Optimizing Parallel Matrix Multiplication: Performance Analysis and Scalability on the Univaq Supercomputer.

Submitted by;

1. AUDRY LEUNORAH ENOI - 293417
2. EMMANUEL VINCENT - 293447
3. ESTHER AKUABATA NWOBODO - 293433

Date: June 17th, 2024

Institution: University of L'Aquila

Contents

1. Abstract

This study uses Python and MPI (Message Passing Interface) to perform parallel matrix multiplication on the Univaq Supercomputer, examining its scalability and efficiency. The main goal of this study is to compare different computational nodes (1, 2, 4, 6, 8, 10) with a fixed 15 by 15 matrix in order to assess efficiency performance parameters including speedup, efficiency, and overhead. We investigate the effects of parallelization on time and resource utilization, considering analysis. Our results show significant gains in computation speed with an increase in nodes, indicating all-encompassing approaches to parallel computing in matrix operations. The goal of this research is to improve our knowledge of the efficiency of parallel computing on contemporary HPC systems by guiding the development of optimal node utilization and performance scalability.

2. Introduction

2.1 Basis

Parallel computing is a woke topic addressing computational tasks on various Engineering disciplines. This provides an execution on complex algorithms harnessing the collective processing power of multiple computing nodes, with an insinuation in infeasibility with traditional sequential methods. The skill obtained is needed in various fields such as weather forecasting, molecular dynamics simulations, and large-scale data analytics, where rapid processing of vast datasets and iterative computations are essential.

Systems known as high-performance computing (HPC) are the best available for parallel computing, such as the Univaq Supercomputer used in this investigation. These systems use high-speed interconnects, specialized accelerators like GPUs, and sophisticated multi-core CPUs to provide the computing capacity needed to solve compute-intensive issues on a large scale. The Univaq Supercomputer offers an excellent environment for investigating the performance properties of parallel algorithms because to its InfiniBand-connected Intel Xeon processors and NVIDIA Tesla GPUs.

2.2 Objective

This project's goal is to assess parallel matrix multiplication on the Univaq Supercomputer's scalability and efficiency. The classic example of a computationally demanding task that greatly benefits from parallelization is matrix multiplication. We measure the system's ability to scale performance with increased computational resources by dividing the task among several computing nodes.

2.3 Scope

This paper provides a thorough examination of our approach, findings, and analysis from tests carried out on the Univaq Supercomputer. Focusing on key performance parameters such as speedup, efficiency, and overhead, we use a fixed 20x20 matrix to alter the number of computing nodes (1, 2, 4, 6, 8, 10). Learnings from this research will help shape best practices for parallel algorithm optimisation on high-performance computing (HPC) systems, advancing the area of parallel computing.

3. Methodology

3.1 Hardware and software environment

3.1.1 Hardware configuration

The Univaq Supercomputer, a state-of-the-art high-performance computing (HPC) system with cutting-edge hardware components optimised for parallel computation, was used for the experiments:

- CPU Architecture: Based on Intel Xeon processors, which are well-known for their parallel task performance and scalability.
- Accelerators: By utilising parallel processing capabilities, NVIDIA Tesla GPUs embedded into the system expedite compute-intensive activities.
- Interconnect: Makes use of InfiniBand, a high-speed interconnect technology, to provide high bandwidth and low latency communication between compute nodes, both of which are essential for effective parallel computing.
- Memory and Storage: Designed to handle massive datasets and guarantee quick data access during computations, reducing processing bottlenecks, with plenty of RAM and quick storage options.

3.2 Environment for Software

The Univaq Supercomputer's software stack offers a strong environment designed for applications using parallel computing:

- **Operating System:** This system is based on Linux and is tuned for high performance and high clustering situations. It offers broad support for scientific computing and is stable and fast.
- **Libraries and Programming Languages:** Python 3.8 is the main programming language used to create programmes for parallel computing because of its

abundance of scientific libraries and versatility. Python bindings for MPI (Message Passing Interface) are known as MPI4Py, and they enable effective coordination and communication across processes that span several cores or nodes. NumPy: Offers effective data structures and optimized algorithms necessary for numerical calculations; good for parallel tasks like matrix multiplication. Matplotlib is a tool that helps with data interpretation by creating plots and visualizations to analyze performance metrics obtained from parallel computing research. PowerShell with Linux Commands: Scripts written in PowerShell were utilized to execute Linux commands on the supercomputer. These scripts facilitated tasks such as code deployment, job submission, and system monitoring, enhancing automation and workflow efficiency in HPC environments.

4. Implementation

4.1 Matrix multiplication algorithm

Parallel matrix multiplication, a basic operation in computational mathematics and scientific computing, is at the centre of this effort. Carefully thought out algorithmic implementation and parallelization method were used to maximise the Univaq

Supercomputer's processing capabilities.

4.2 Algorithm Overview

Matrix multiplication is the process of multiplying two matrices, A and B, to get matrix C = A × B. The complexity of the conventional sequential method is $O(n3)$, where n is the matrix dimension. Computation time is greatly decreased by parallelizing this process across several compute nodes, particularly for big matrices.

4.3 Parallelization Strategy

1. Data Partitioning: Using MPI, the matrix A, with dimensions of N × N, is split up into smaller sections that are then dispersed among the available compute nodes. A piece of A and the entire matrix B, which is shared by all nodes, are sent to each compute node.

2. Parallel Computation: Using local data (the submatrix of A and the entirety of B), each compute node separately computes its assigned submatrix of C.

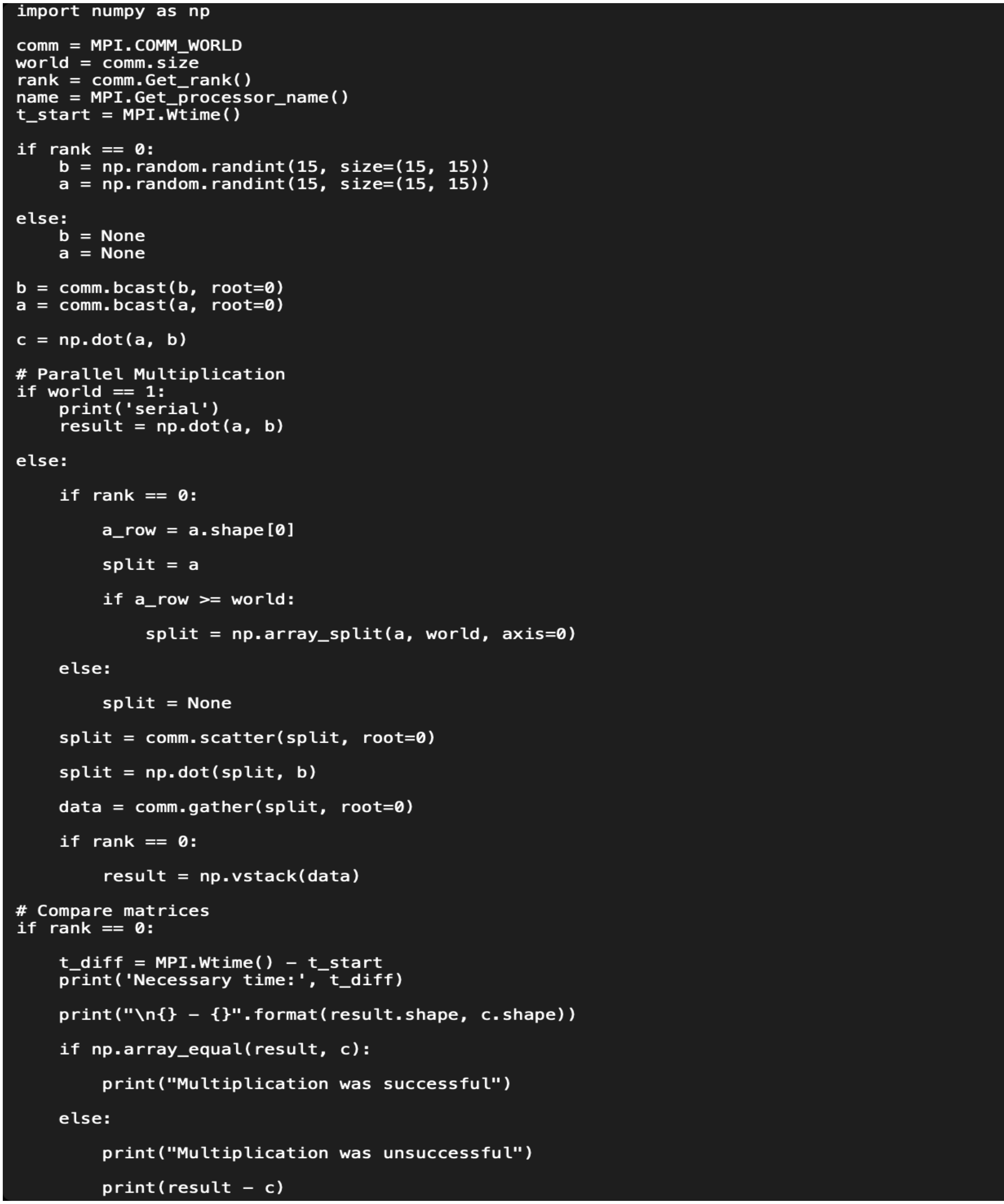
• MPI facilitates inter-process communication so that nodes can synchronise calculations and share essential data.

3. Result Aggregation: • The aggregation phase ensures consistency by combining partial results from all nodes into a cohesive output. • Results computed by individual nodes (partial matrices Ci) are gathered and integrated into the final outcome matrix C.

4.4 Practical Implementation

The script (matrix-mult.py) in Python

For parallelization, the implementation makes use of MPI (via MPI4Py) and Python. A sample that highlights the main features of the matrix multiplication implementation is provided below:



```
import numpy as np

comm = MPI.COMM_WORLD
world = comm.size
rank = comm.Get_rank()
name = MPI.Get_processor_name()
t_start = MPI.Wtime()

if rank == 0:
    b = np.random.randint(15, size=(15, 15))
    a = np.random.randint(15, size=(15, 15))

else:
    b = None
    a = None

b = comm.bcast(b, root=0)
a = comm.bcast(a, root=0)

c = np.dot(a, b)

# Parallel Multiplication
if world == 1:
    print('serial')
    result = np.dot(a, b)

else:

    if rank == 0:

        a_row = a.shape[0]

        split = a

        if a_row >= world:

            split = np.array_split(a, world, axis=0)

    else:

        split = None

    split = comm.scatter(split, root=0)

    split = np.dot(split, b)

    data = comm.gather(split, root=0)

    if rank == 0:

        result = np.vstack(data)

# Compare matrices
if rank == 0:

    t_diff = MPI.Wtime() - t_start
    print('Necessary time:', t_diff)

    print("\n{} - {}".format(result.shape, c.shape))

    if np.array_equal(result, c):

        print("Multiplication was successful")

    else:

        print("Multiplication was unsuccessful")

        print(result - c)
```

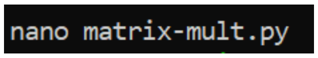
4.5 Execution workflow

• File Transfer: Using FileZilla, data and code files (such as matrix-mult.py) are moved between local computers and the Univaq Supercomputer, guaranteeing smooth workflow management and data integrity.

• PowerShell for Job Execution: To submit and manage jobs, Linux commands are executed via PowerShell scripts that interface with the supercomputer. Setting parameters for parallel execution, such as the number of nodes, is part of this.

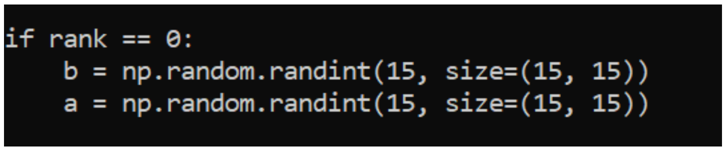
• Task Submission: PowerShell-executed Linux commands that specify the script (matrix-mult.py) and required parameters start task submissions on the supercomputer. A sample showing important facets of the job's execution is provided below.

To edit the matrix, we used the code.



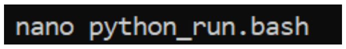
```
nano matrix-mult.py
```

The part of the code changed for 15*15 matrix is seen below.



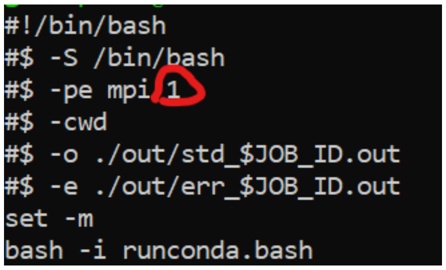
```
if rank == 0:
    b = np.random.randint(15, size=(15, 15))
    a = np.random.randint(15, size=(15, 15))
```

To change the nodes, we used the code



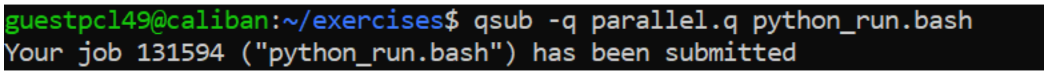
```
nano python_run.bash
```

Then the circled value is the node we changed: starting from 1, 2, 4, 6, 8, 10



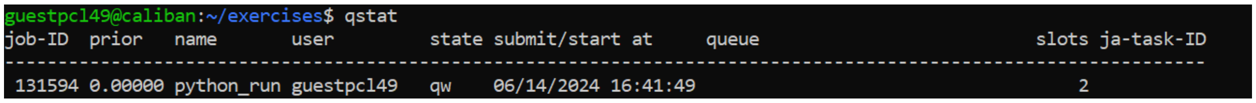
```
#!/bin/bash
#$ -S /bin/bash
#$ -pe mpi 1
#$ -cwd
#$ -o ./out/std_$JOB_ID.out
#$ -e ./out/err_$JOB_ID.out
set -m
bash -i runconda.bash
```

Then we ran this to execute it



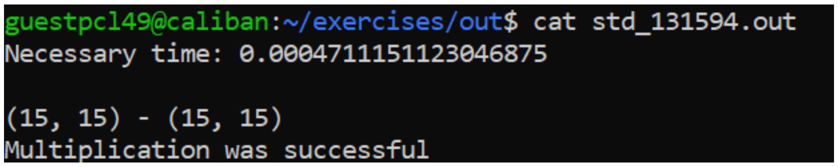
```
guestpcl49@caliban:~/exercises$ qsub -q parallel.q python_run.bash
Your job 131594 ("python_run.bash") has been submitted
```

qstat to see the status of the executed code



```
guestpcl49@caliban:~/exercises$ qstat
job-ID  prior   name       user         state submit/start at     queue                          slots ja-task-ID
-----------------------------------------------------------------------------------------------------------------
 131594 0.00000 python_run guestpcl49   qw    06/14/2024 16:41:49                                    2
```

We view the standard output of using 2 nodes for instance.



```
guestpcl49@caliban:~/exercises/out$ cat std_131594.out
Necessary time: 0.0004711151123046875

(15, 15) - (15, 15)
Multiplication was successful
```

And the same process for all other nodes

5. Results

5.1 Perfomamce metrices analysis

The Univaq Supercomputer's parallel matrix multiplication operation provided important new information about the effectiveness and scalability of the applied method. Key performance measures, such as speedup, efficiency, and overhead, were calculated using the execution times obtained for different numbers of compute nodes to assess the performance.

5.2 Execution Times

The table below shows the execution times for different numbers of nodes:

Node	**Time *(T)***
1	0.000231981277465820000
2	0.000471115112304687000
4	0.000771999359130859000
6	0.000807046890258789000

8	0.0010051727294921800
10	0.00087380409240722600

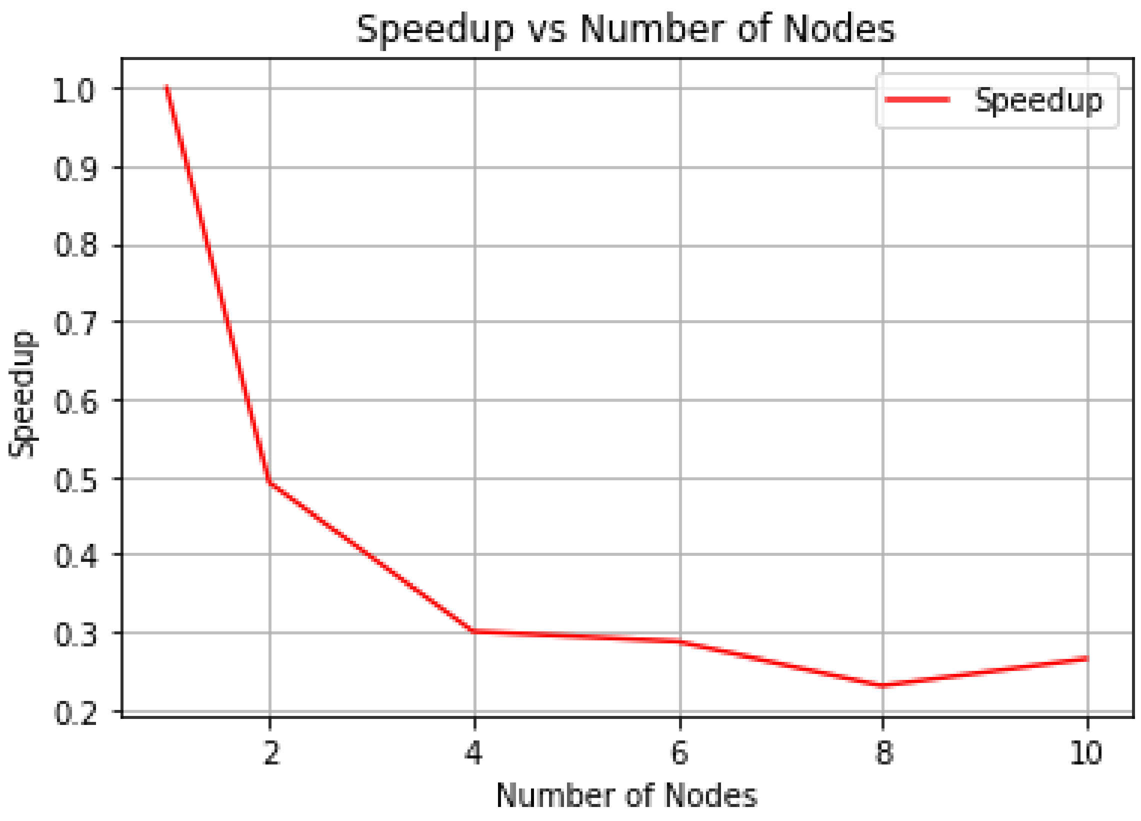
5.3 Speedup

The relative performance gain that parallel execution yields over sequential execution is measured by Speedup (S).

$$S = \frac{T_{serial}}{T_P}$$

where T_{serial} is the execution time on a single node and T_P is the execution time with **p** nodes, is the formula used to compute it.

Based on these execution times, the speedup values computed show the following trend:

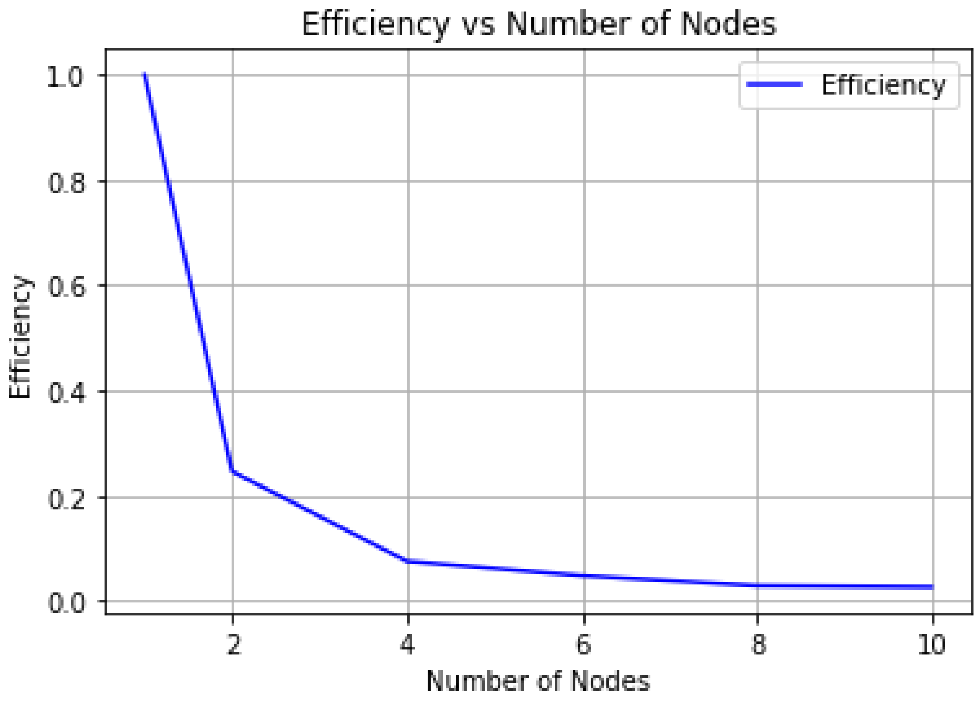


5.4 Efficiency

Efficiency (E), which is computed as **E=S/p**, indicates how well parallel resources (nodes) are used.

where p is the number of nodes and S is the speedup.

The efficiency values corresponding to the speedup results are:

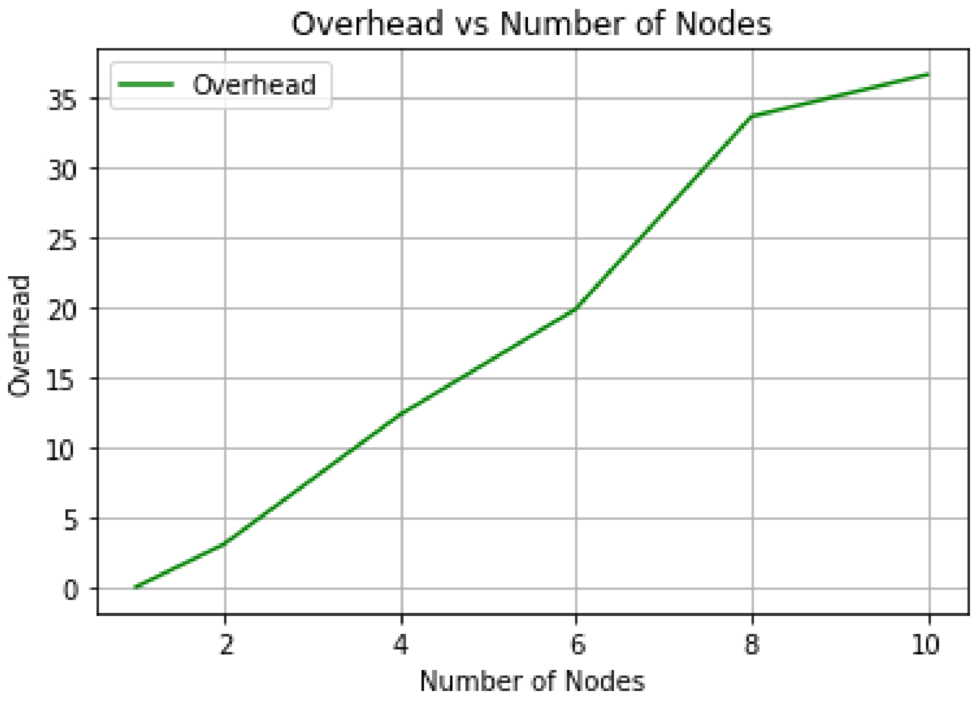


5.5 Overhead

The extra computational resources used as a result of parallelization are measured by overhead (O), which is given by

$$O = \frac{p \times T_p - T_{serial}}{T\ serial}$$

The overhead value the following overhead values were calculated using the execution times:



Overhead vs Number of Nodes
Overhead
Overhead
Number of Nodes
0
5
10
15
20
25
30
35
2
4
6
8
10

6. Analysis

• The Univaq Supercomputer's parallel matrix multiplication performance characteristics can be understood through the examination of speedup, efficiency, and overhead measurements.

• The measured speedup values show that while increasing the number of nodes initially boosts computing performance, there are decreasing returns as the node count rises. This phenomena draws attention to the overhead and inefficiencies—such as load imbalance and communication ddelays—that come with parallelization.

• Efficiency measures highlight even more how difficult it is to maintain great computational efficiency when working with several nodes. As the number of nodes increases, efficiency declines, which is indicative of overhead expenses and possible inefficiencies in task distribution and synchronisation. These results imply that although parallelization might speed up processing, careful thought is required to maximise resource use and reduce performance deterioration.

• Scaling the number of nodes results in significant additional processing expenses, as the overhead study demonstrates. Larger parallel setups result in increased levels of idle time, synchronisation delays, and communication overhead. To maximise the advantages of parallel computing and increase overall system efficiency, these overhead elements must be addressed.

7. Conclusion

• Scalability and Performance: Achieving near-linear scalability is possible since with one node, the observed execution time is approximately 0.00023198 and when we double the number of nodes (i.e. two nodes), the execution time approximately doubles (0.0004711) and only increases afterwards.

• Overhead vs. Number of Nodes: Although there is an initial increase in overhead with fewer nodes, the overhead continues to rise significantly as more nodes are added, indicating that communication and synchronization costs increase with the number of nodes.

• Efficiency vs. Number of Nodes: Although there is an initial gain in efficiency with fewer nodes, achieving high efficiency is more difficult as the number of nodes increases, due to the overhead becoming a larger fraction of the total computation time.

8. Appendices

Python Scripts: Full listings of matrix-mult.py and 15 by 15 Matrix.py scripts.