Εθνικό και Καποδιστριακό Πανεπιστήμιο Αθηνών
Τμήμα Πληροφορικής και Τηλεπικοινωνιών

# Parallel Systems

Assignment 3

**Name:** Giorgos Nikolaou **sdi:** 1115202000154

**Name:** Helen Fili **sdi:** 1115202100203

---

# 1 Exercise 3.1

## 1.1 Description of the problem

In this problem, the task is to use the **Monte Carlo** method to approximate $\pi$ by sampling random points in a $2 \times 2$ square (centered at $(0, 0)$) and observing how many of them fall within the unit circle. We approach the problem with both serial and parallel implementations (using `MPI`). As expected, the method produces better approximations as we increase the number of iterations, but that is not efficient when using the serial algorithm. To accelerate the computation, we utilize `MPI` to parallelize the random sampling among the processes.

## 1.2 Brief description of the solution

Description of Algorithms:

- **Serial**: The serial version of the algorithm is executed when the command line argument `<n>` is set to 1.
- **Parallel**: The parallel version of the algorithm is executed when the command line argument `<n>` is greater than 1. In this version, the samples of random points from arrow throwing are divided equally among the processes.

The `main()` function initiates the `MPI` environment by initializing essential variables with a call to `MPI_Init()`. Afterwards, the rank of the current process and the total number of processes are retrieved. Subsequently, the number of arrows is broadcasted and evenly distributed among the processes. Each process then calculates the number of arrows that landed in the critical area (inside the unit circle) and stores the result in its local variable `arrows`. The `MPI_Reduce()` function is then utilized to sum the local variables into a global variable `total_arrows`. Finally, process 0 is responsible for calculating the approximation of $\pi$ and then `MPI_Finalize()` is executed to perform cleanup operations and finalize the `MPI` environment.

**Implementation Details**:

- Type `long long` is macro-defined as `LONG`.
- For the point sampling, a thread-safe version of `rand()` is used (`rand_r()`).

## 1.3 Presentation of experiments and analysis

Below, we provide the results for various numbers of task, nodes and sampling points.

Specifically, we executed the program with every combination of the following parameters:

- `<nodes>`: $[1, 2, 4]$
- `<tasks>`: $[1, 2, 4, 8, 16]$
- `<n>`: $[10^7, 10^8, 10^9, 5 \cdot 10^9, 10^{10}]$

To conduct this grid search and generate the corresponding plots, we developed the scripts `exp.sh` and `plots.py`. To reproduce, execute:

```
$ cd monte_carlo
$ make monte_carlo
$ ./exp.sh
$ python3 ./plots.py
```

Listing 1: Monte Carlo Plotting

To perform manual executions, run:

```
$ cd monte_carlo
$ make monte_carlo
$ make single-node N=<n> or make multi-node N=<n> M=<m>
```

Listing 2: Monte Carlo Execution

Experiments were conducted in the Linux Lab, each repeated 4 times, using the following specifications (with no significant variance between different machines):

- **OS**: `Ubuntu 20.04.6 LTS`.
- **CPU**: `Intel Core i5-6500 CPU @ 3.20GHz`, featuring 4 cores with 1 thread each.
- **Compiler**: `gcc (Ubuntu 9.4.0-1ubuntu1~20.04.2) 9.4.0`

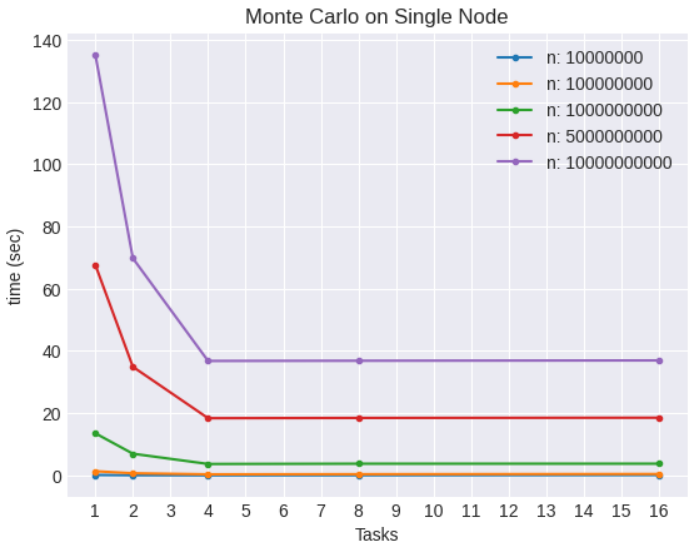



Figure 1: Monte Carlo Single-Node Results

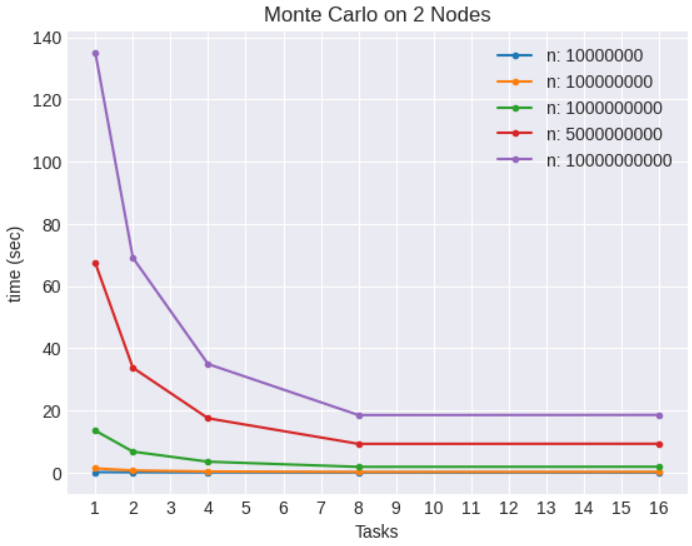



Figure 2: Monte Carlo 2-Node Results

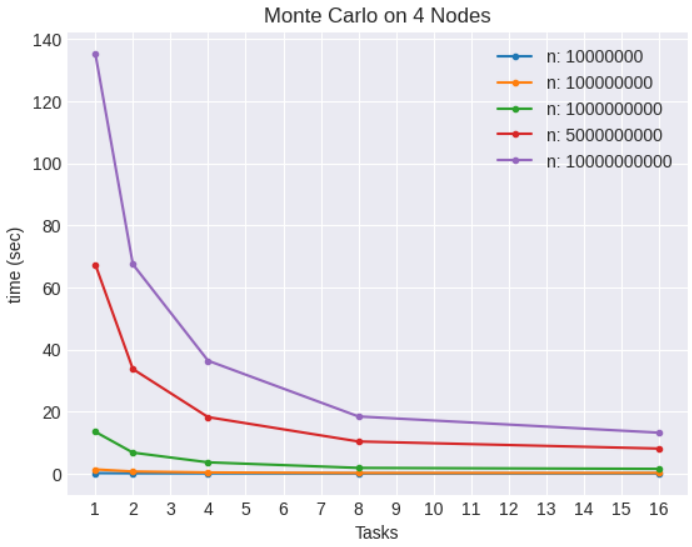



Figure 3: Monte Carlo 4-Node Results

We make the following observations:

- **Doubling** the number of processes results in **reduction** of the execution time by *half*, as expected due to the shared workload across multiple processes.
- In our single-node configuration, which has 4 cores, we observe improvement up to the use of 4 processes. However, increasing the task count beyond 4 does **not** yield further reductions in execution time.
- Similarly, with multiple nodes, we observe improvement up to the total number of cores in the configuration, as anticipated.
- While using more nodes does lead to performance gains, it's important to note that the increase is not strictly linear. This is attributed to the inclusion of communication time between threads in the overall computation.

# 2 Exercise 3.2

## 2.1 Description of the problem

The goal of this exercise is to implement a program for efficient **matrix-vector multiplication**, utilizing a block-wise distribution of the matrix. The aim is to optimize the computation of multiplying a square matrix with a vector by harnessing the parallelization capabilities of `MPI`.

## 2.2 Brief description of the solution

**Main idea**: The matrix is treated as a collection of blocks, where each block represents a sub-matrix of the original matrix and is processed by an individual process. Consequently, the matrix is partitioned into `comm_sz` blocks. The next step is to determine the number of rows and columns in this block-wise representation of the matrix. Given that the value of `comm_sz` is a perfect square, and its square root is a divisor of the matrix size `n`, it follows that the number of rows will be *equal* to the number of columns and that the sub-matrices will also be *square*. After establishing these facts, we can easily compute the dimensions of the sub-matrices using the formula $dim = \frac{n}{\text{columns}}$, resulting in sub-matrices of size $dim \times dim$.

In this implementation, we **flatten** the $2D$ matrix into an $1D$ vector as seen in Figure 4. This enables us to effortlessly scatter the flattened matrix through `MPI_COMM_WORLD`, with each process obtaining its respective block with ease. The total computations are divided among the processes, with each one handling the computations for a specific block.

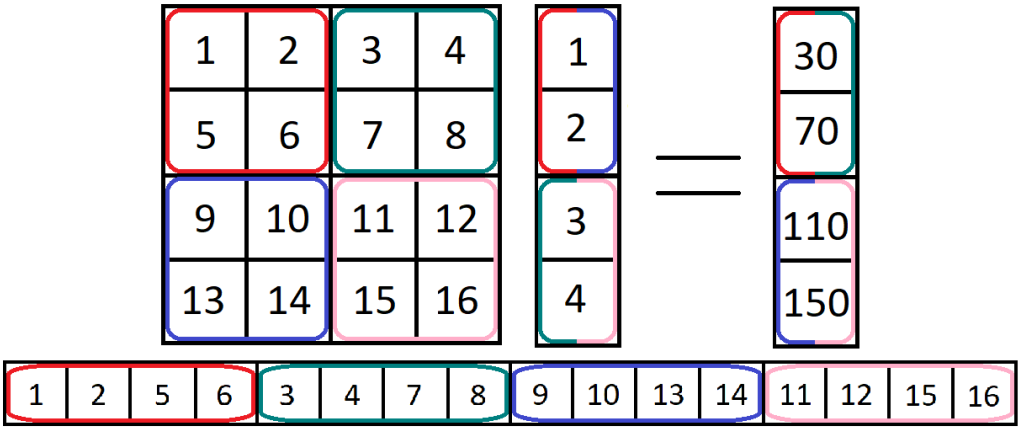



Figure 4: The Flattened Matrix and the Mapping of Vector Elements to Blocks

**Custom Communicators**: Each block requires a segment of the vector for multiplication and another segment of the product vector for its result. Therefore, we need a method to scatter the original vector across multiple processes so that each column of blocks has access to the corresponding part of the vector (as illustrated in Figure 4).

- **Scattering Vector**: Using one call to `MPI_Scatter` is not an option, and instead we must create custom communicators for this purpose. Initially, we scatter the vector among the processes in the first row (of blocks), after which each process broadcasts its specific segment to every block in its corresponding column.
- **Gathering Product**: To correctly retrieve the partial products and place them into the product vector, we first reduce the output of each row (of blocks) into a `Head Product` that exists in the first process of each row. Subsequently, we gather the individual products into the original vector.

**Implementation Details**:

- Only one dimension of the matrix needs to be provided since the matrix is square.
- The program validates the multiplication result, providing assurance that our implementation is accurate.

## 2.3 Presentation of experiments and analysis

Below, we provide the results for various numbers of processes, nodes and matrix dimensions.

Specifically, we executed the serial version of the program and then repeated the process for every combination of the following parameters:

- `<nodes>`: $[1, 4, 16]$
- `<world size>`: $4 \cdot$ `<nodes>`
- `<n>`: $[1000, 5000, 10000, 15000]$

**Note**: The experimentation is limited by the nature of the parallelization, specifically the constraints regarding the `comm_sz` mentioned earlier. We choose a `comm_sz` equal to 4 times the number of nodes to fully utilize the available cores.

To perform the grid search described in the assignment and produce the corresponding plots, we created the scripts `sizes.sh`, `exp.sh`, `plots.py`. To reproduce the results, execute the following commands:

```
$ cd mat_vec_mul
$ make mvm SIZE=<size>
$ ./sizes.sh
$ python3 ./plots.py
```

Listing 3: Matrix-Vector Multiplication Plotting

To perform manual executions, run:

```
$ cd mat_vec_mul
$ make mvm SIZE=<size>
$ make single-node N=<n> or make multi-node N=<n> M=<m>
```

Listing 4: Matrix-Vector Multiplication Execution

Experiments were conducted in the Linux Lab, each repeated 4 times, using the following specifications (with no significant variance between different machines):

- **OS**: `Ubuntu 20.04.6 LTS`.
- **CPU**: `Intel Core i5-6500 CPU @ 3.20GHz`, featuring 4 cores with 1 thread each.
- **Compiler**: `gcc (Ubuntu 9.4.0-1ubuntu1~20.04.2) 9.4.0`

### 2.3.1 Performance

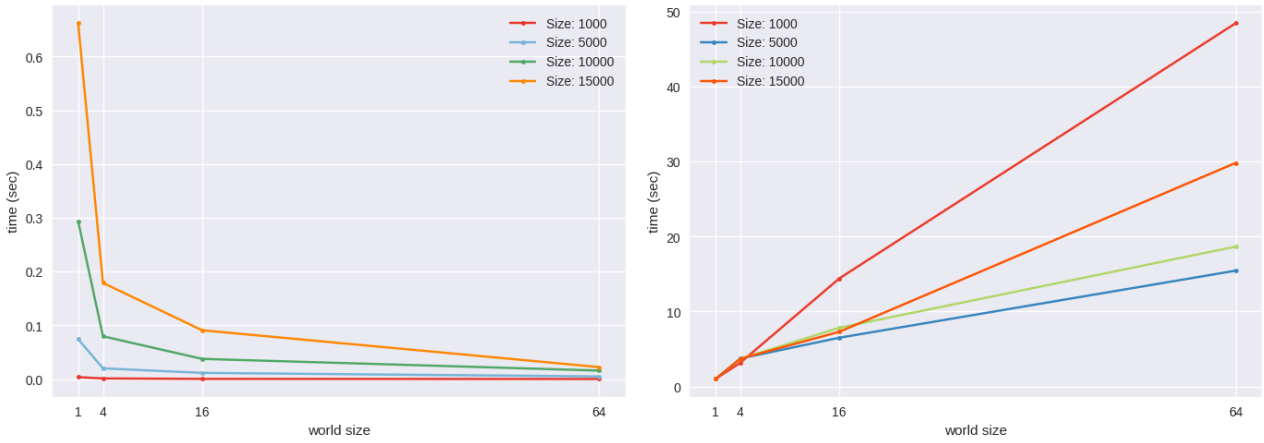
It is important to note that when using a multi-node configuration, transferring data from one node to another is done through expensive connections. Especially when the matrix size becomes large, this has a significant impact on performance. In practice, this overhead becomes negligible since we execute multiple operations on each process, not solely limited to a single matrix multiplication. Nonetheless, we measure both times for a more comprehensive analysis.

### 2.3.2 Performance without Communication Time

We make the following observations:

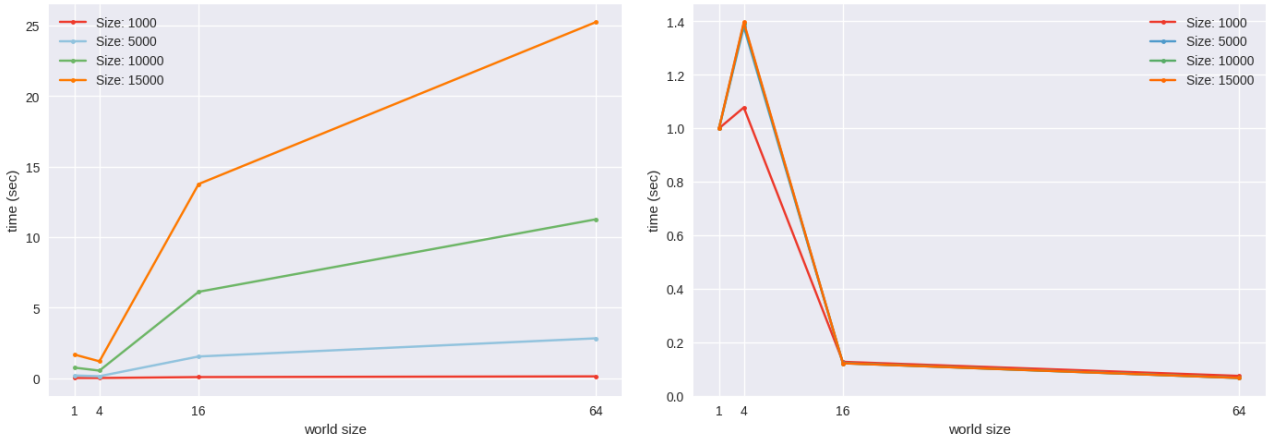
- As expected, execution time decreases as the number of processes increases, since each process has a lesser workload.
- Generally, the speedup falls considerably short of linearity. This is largely attributable to the fact that our configuration is publicly accessible to students, with each core fully engaged due to the substantial number of concurrently executed processes.
- We anticipated an improvement in speedup as we increased the problem dimension. However, the most significant speedup occurs when `n` is equal to 1000. This is likely attributed to the smaller size, which necessitates considerably fewer context-switches and achieves better cache efficiency.
- As the matrix dimension becomes sufficiently large, we observe the expected increase in performance, as the cache efficiency remains consistent in each dimension.

Figure 5: Time Performance and Speedup of Multiplication




### 2.3.3 Performance with Communication Time

Figure 6: Time Performance and Speedup of Multiplication and Comms




We observe that the overall time, including communication between processes significantly increases when scaling the number of nodes. This is anticipated, given the comparatively high cost of communication, while the limited operations executed by each process fall short of compensating for the resultant overhead.

# 3 Exercise 3.3

## 3.1 Description of the problem

The goal of this exercise is to modify our implementation of Exercise 3.2 to use a hybrid approach with `MPI` -for node-level parallelism- and `OpenMP` -for parallelism within the cores on each node-.

## 3.2 Brief description of the solution

**Implementation Details**:

- The `MPI` implementation from Exercise 3.2 is enhanced as `OpenMP` is strategically employed on the outer loop during the multiplication of the sub-matrix with the sub-vector.
- The number of threads `OpenMP` uses is set to 4 to maximize the utilization on our configuration.

**Note**: Throughout our experiments, we identified potential false-sharing issues, as observed in Figure 7 when world size is equal to 16. To address this concern, we implemented a strategy using a private variable to compute the result. Subsequently, we assigned it once to the corresponding position in the product vector.

For compiling and executing the program, please consult subsection 2.3. Keep in mind that the code is located in `mat_vec_mul_hybrid`.

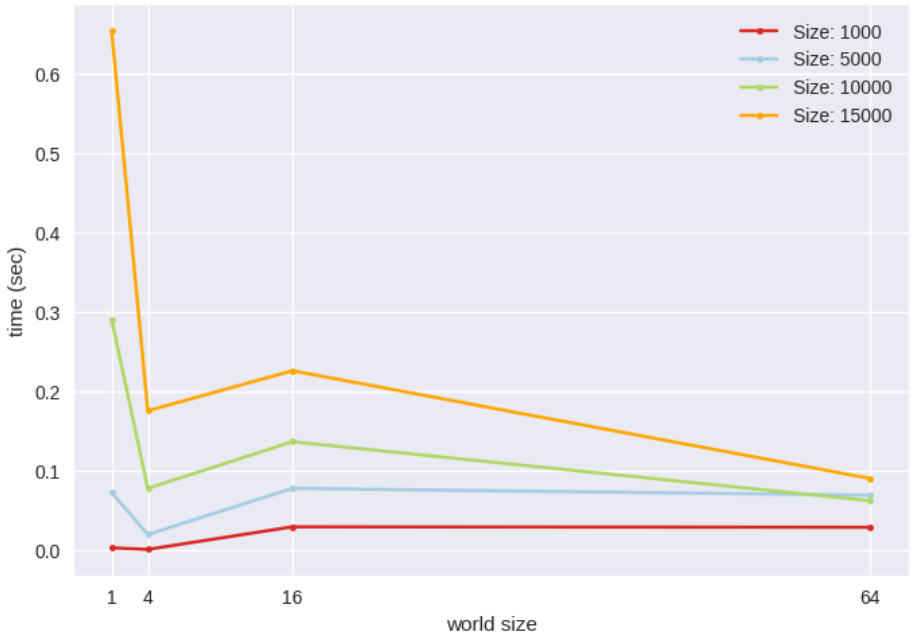



Figure 7: False Sharing using the Hybrid Method

## 3.3 Presentation of experiments and analysis

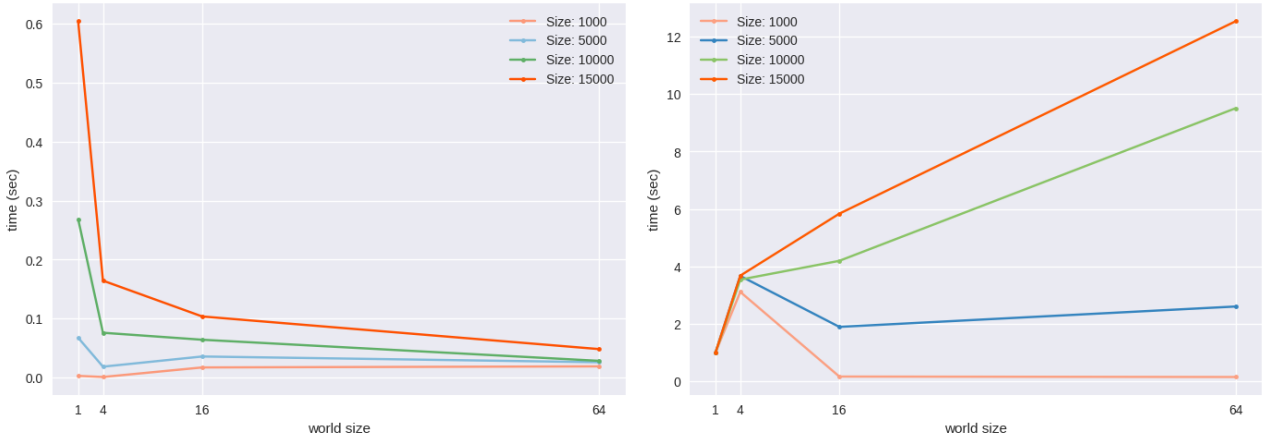
Similar to the previous exercise, the multi-node configuration introduces a notable communication overhead. To conduct a thorough analysis, we measure both scenarios—taking into account both with and without communication time.

### 3.3.1 Performance without Communication Time

We make the following observations:

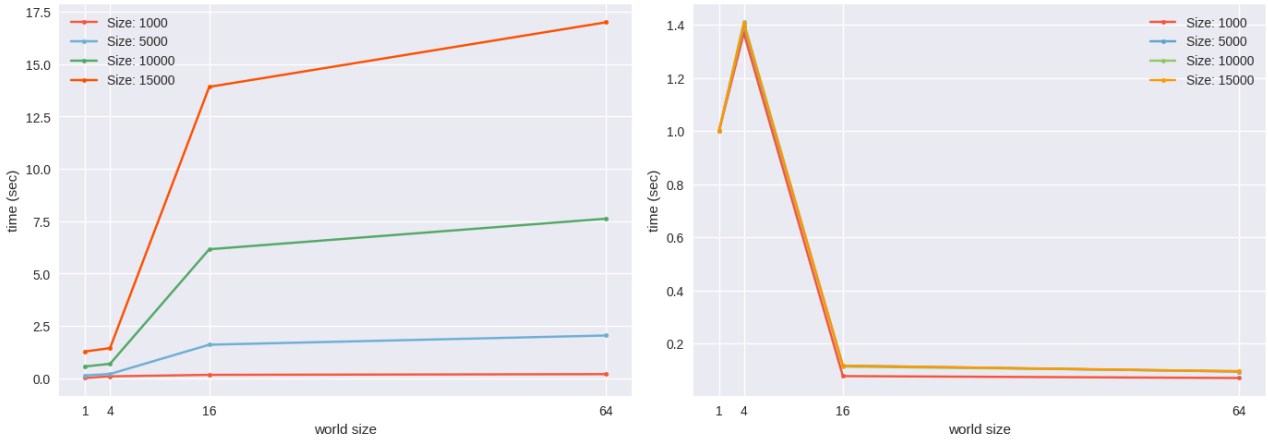
- There is a decrease in performance when employing the hybrid method compared to the `MPI` implementation. This outcome was unexpected, as we initially anticipated an improvement in performance, considering threads are typically considered a lighter-weight version of processes.
- This is likely due to a combination of the following:
  - If the amount of work done in each iteration of the loop is relatively small, the overhead of thread creation may become more noticeable. This is observed for smaller matrix dimensions, and as expected, with the increase in dimensionality, the speedup also improves.
  - When running on a configuration with more nodes, the anticipated increase in performance is not consistently observed. This discrepancy is attributed to heightened contention for system resources, as confirmed by executing the program on different hosts with notably varied results.

Figure 8: Time Performance and Speedup of Multiplication




### 3.3.2 Performance with Communication Time

Figure 9: Time Performance and Speedup of Multiplication




Similarly to Exercise 3.2, we observe that the overall time, including communication between processes significantly increases when scaling the number of nodes. Nevertheless, we note a 30% decrease in overall execution time compared to the previous exercise. This suggests that the hybrid implementation is preferable.