Tomas Conti's Operative Systems project.
Project code: **gitHub**

Chapter 0

The goal of this paper is to develop and validate a parallel algorithm for efficiently calculating the beta coefficient of financial portfolios.
We explore how High-Performance Computing (HPC) can enhance data mining operations by taking advantage of the MIMD (Multiple Instruction, Multiple Data) architecture.

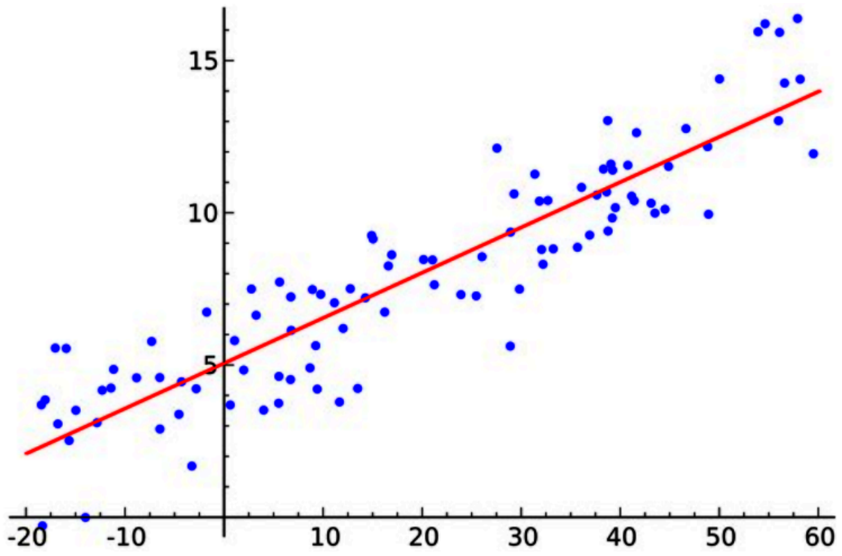
Leonardo, the CINECA supercomputer, will serve as the HPC architecture for testing the project.

Linear Regression

Linear regression is a fundamental statistical method widely employed in finance to model the relationship between a dependent variable and one or more independent variables. It is particularly useful in analyzing the connection between asset returns and market returns, providing insights into risk and performance assessment.

$$Y = \alpha + \beta X + \varepsilon$$

- Y represents the dependent variable (return of a single asset or portafolio).
- X denotes the independent variable (return of a market index).
- α (the intercept) reflects the asset’s expected return independent of market movements.
- ε is the error term for capturing idiosyncratic variations.
- β (the coefficient) measures the asset’s sensitivity to changes in the market return.



Financial portfolios represent an amout of single asset investment, with different weights the "portfolio creator" decides the kind of investment strategy.
In financial applications, linear regression is instrumental in estimating an asset's exposure to systematic risk. One of its key applications is in determining the beta coefficient, a measure of an asset's volatility relative to the overall market.
The beta coefficient plays a crucial role in the Capital Asset Pricing Model (CAPM), where it serves as a key determinant of expected returns. A portfolio's overall beta, given a set of constituent assets with respective weights and betas is computed as:

$$\beta = \sum_{i=0}^{N-1} \beta_i \cdot \omega_i$$

Now change the focus on single Beta coefficient elaboration, computed as:

$$\beta_i = \frac{Cov(R_p, R_m)}{Var(R_m)}$$

- β=1 The asset moves in sync with the market, if the market increases by 1% the asset will do the same.
- β>1 The asset is more volatile than the market, if the market increases by 1% the asset will do the same but more aggressively.

- $0<\beta<1$ The asset moves in the same direction as the market but with a lower magnitude, if the market increases by 1% the asset moves by less.
- $\beta=0$ There is an uncorrelation between the market and the asset.
- $\beta<0$ The asset moves in the opposite direction of the market.

We have to consider the market index and single title yields, these indicate the trend of the market. The yields calculation came from historical returns. The single Beta operation are embarrassingly parallelizable:

$$Cov(X,Y) = \frac{1}{N}\sum_{i=0}^{N-1}(X_i - \overline{X})(Y_i - \overline{Y})$$

$$Var(X) = \frac{1}{N}\sum_{i=0}^{N-1}(X_i - \overline{X})^2$$

The Beta explains the volatility of a specific title relative to the market index yields. To validate the Beta operation, we must consider the correlation between the two yields:

$$-1 \leq \rho(R_p, R_m) = \frac{Cov(R_p, R_m)}{\delta_p \cdot \delta_m} \leq 1$$

- $\rho=1$. High correlation between index and title yields.
- $\rho=-1$. High inverse correlation between index and title yields.
- $\rho=0$. Weak correlation between index and title yields.

The paper and project scope is about how using a HPC (high parallel computing) infrastructure can increase the speed up of Beta calculation. The idea start from a sequential version of algoritm, by the data incrasing the execution time became bigger and bigger. The algorithm tries to handle that by MIMD operation by MPI library and the architecture.

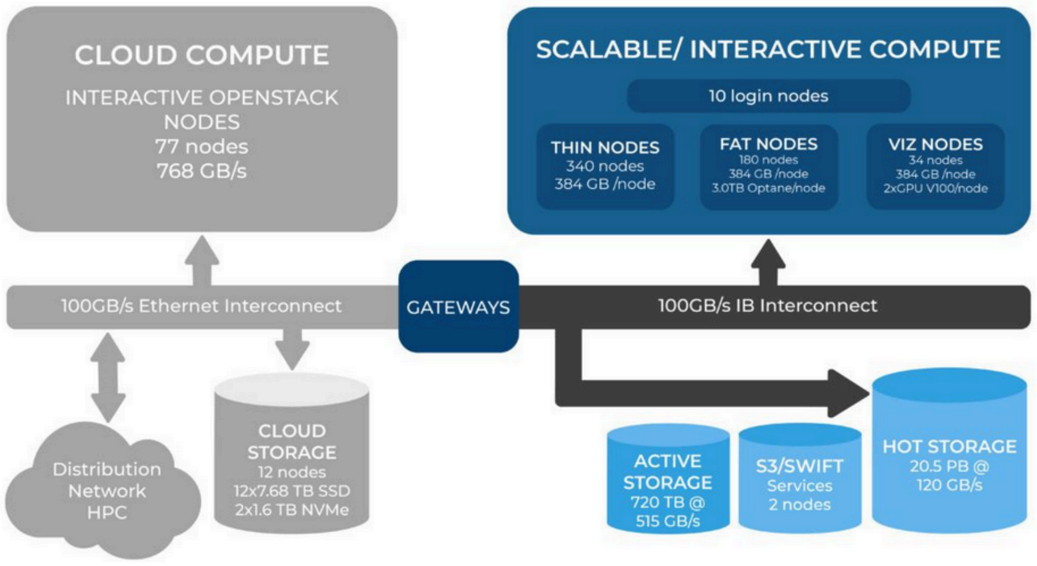
This algorithm can help to create different and more efficient Beta portfolio generation for financial data trading and analysis.

Chapter 1

Galileo100 will serve as the HPC architecture for project code execution. The cloud compute and interactive execution environments are separated by a gateway component.
We access the infrastructure through one of the 10 login nodes and execute our workloads within the cloud interactive compute environment.
There are different kind of node availables:

- 340 THIN NODES
- 180 FAT NODES
- 34 GPU NODES

The THIN NODES are ideal for the project scope.

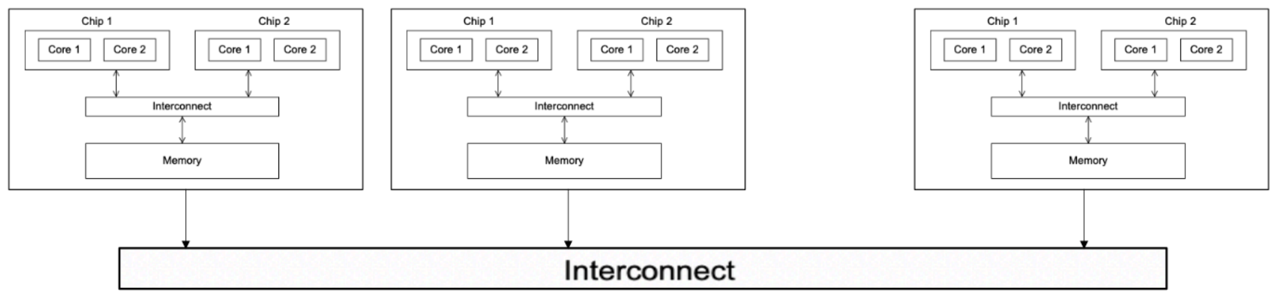


To deploy the execution operation on architecture we have to interact with a SLURMD, a workerload daemon scheduler.
The interaction with the scheduler involves a script bash, by this file we can describe the execution architecture like nodes, tasks, partition of the execution and modules.

```
#!/bin/bash
#SBATCH --account=tra24_IngInfB2
#SBATCH --partition=g100_usr_prod #partition of execution
#SBATCH --nodes=1
```

```
#SBATCH --ntasks-per-node=1
#SBATCH -o TEST_Strong.out
#SBATCH -e TEST_Strong.err
module load autoload intelmpi
srun ./TEST
```

The achitecture configuration includes an interconnection layer for each node. Inside the single node there is a shared memory configuration with 2 CPUs composed of 24 cores each (48 process for each node).



By leveraging a specialized library like MPI, we can design an algorithm optimized for HPC architecture and high degrees of parallelization. Distributing the algorithm's execution across different execution units can significantly enhance its speed.

Chapter 2

This chapter focuses on the study of code and data configuration. At the end, a summary of the collected statistical results is provided.

Implementation

The input data consists of a table containing a title and the daily price of a market index. Using a broadcast or scatter operation, we distribute the processed data to each core for computation.

	day0	day1	day2	day3	day4
title0	100	110	130	125	140
title1	200	150	100	100	90
index0	100	105	115	113	120
index1	50	90	100	120	150

Starting from this table, the objective is to extract key data parameters such as beta and correlations. The first step involves computing the returns for both the title and the market index. Given the number of rows, the data processing is distributed across multiple cores (two cores in this example) to enhance computational efficiency.

```
//work distribution
int block_size = (int)ceil((double)(TITLES + INDEX) / size);
int start_idx = rank * block_size;
int end_idx = (rank + 1) * block_size;
if (end_idx > TITLES + INDEX) end_idx = TITLES + INDEX;
int local_size = end_idx - start_idx;

void calculateReturns(float* prices, float* returns, int local_size) {
   for (int row = 0; row < local_size; row++) {
       for (int day = 0; day < (DAYS - 1); day++) {
           returns[row * (DAYS - 1) + day] = (prices[row * DAYS + day + 1] -
                    prices[row * DAYS + day]) / prices[row * DAYS + day];
       }
   }
}
//function call
calculateReturns(data.prices + start_idx * DAYS, localReturns, local_size);
```

At this stage, each core processes a subset of row return data. The computation proceeds with the calculation of the Average and Variance.
The table is now transformed into a return data table, where each row contains DAYS−1 data points.

	day0	day1	day2	day3	Average	Variance
title0	0.10	0.1818	-0.0384	0.12	0.1079	0.0064
title1	-0.25	-0.333	0.000	-0.10	-0.1707	0.0167
index0	0.05	0.095	-0.017	0.061	0.0474	0.0016
index1	0.80	0.111	0.20	0.25	0.3402	0.0729

The code for managing those operations required the return data of the previous step, each core already owns the data.

```
void calculateAverages(float* returns, float* averages, int local_size) {
   for (int row = 0; row < local_size; row++) {
       for (int day = 0; day < (DAYS - 1); day++) {
           averages[row] += returns[row * (DAYS - 1) + day];
       }
       averages[row] /= (DAYS - 1);
   }
}
void calculateVariances(float* returns, float* averages, float* variances, int local_size, int
rank) {
   for (int row = 0; row < local_size; row++) {
       variances[row] = 0;
       for (int day = 0; day < (DAYS - 1); day++) {
           float diff = returns[row * (DAYS - 1) + day] - averages[row];
           variances[row] += diff * diff;
       }
       variances[row] /= (float)(DAYS - 1);
   }
}
```

At this stage, each core holds its computed yields and variance. To proceed, we need to distribute this data across the entire system. Using the MPI function MPI_Allgatherv, we can achieve this distribution, though these operations represent a bottleneck, with latency increasing as the number of cores and data size grow. Once completed, each core will have all the necessary data to perform the final two steps of the algorithm.

```
MPI_Allgatherv(localReturns, local_size * (DAYS - 1), MPI_FLOAT,data.returns,
sendcounts_returns, displs_returns, MPI_FLOAT,MPI_COMM_WORLD);
   MPI_Allgatherv(localAverage, local_size, MPI_FLOAT,data.averages, sendcounts_other,
displs_other, MPI_FLOAT,MPI_COMM_WORLD);
   MPI_Allgatherv(localVariance, local_size, MPI_FLOAT,data.variances, sendcounts_other,
displs_other, MPI_FLOAT,MPI_COMM_WORLD);
```

The possible covariance results amount to INDEX×TITLES, with the computational workload distributed based on the length of the market index row. For the Beta and Correlation calculations, the workload distribution considers the entire TITLES×INDEX matrix.

```
void calculateCovariances(float* returns, float* averages, float* covariances, int size, int rank, int* recvcounts,
int* displs, float* local_cov) {
   int index_block_size = INDEX / size;
   int index_remainder = INDEX % size;
   int index_start_idx = rank * index_block_size + (rank < index_remainder ? rank : index_remainder);
   int index_end_idx = index_start_idx + index_block_size + (rank < index_remainder ? 1 : 0);
   int local_index_size = index_end_idx - index_start_idx;
   for (int i = 0; i < local_index_size; i++) {
       for (int t = 0; t < TITLES; t++) {
           float sum = 0.0f;
           for (int d = 0; d < DAYS - 1; d++) {
               sum += (returns[t * (DAYS - 1) + d] - averages[t]) * (returns[(TITLES + index_start_idx + i) * (DAYS
- 1) + d] - averages[TITLES + index_start_idx + i]);
           }
           local_cov[i * TITLES + t] = sum / (DAYS - 1);
           }
   }
   MPI_Allgatherv(local_cov, local_index_size * TITLES, MPI_FLOAT, covariances, recvcounts, displs, MPI_FLOAT,
MPI_COMM_WORLD);
}

void calculateBetasAndCorrelations(float* covariances, float* variances, float** betas, float** correlations, int
size, int rank) {
   int total_combinations = TITLES * INDEX;
   int block_size = total_combinations / size;
   int remainder = total_combinations % size;
   int start_idx = rank * block_size + (rank < remainder ? rank : remainder);
   int end_idx = start_idx + block_size + (rank < remainder ? 1 : 0);
   const float epsilon = 1e-7;
   for (int i = start_idx; i < end_idx; i++) {
       int title = i / INDEX;
        int idx = i % INDEX;
       if (fabsf(variances[TITLES + idx]) > epsilon) {
           betas[title][idx] = covariances[title + idx * TITLES] / variances[TITLES + idx];
       } else {betas[title][idx] = 0.0f; }

       if (fabsf(variances[TITLES + idx]) > epsilon && fabsf(variances[title]) > epsilon) {
       correlations[title][idx]=covariances[title+idx*TITLES]/(sqrtf(variances[TITLES+idx])*sqrtf
variances[title]));
       } else {
           correlations[title][idx] = 0.0f;} } }
```

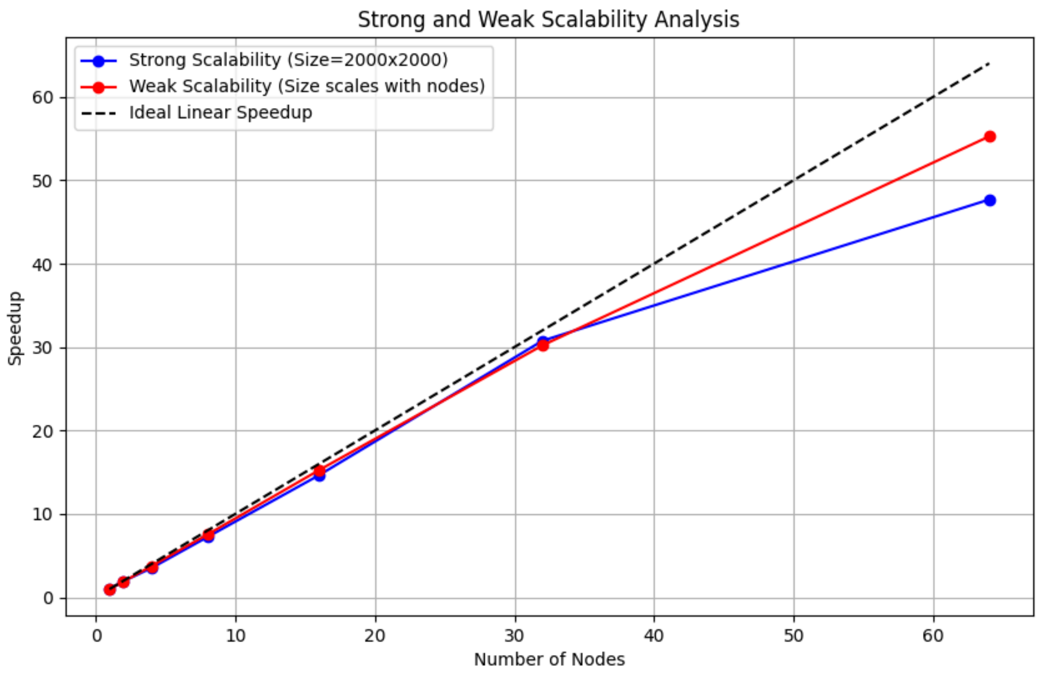
Final elaboration values:

(title, index)	Covarince	Betas	Correlations
(0, 0)	0.003294	1.96544	0.99
(0, 1)	-0.00028	-0.00388	-0.01
(1, 0)	-0.00450	-2.68729	-0.85
(1, 1)	-0.00737	-0.10118	-0.21

Results

To exploit the elaboration we use a small data set, for the real testing this is considerably bigger. With a normal CPU sequenzial elaboration the elaboration time is high and the MIMD elaboration allows to reduce it.

- Strong scalability. Starting from a fixed data size, let's increase the process number. The speedup does not increase proportionally.
- Weak scalability. Starting from a small data size, let’s increase proportionally the data size and the number of processors. The speedup has a constraint to grow up, each processor manages the same data size.



The strong scalability speedup decreases as the number of processors increases. On the other hand, the weak scalability graph demonstrates how the seed size grows more consistently with the increase in the number of processors.

With compiler optimizations such as -O3 in g++, there is a significant reduction in execution time.

```
//no g++ compiler optimizations
mpiicc -std=c99 TEST.c -o TEST
//O3 g++ compiler optimizations
mpiicc -O3 -std=c99 TEST5.c -o TEST5
```

This optimization applies various enhancements during the compilation process. Intrinsec operations are one of these optimizations, boosting SIMD processor operations using AVX CPU registers. The timing execution gap (for this specific execution case with the 2000x2000 configuration) is as follows:

	1 processor	2 processors	64 processors
Not optimized (second)	61.514968	31.636622	1.296374
Optimized (second)	3.794895	2.611563	0.576040

There is a significant time gap, and this optimization contributes to a substantial reduction in execution time.

future implementations

As a future implementation, it would be possible to integrate the OpenMP library to enhance parallel execution on each core.
This could improve the execution speedup by achieving a higher level of parallelism for each iteration and optimizing workload distribution at the core level.