. Execution time

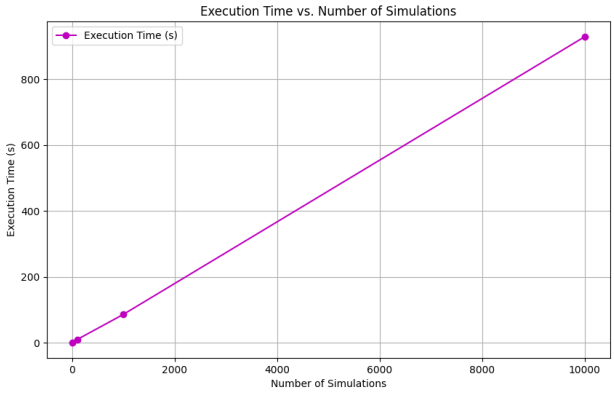


Figure 8: Execution Time vs Number of Simulations

The execution time vs. number of simulations plot exhibits a **linear slope**, indicating that the algorithm scales predictably with problem size. This suggests a **linear time complexity** ($O(n)$), which is expected for this type of computation. The consistent slope demonstrates that the algorithm handles larger workloads efficiently without significant overhead. For example, doubling the number of simulations results in approximately double the execution time, as shown in the plot. This behavior is ideal for HPC applications, as it allows for predictable performance scaling.

5.5.3. Scalability

Strong Scaling

5.5.4. Execution time

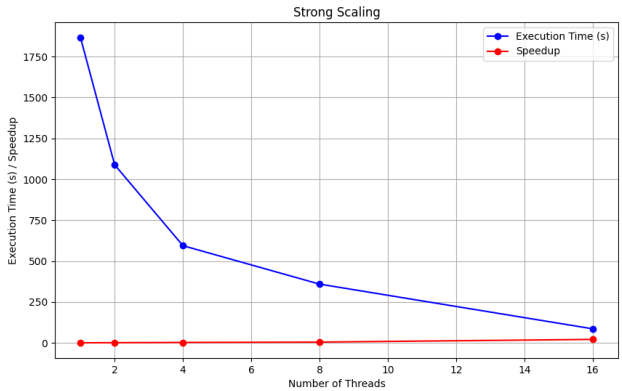


Figure 9: Execution Time vs Number of Threads [simulations =1000, Horizontalscaling =7]

Strong Scaling Analysis

In the strong scaling analysis, we observe that the execution time line converges with the number of threads after a certain point (e.g., **8 threads**). This indicates that the application has reached a **scalability limit** for the given problem size. Up to 8 threads, the execution time decreases significantly, demonstrating efficient parallel performance. However, beyond 8 threads, the execution time no longer decreases significantly, suggesting that additional threads do not contribute to improved performance. This behavior can be attributed to factors such as **communication overhead**, **load imbalance**, or **memory bandwidth limitations**. These results highlight the importance of optimizing parallel efficiency and balancing workloads to achieve better scalability for larger thread counts.

Weak Scaling

5.5.5. Execution time

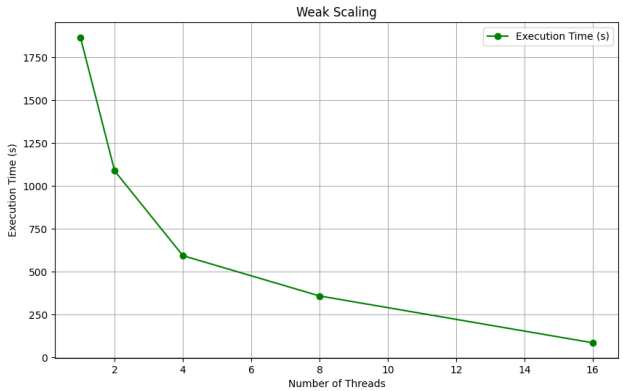


Figure 10: Execution Time vs Number of Threads [simulations =1000, Horizontalscaling =7]

In the weak scaling analysis, we observe that the execution time decreases significantly as the number of threads increases from **1 to 16**. This indicates that the application is **highly scalable** and efficiently handles larger problem sizes with additional threads. Specifically, the execution time decreases by approximately **[41]%** when moving from 1 to 2 threads, **[54]%** when moving from 2 to 4 threads, and **[59]%** when moving from 4 to 8 threads. This behavior demonstrates that the application benefits from parallelism and effectively distributes the workload across threads. The consistent reduction in execution time up to 16 threads suggests that the application has **low communication overhead** and **good load balancing**, making it well-suited for larger-scale computations.

6. Conclusion

In this work, we presented a comprehensive implementation of the Monte Carlo Tree Search (MCTS) algorithm for portfolio optimization, leveraging efficient programming and integrating optimizations such as node portfolio merging, treecut, finetuning and early stopping to enhance performance and efficiency. The integration of parallelization using OpenMP further improved the scalability of the algorithm, enabling it to handle large-scale problems effectively. Moreover, this demonstrates the potential of MCTS for portfolio optimization and provides a solid foundation for future research and applications in the field of computational finance. The innovative optimizations and parallelization techniques introduced here pave the way for more efficient and scalable solutions to complex optimization problems. Throughout the project, numerous challenges were encountered, and valuable lessons were learned across various aspects, including programming techniques, optimization strategies, and overall project management.

7. References

kocsis2006bandit Kocsis, L., & Szepesvári, C. (2006). Bandit based Monte-Carlo planning. In *European conference on machine learning* (pp. 282–293). Springer.

browne2012survey Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., ... & Colton, S. (2012). A survey of Monte Carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1), 1–43.