Parallel Programming Lab Book

School of Computing

M30248 Parallel Programming

Date Issued: 26 September 2025

Assessment Item number: 1

Module Coordinator: Dr. Hamidreza Khaleghzadeh

Student ID:

UP2154598

UP2540661

UP2543144

Table of Content

List of Tables

Lists of Figures

1. Lab 01 - Java Threads for Parallelism

1.1 Introduction

In this session we will be looking at a mathematical function to approximate π using multiple thread to accelerate a calculation task as an introduction to reveal the true power of the parallel programming. The formula to approximate π is listed below:

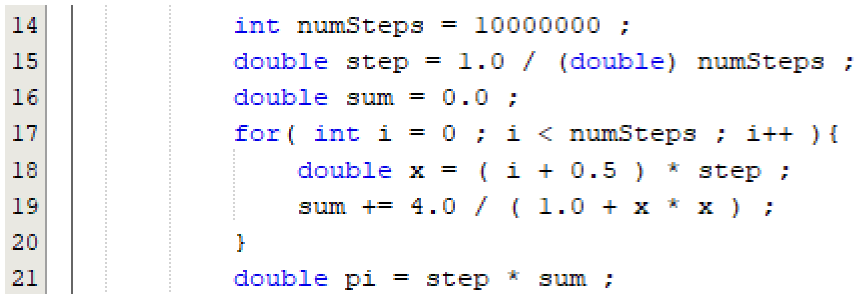
$$\pi \approx h \sum_{i=0}^{N-1} \frac{4}{1 + (h(i + \frac{1}{2}))^2}$$

Where ***N*** is the number of integration steps and ***h = 1/N*** is the step size. The computational workload is inherently parallelisable, as the summation consists of independent terms that can be distributed across multiple threads.

1.2 Approaches and Result

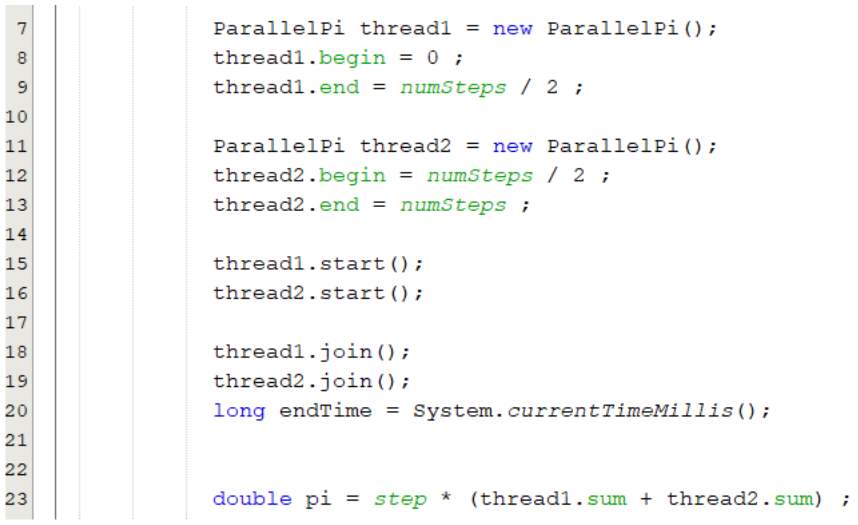
1.2.1 Methodology

Firstly, we need to establish a baseline for the task, using Java codes to implement the mathematical function and run the programme sequentially to get the baseline running time.



```java
int numSteps = 10000000 ;
double step = 1.0 / (double) numSteps ;
double sum = 0.0 ;
for( int i = 0 ; i < numSteps ; i++ ){
    double x = ( i + 0.5 ) * step ;
    sum += 4.0 / ( 1.0 + x * x ) ;
}
double pi = step * sum ;
```

Subsequently, a parallel version is used with multi-thread coding. This divides the total number of steps, numSteps, into contiguous ranges, assigning each range to a separate thread instance of a class extending Thread. Each thread will calculate a partial sum of the π and used the join() method to synchronise and aggregate the results. A sum will be added to obtain the final value of π. During this lab session, we will be looking at the performance of parallel programming with 2, 4, and 8 threads.



```java
ParallelPi thread1 = new ParallelPi();
thread1.begin = 0 ;
thread1.end = numSteps / 2 ;

ParallelPi thread2 = new ParallelPi();
thread2.begin = numSteps / 2 ;
thread2.end = numSteps ;

thread1.start();
thread2.start();

thread1.join();
thread2.join();
long endTime = System.currentTimeMillis();


double pi = step * (thread1.sum + thread2.sum) ;
```

The performance measurements are taken using currentTimeMillis() functions in JAVA. A starting time, ***Ts***, will be recorded when the programme starts and another time, ***Te***, will be recorded when the threads finish its calculation and sum up the value of π using currentTimeMillis(). The total time consume, ***T***, will be time difference between ***Te*** and ***Ts***. To obtain more precise timings, the experiment was repeated using nanoTime() function, which provides nanosecond resolution. All benchmarks were conducted on a system with a multi-core processor.

A new parameter, ***parallel speedup***, is calculated by using the sequential time and the parallel time under different core numbers to evaluate the ability of parallel programming.

$$parallel\ speedup = \frac{sequential\ time}{parallel\ time}$$

1.2.2 Results and Analysis

The initial benchmarking was performed with numSteps = 10,000,000 using currentTimeMillis() function.

Nodes	1st Time (ms)	2nd Time (ms)	3rd Time (ms)	4th Time (ms)	5th Time (ms)	Average Time (ms)	Parallel Speedup	Efficiency (%)
Sequential	51	50	51	50	50	50.4	1.00	100.00
2	31	30	30	29	31	30.2	1.67	83.50
4	23	22	21	21	20	21.4	2.36	59.00
8	19	20	19	20	17	19.0	2.65	33.10

Table1.2.2.1: The Relation of Parallel Speedup with Different number of Threads in Millisecond.

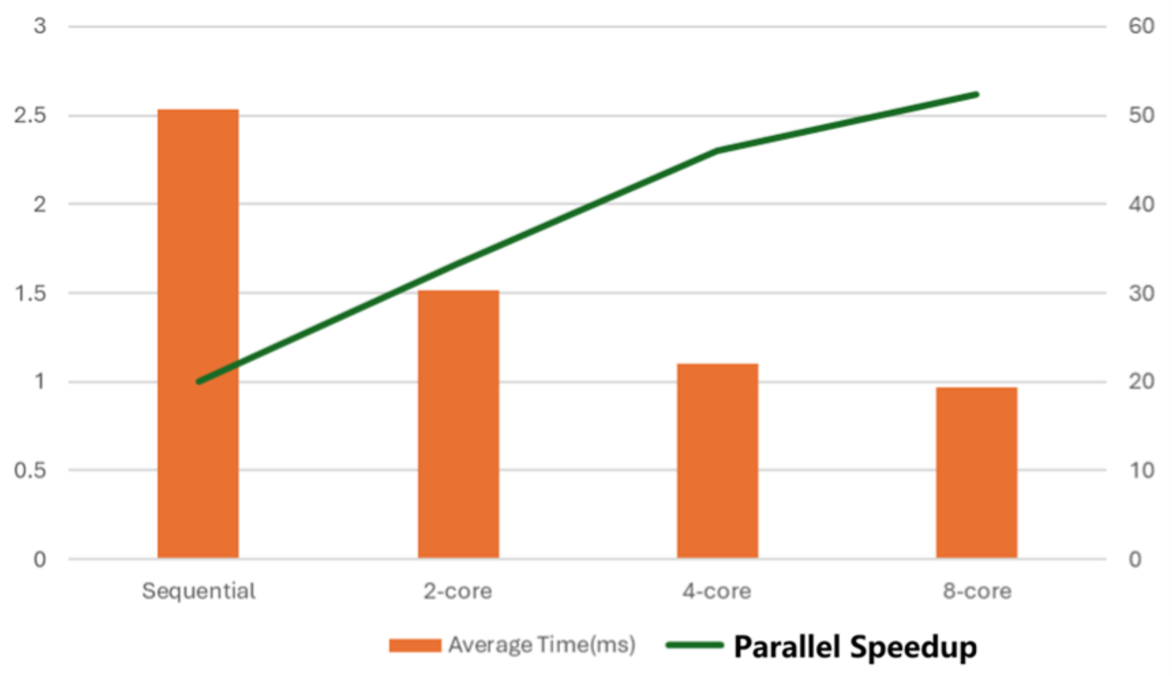


Figure1.2.2.1: Total Time (ms) under Different Core Numbers and Parallel Speedup

The results demonstrate a clear benefit from parallelisation. The 2-thread implementation achieves a 1.67 speedup. The 4-thread version shows a more substantial speedup of 2.36, shows nearly a linear increase of parallel speedup as the thread number increases. However, the progression from 4 to 8 threads yields a diminished return, with a speedup of only 2.65. The increases are no longer linear.

1.2.3 Increase Time Precision

To achieve a more accurate measurement, the experiment was repeated using nanoTime(). The results, displayed below, corroborate the trends observed with millisecond timings but provide a more granular and reliable dataset.

Nodes	**1st Time (ns)**	**2nd Time (ns)**	**3rd Time (ns)**	**4th Time (ns)**	**5th Time (ns)**	**Average Time (ns)**	**Parallel Speedup**	**Efficiency (%)**
Sequential	53176600	49276000	50155900	50968700	50770300	50869500	1.00	100.00
2	31033000	28903500	28813300	29313200	29853300	29583260	1.72	86.00
4	22,784100	21838199	21191000	22037700	21837800	21937760	2.32	58.00
8	21337900	18998200	21126700	20189500	21792400	20490940	2.48	31.00

Table1.2.3.1: The Relation of Parallel Speedup with Different number of Threads in Nanoseconds.

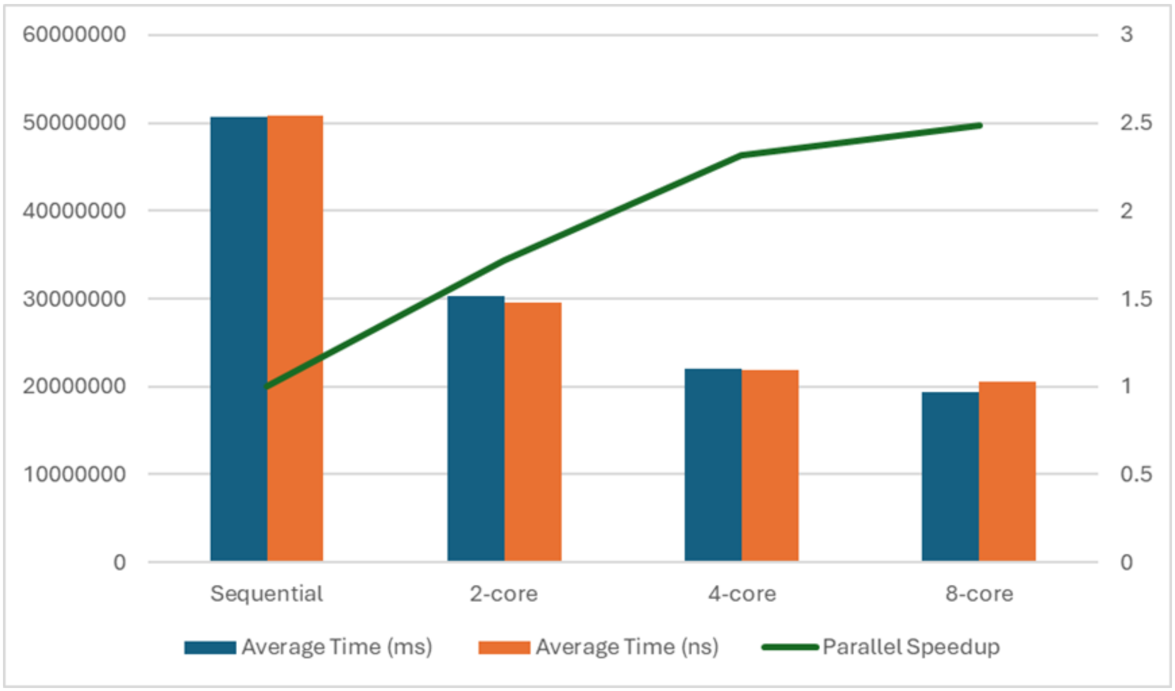


Figure1.2.3.1: Total Time (ns) of Different Core Numbers and Parallel Speedup

The use of nanoTime() is unequivocally superior for micro-benchmarking. The nanosecond resolution captures the true execution time more accurately than millisecond timings, which can be subject to rounding errors for short-duration computations. This is evident in the more consistent speedup ratios calculated from the nanosecond data. The observed speedups (1.72, 2.32, 2.48) provide a more precise measure of the parallel efficiency. Despite the time precision, both graphs of parallel speedup show the same trend in parallel speedup decrease when we increase to 8-core.

1.2.4 The Impact of Workload Size on Parallel Efficiency

The experiment compared the execution time of a sequential and a two-threaded parallel implementation of π (pi) computation using different iteration counts.

Nodes	**Sequential Speed (ms)**				
1,000,000	10	10	11	12	11
10,000,000	37	30	25	32	39
1,000,000,000	1930	2000	2285	2108	1979

Table1.2.4.1: Sequential Speed

Nodes	Parallel 2 Core Speed (ms)				
1,000,000	16	15	19	12	15
10,000,000	29	26	22	29	27
1,000,000,000	1272	1362	1271	1233	1283

Table1.2.4.2: Parallel 2 Core Speed

Nodes	Sequential Average (ms)	Parallel Average (ms)	Parallel Speedup	Efficiency (%)
1,000,000	10.80	15.40	0.71	35.50
10,000,000	32.60	26.60	1.23	61.50
1,000,000,000	2160.40	1284.00	1.62	81.50

Table1.2.4.3: The Relation of Parallel Speedup with Different number of Steps in Sequential and 2-Threads in Microsecond.

The data clearly shows that the performance benefit of parallelisation depends heavily on the problem size. For the smallest workload (1,000,000 iterations), the parallel version was slower than the sequential one, giving a speedup of 0.71. This happened because the computation time was too short to offset the overhead of thread creation, context switching, and memory synchronisation.

When the workload increased to 10,000,000 iterations, the parallel implementation began to outperform the sequential version, achieving a modest speedup of 1.23. This indicates that as the number of computations grows, the cost of thread management becomes relatively smaller compared to the total computation time.

At the highest workload of 1,000,000,000 iterations, the performance gain became much more pronounced, reaching a speedup of 1.63. This demonstrates that parallelisation becomes increasingly effective as the workload grows, since both threads can remain busy performing useful calculations for longer periods, reducing the impact of overhead and improving CPU utilisation across cores.

Overall, the trend suggests that for compute-intensive tasks, parallel execution can substantially reduce total computation time, but for smaller problems, it may degrade performance due to fixed overheads and timing limitations.

1.3 Conclusion

In summary, this lab successfully demonstrated the parallel programming using Java Threads. The implementation of a parallel π calculation resulted in significant performance improvements, with a parallel speedup of up to 2.48 observed on a multi-core system.

However, the experiment also showed that performance gains are not strictly proportional to the number of threads used. While increasing parallelism from one to four threads yielded substantial benefits, further increasing eight threads provided only marginal gains. The exercise underscored the necessity of using high-precision timing functions like nanoTime() for accurate performance analysis. This foundational understanding of performance scaling and measurement is crucial for developing efficient parallel applications in more complex scenarios.

In addition, it was observed that smaller workloads sometimes performed worse in the parallel version. This can be because of the time limitation of the thread creation, context switching and memory synchronisation.

2. Lab02 - Threads and Data

2.1 Introduction

In this session, we used the Mandelbrot set as a case study in investigating the role of problem decomposition in parallel programming. The Mandelbrot set is a classic example of an “embarrassingly parallel” problem because the value of the output image can be calculated independently. However, in this session the workload is not uniform; pixels inside the set require more iterations (up to CUTOFF) than those far outside.

Primarily while data parallelism is straight forward to implement, we need to understand the importance of memory architecture and selecting the right decomposing strategies for efficient parallel performance. The report benchmarks the different decomposition strategies for 2 and 4 threads, analysing the resulting parallel speedup and parallel efficiency.

2.2 Approaches and Analysis

2.2.1 Methodology

A Java-based Mandelbrot set calculator with an image size of 4096x4096 and a computational cutoff of 100 iterations. This image size was chosen to provide high visual resolution while maintaining a tractable computation time for benchmarking. The baseline performance in sequential time will be implemented first. The result will be coloured and saved in a png file as illustrated below in Figure 2.2.1.1.

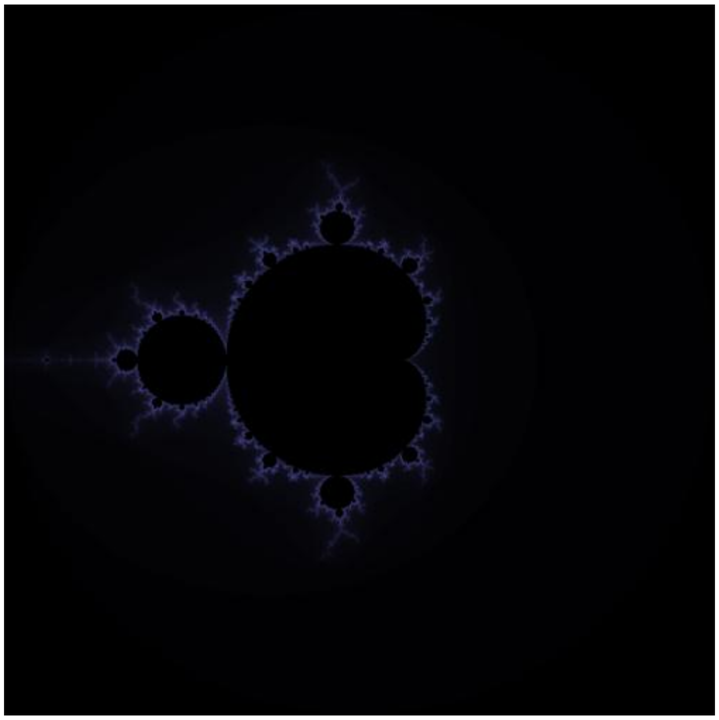


Figure2.2.1.1: The Mandelbrot Set

Using Java, we can compute parallel programming at the next step. Initially, we will be using 2-thread and divide the image into two sections. We will be testing the implementation with a horizontal division and a vertical division. Then, we will run parallel programming using four threads. This time, the process will be divided horizontally into four horizontal strips, and another way is divided into four squares (2x2).

After running each program multiple times to get an average running time in milliseconds, we can use benchmark running time when we run the process sequentially and obtain the Parallel Speedup to provide a representative performance measure.

2.2.2 Results and Analysis

The performance of the 2-thread implementations varied based on the decomposition strategy. The results are listed in Table2.2.2.1 below.

Nodes	**1st Time （ms)**	**2nd Time （ms)**	**3rd Time （ms)**	**4th Time （ms)**	**5th Time （ms)**	**Average Time （ms)**	**Parallel Speedup**	**Efficiency (%)**
Sequential	613	869	672	624	883	636	1.00	100.00
Horizontal Cut	476	322	448	325	322	323	1.96	98.00
Vertical Cut	483	616	615	613	622	616	1.03	51.50

Table2.2.2.1: Performance of Two-Thread Decomposition Strategies in milliseconds.

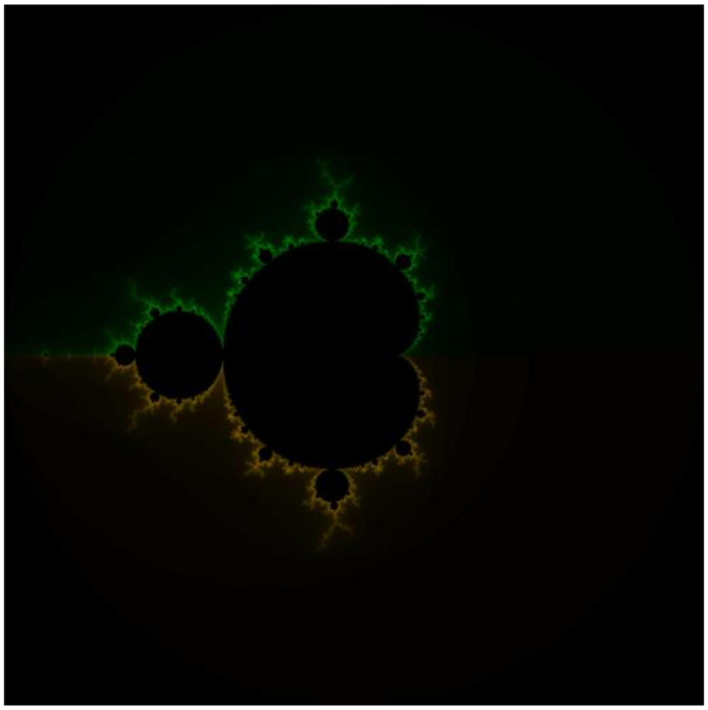


Figure2.2.2.1: Horizontal (i-loop) Decomposition with 2 Threads

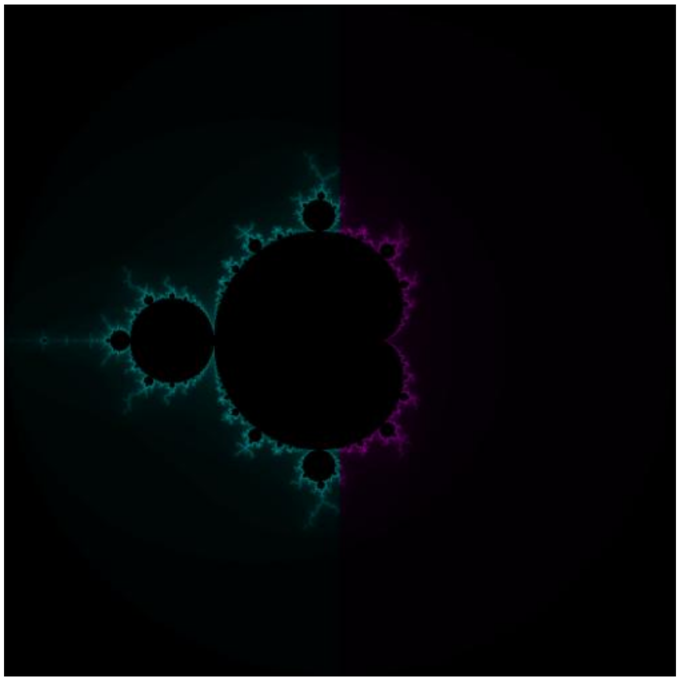


Figure2.2.2.2: Vertical (j-loop) Decomposition with 2 Threads.

The horizontal cut method achieved a significantly higher speedup (1.96) than the vertical cut. This result indicates that the distribution of computational workload is not symmetric about the vertical axis, which matched graph produced by the program after colouring different processed regions. It also shows the effect of the decomposition method on efficiency.

The drastically lower speedup of the vertical division (1.03) suggests a severe load imbalance. This implies that the computationally expensive regions of the Mandelbrot set are not distributed evenly across the left and right halves of the image in this specific implementation.

The result when using 4-thread is illustrated in Table2.2.2.2 below.

Nodes	**1st Time （ms)**	**2nd Time （ms)**	**3rd Time （ms)**	**4th Time （ms)**	**5th Time （ms)**	**Average Time （ms)**	**Parallel Speedup**	**Efficiency (%)**
Sequential	613	869	672	624	883	636	1.00	100.00
4 Horizontal Strip	549	519	510	522	515	523	1.21	30.30
Squares (2x2)	318	311	316	307	230	313	2.03	50.80

Table 2.2.2.2: Performance of Four-Thread Decomposition Strategies in milliseconds.

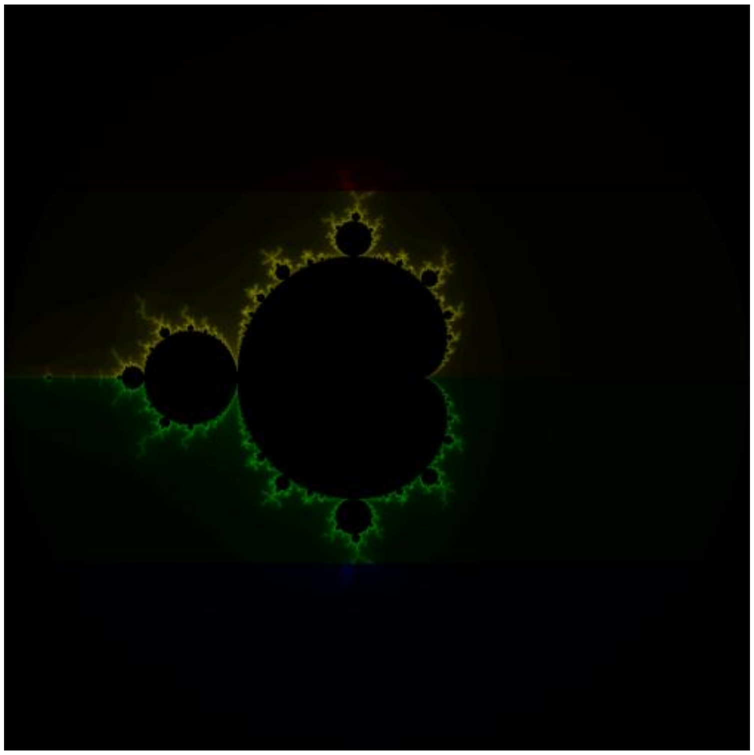


Figure2.2.2.3: Horizontal Strip Decomposition with 4 Threads

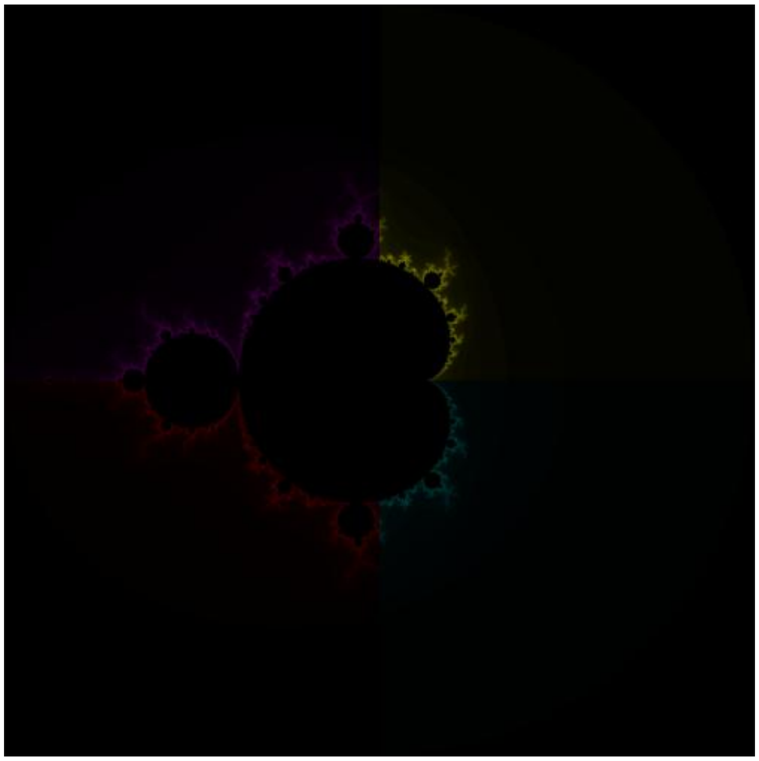


Figure2.2.2.4: 4 Squares Decomposition (2x2) with 4 Threads

The parallel speedup of decomposing area into 2x2 squares using 4-thread is 2.03. This method significantly outperformed the performance of divided the graph into 4 horizontal strips, which only achieved 1.21 in the Parallel Speedup. The parallel speedup of four horizontal strips using 4-thread (1.21) is even lower than the parallel speedup of one horizontal cut using 2-thread (1.96). Splitting into four horizontal strips likely exacerbates the imbalance of the computational workload.

The process of 2x2 squares using 4-thread is by far one of the most efficient decomposing methods that we have tested as it creates smaller work units. This method allows for a much better distribution in the computational workload across all 4 threads and has a parallel speedup of 2.03.

These results conclusively demonstrate that the optimal decomposition strategy is highly dependent on the problem's data domain and memory structure. On the other hand, increasing the number of threads using the same cutting methods does not guarantee an increase in parallel speedup.

2.3 Conclusion

This Lab provided a practical demonstration of the concepts of problem decomposition and load balancing. It clearly showed achieving higher parallel speed is not just based on the dividing data among threads;

The results illustrate that the decomposition strategy must match the problem's structure. For irregular workloads like the Mandelbrot set, careful work distribution is essential to maximise processor utilisation and achieve significant performance gains from parallel processing.

3. Lab 03 - Workload Decompositions and Simulations

3.1 Introduction

Conway's Game of Life represents a class of simulation problems that are not embarrassingly parallel. Unlike the Mandelbrot Set (previous week's exercise), each cell's future state depends on its neighbours' current states, requiring careful synchronisation between threads to ensure correctness. In this session, we focus on workload decomposition strategies and parallelising simulation types using block-wise decomposition.

3.2 Approaches and Analysis

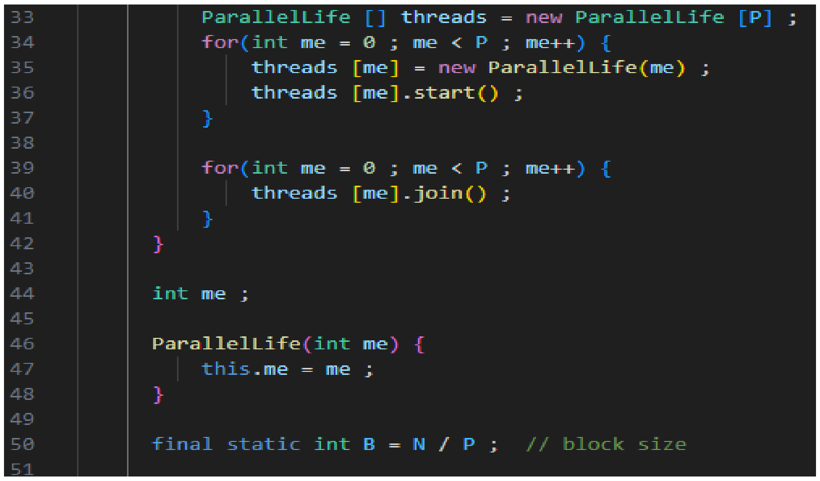
3.2.1 Methodology

This program simulates Conway's Game of Life a cellular automaton where cells on a grid live, die, or reproduce based on simple rules. It is like watching a colony of organisms evolve over time. In the sequential code we tried experimenting on different values like the constant N, CELL-SIZE and DELAY to see how it affects the display of the code.

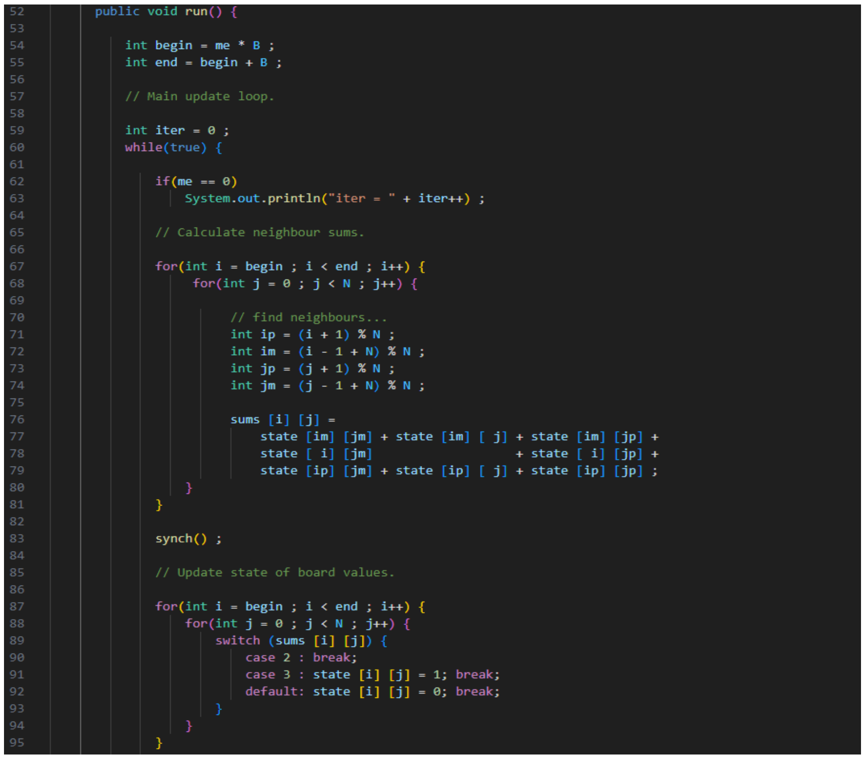
For parallelisation, we moved the main update loop into the "run method" of the ParallelLife class extending Thread, we then decomposed both computational loops (neighbour sum calculation and state update) using block wise decomposition we also made sure to keep I/O operations (printing and display) on thread 0 only, and lastly, we Implemented barrier synchronisation between computational phases using the "java.util.concurrent.CyclicBarrier" .

The parallel game of life employs a block decomposition strategy were the 256*256 grid is divided into horizontal strips, with each of the P threads assigned B=N/P consecutive rows to process. Each threads calculates neighbour sums independently and update cell states for its assigned rows.

In the program we used a two-barrier synchronisation pattern implemented with "Java's CyclicBarrier to prevent race conditions. The first barrier prevents reading partially updated neighbour sums, while the second barrier prevents mixing data from different iterations. Without the barriers you'd get race conditions where threads read inconsistent data, producing an incorrect game of life evolution



```java
        ParallelLife [] threads = new ParallelLife [P] ;
        for(int me = 0 ; me < P ; me++) {
            threads [me] = new ParallelLife(me) ;
            threads [me].start() ;
        }

        for(int me = 0 ; me < P ; me++) {
            threads [me].join() ;
        }
    }

    int me ;

    ParallelLife(int me) {
        this.me = me ;
    }

    final static int B = N / P ;  // block size

```



```java
    public void run() {

        int begin = me * B ;
        int end = begin + B ;

        // Main update loop.

        int iter = 0 ;
        while(true) {

            if(me == 0)
                System.out.println("iter = " + iter++) ;

            // Calculate neighbour sums.

            for(int i = begin ; i < end ; i++) {
                for(int j = 0 ; j < N ; j++) {

                    // find neighbours...
                    int ip = (i + 1) % N ;
                    int im = (i - 1 + N) % N ;
                    int jp = (j + 1) % N ;
                    int jm = (j - 1 + N) % N ;

                    sums [i] [j] =
                        state [im] [jm] + state [im] [ j] + state [im] [jp] +
                        state [ i] [jm]                   + state [ i] [jp] +
                        state [ip] [jm] + state [ip] [ j] + state [ip] [jp] ;
                }
            }

            synch() ;

            // Update state of board values.

            for(int i = begin ; i < end ; i++) {
                for(int j = 0 ; j < N ; j++) {
                    switch (sums [i] [j]) {
                        case 2 : break;
                        case 3 : state [i] [j] = 1; break;
                        default: state [i] [j] = 0; break;
                    }
                }
            }
```

3.2.2 Results and Analysis

The performance and behavioural characteristics of the parallel Game of Life implementation were systematically evaluated through controlled experiments varying the grid dimension (N) and thread count (P). Experimental observations indicate that the number of iterations required to reach stable configurations exhibits a positive correlation with grid size. For instance, configurations with N = 128 and P = 128 typically achieved stability around 2000 iterations, whereas larger grids of N = 256 required approximately 3000 iterations to reach analogous stable states. And when the N = 256, P = 128, it takes about 4800 iterations to reach a stable state.

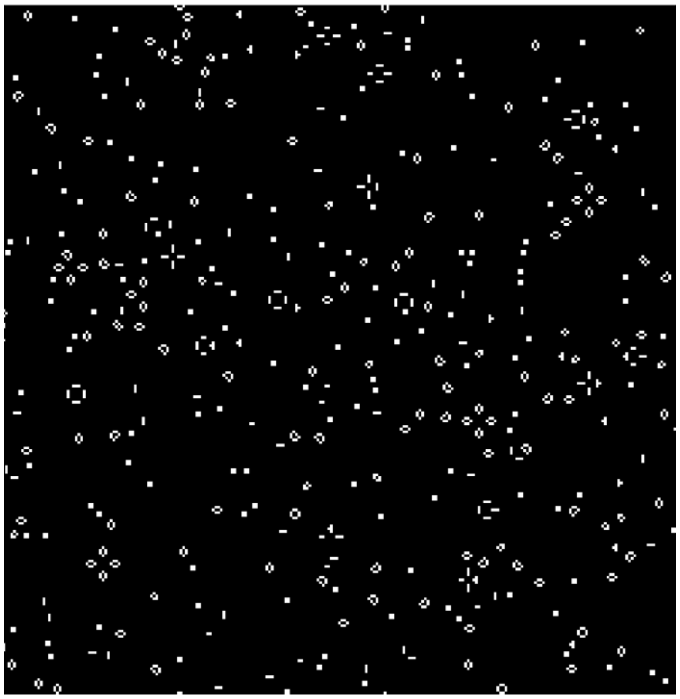


Figure3.2.2.1: The Game of Life Reached a Stable State.

This indicates that despite the computational force increasing with the grid size, the number of iterations will also increase with the size. The parameter DELAY will force the display to freeze for a given time, and the parameter CELL_SIZE only affects the display's cell size. The rough number of iterations to reach a stable state remains unchanged.

A fundamental constraint emerged during experimentation: the implementation requires N ≥ P to function correctly. When P exceeds N, the block decomposition calculation B = N / P produces zero due to the nature of Java. Consequently, all threads received empty work ranges, resulting in a dead-lock situation where the initial random state persists indefinitely without evolution.

This condition arises because the block decomposition strategy assigns each thread a contiguous row block of size B. When P > N, B becomes zero, leaving no row for processing. We can modify the block to ensure that when it happens, we reassigned the block size to 1. However, this would leave excess threads idled.

3.3 Conclusion

This lab showed how Conway's Game of Life can be parallelised using block-wise decomposition while still requiring careful coordination between threads to keep the simulation accurate. Splitting the grid into horizontal sections and synchronising them with a dual barrier approach allowed the program to run correctly on multiple threads, though performance depended heavily on the balance between grid size and the number of threads. Larger grids took longer to stabilise and ensuring that the grid size was at least as large as the thread count was crucial to avoid invalid partitions and deadlocks. Overall, the exercise demonstrated that although the Game of Life can benefit from parallelism, it is not naturally parallel, and effective results rely on proper synchronisation, thoughtful workload division, and matching the problem size to the available parallel resources.

4. Lab 04 - Parallel Programs with Interacting Threads

4.1 Introduction

This lab explores the numerical solution of the two-dimensional Laplace equation using the relaxation method, implemented both sequentially and in parallel using Java threads. The Laplace equation is a fundamental partial differential equation that appears in various fields of physics and engineering, the sequential relaxation method involves iteratively updating each grid point based on the average neighbours, this algorithm presents significant opportunities for parallel execution. The aim of this experiment is to parallelise the Laplace solver using a multi-threaded approach with barrier synchronisation, benchmark its performance under thread counts, and critically evaluate the impact of synchronisation on parallel efficiency.

4.2 Approaches and Analysis

4.2.1 Methodology

The sequential Laplace equation solver was initially executed to establish a performance baseline. Rendering the graphical output is computationally expensive. Therefore, to minimise this overhead, visualisation was limited to every 1,000 iterations rather than each individual iteration. This speed up the process by reducing the number of times we need for illustration while shows the process of our simulation.

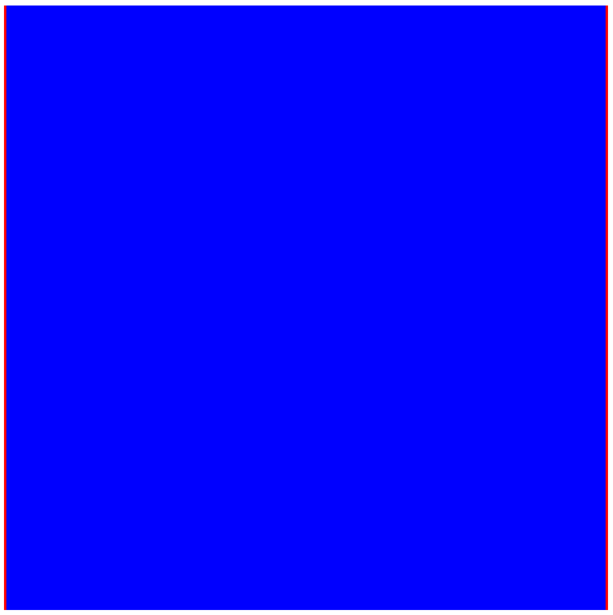


Figure4.2.1.1: The Initial State of the Simulation.

The boundary conditions for Laplace's equation were defined as follows: the left and right edges were maintained at a fixed voltage of 1.0, while the top and bottom edges were held at 0.0. This configuration models a hollow conductive box with prescribed potentials on opposing faces. The maximum value is displayed in red, and 0 values are displayed in blue. That is the reason why both left and right edges in the picture above are red, while the rest of the area are in blue.

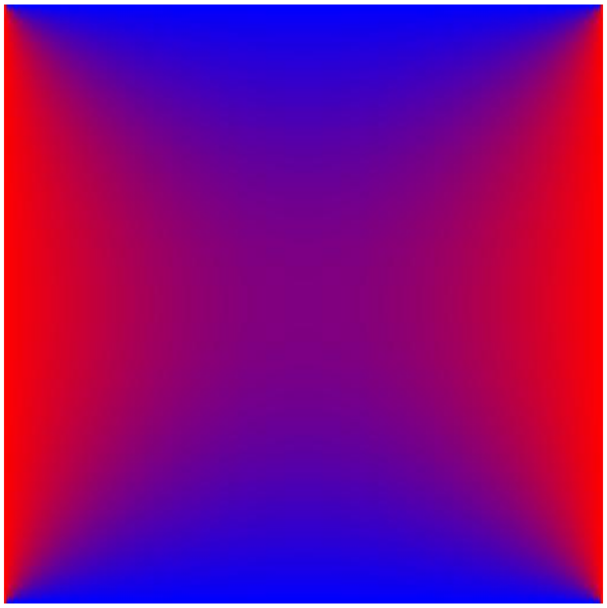


Figure4.2.1.2: The Simulation After 100,000 Iterations.

The simulation after 100,000 iterations is illustrating above. The result fit with the expectation. The sequential Laplace solver was parallelised adopting the methodology demonstrated in the Game of Life case study from Chapter 3. The computational domain was decomposed using a block partitioning strategy, wherein the two-dimensional grid was divided into horizontal strips allocated to individual threads. Each Thread is responsible for updating a contagious block of rows.

Two CyclicBarrier synchronisation points were implemented to ensure computational correctness, following the principles established in Chapter 3. The first barrier guarantees that all threads complete the calculation phase before any thread proceeds to update shared state variables, thereby preventing race conditions where threads might access inconsistent generations of data. The code has two barrier synchronisations. The first barrier ensures all threads complete the calculation of the new values (new phi). The second barrier ensure all threads finish the updating new values before proceeding to the next iteration.

Furthermore, to quantify the overhead attributable specifically to synchronisation, a modified version of the parallel implementation was executed with both barrier calls removed. While this unsynchronised variant produces numerically incorrect results due to race conditions, this data allows us to analyse the performance impact of synchronisation mechanisms.

All experiments employed a fixed grid size of N = 256 across 100,000 iterations. To ensure that performance measurements reflected computational workload rather than graphical rendering overhead, visualisation was triggered only every 1,000 iterations. Each configuration was expected five times, and average execution time was used for analysis.

4.2.2 Result and Analysis

With Barrier synchronisation

Threads	1st(ms)	2nd (ms)	3rd (ms)	4th (ms)	5th (ms)	Mean (ms)
Sequential	5076	5089	5000	5119	5140	5084.80
2	3877	3838	3831	3854	3915	3863.00
4	5125	5377	5551	6237	5761	5610.20
8	10547	10751	11438	11374	11039	11029.80

Table 4.2.2.1: Barrier synchronisation

Without Barrier synchronisation:

Threads	1st (ms)	2nd (ms)	3rd (ms)	4th (ms)	5th (ms)	Mean (ms)
2	2670	2625	2772	2673	2725	2693
4	1705	1751	1735	1729	1686	1721
8	1434	1425	1679	1429	1432	1479

Table 4.2.2.2: Without Barrier synchronisation

Threads (P)	Mean Time (ms)	Parallel Speedup	Efficiency (%)
Sequential	5084.80	1.00	100.00
2	3863.00	1.32	65.90
4	5610.20	0.91	22.70
8	11029.80	0.46	5.70

Table 4.2.2.3: Performance Analysis (with Barrier)

With the barrier enabled, the performance degrades as threads increase beyond 2, indicating significant synchronisation and cache contention overhead. The 4-thread and 8-thread versions performed slower than sequential due to excessive barrier waits and thread management costs.

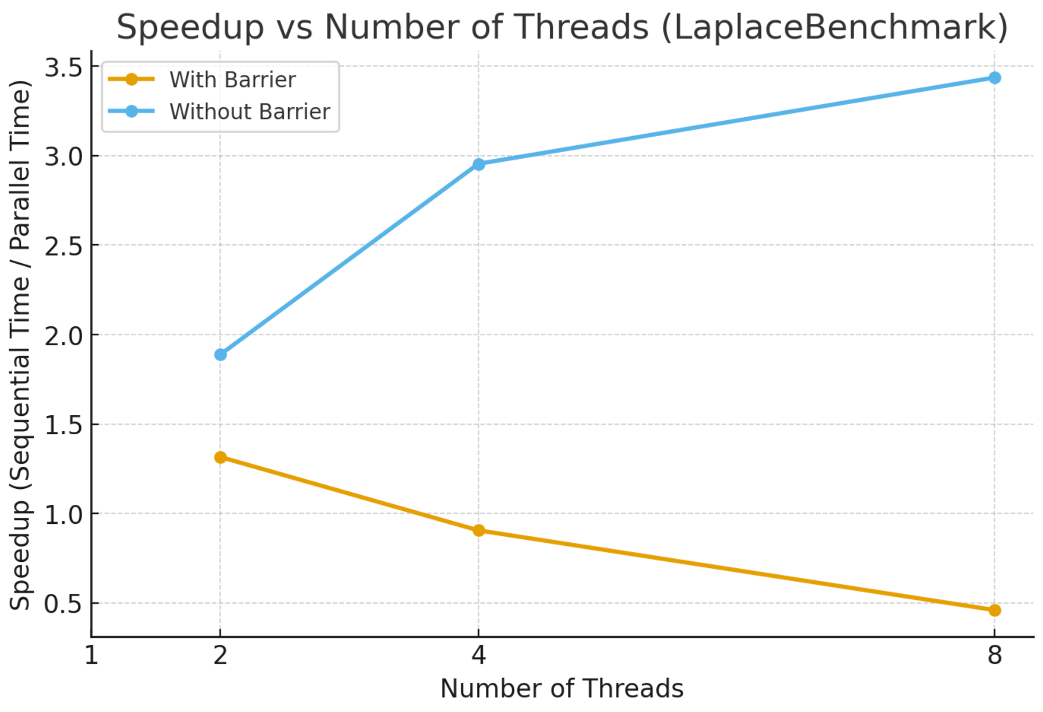


Figure4.2.2.1: Speedup vs Number of Threads.

Threads (P)	Mean Time (ms)	Parallel Speedup	Efficiency (%)
2	2693.00	1.89	94.40
4	1721.20	2.95	73.70
8	1479.80	3.43	42.90

Table 4.2.2.4: Performance Analysis (without Barrier)

When synchronisation barriers were disabled, execution time decreased dramatically and parallel speedup increased close to ideal between 2 to 4 threads. However, correctness was not guaranteed, since threads were not synchronised between iterations. This clearly quantifies the barrier synchronisation overhead.

Threads	With Barrier (ms)	Without Barrier (ms)	Overhead (ms)	Overhead %
2	3863.00	2693.00	1170.00	43.40
4	5610.20	1721.20	3889.00	226.00
8	11029.80	1479.80	9550.00	645.00

Table 4.2.2.5: Barrier Overhead Estimation.

The barrier overhead grows rapidly with thread count. At 8 threads, synchronisation consumes the majority of runtime, making parallel execution slower than sequential.

4.3 Conclusion

In conclusion, this demonstrated that parallel programming is an effective approach for solving the Laplace equation in scientific and engineering applications. The relaxation method, which updates each grid point using the average of its four neighbours, is well suited to parallelisation due to its regular structure and clear data dependencies. By implementing a parallel solver with barrier synchronisation, the workload was distributed across multiple threads while maintaining numerical correctness, with a CyclicBarrier ensuring that all threads completed each iteration before proceeding and preventing race conditions. Benchmarking results showed measurable performance improvements through parallel execution, while also highlighting the cost of synchronisation overhead and the need to balance parallelism with computational granularity. Overall, the exercise reinforced fundamental parallel programming concepts such as thread coordination, data partitioning, and synchronisation, with techniques that are broadly applicable across a wide range of scientific and engineering problems.

5. Lab 05 - Running MPJ Programs

5.1 Introduction

This report focuses on running MPJ (Message Passing Java) programs in a distributed computing environment. This laboratory session marks a significant transition from previous weeks, as we move beyond single-workstation parallel programming to explore distributed memory parallel computing across multiple networked computers. The primary objective of this lab was to gain hands-on experience with MPJ Express, a software framework that provides MPI-like (Message Passing Interface) functionality for Java applications. Unlike earlier labs where parallel programs executed within the multiple cores of a single workstation using shared memory, this week introduced the more challenging concept of distributed memory programming, where separate computers collaborate on solving parts of a single computational problem.

Throughout this session, we explored two operational modes of MPJ Express: multicore mode for initial testing on a single processor's cores, and cluster mode for true distributed computing across multiple nodes. A key practical component involved implementing a parallel version of the π (pi) calculation algorithm, which served as a benchmark to measure and analyse parallel speedup across different configurations. This report documents the setup process, implementation challenges, experimental results, and performance analysis of running MPJ programs in both multicore and cluster modes.

5.2 Approaches and Analysis

5.2.1 Methodology

Initially, a sequential Java program calculating π through numerical integration was implemented in without using the MPI functionality. The sequential method was executed to establish a baseline performance measurement. The computational workload was significant, with the number of integration steps (N) set to 1,000,000,000 to ensure the difference between is measurable.

After that, the parallel version of the function, MPJPi.java, was developed. The work decomposition strategy involved a block-wise distribution of the integration intervals among the available processes. A main host was responsible for aggregating partial sums from all other worker processes using MPI.COMM_WORLD.Send() and MPI.COMM_WORLD.Recv() functions for sending the requests to the nodes for calculation process and receiving the patrial sum and sum up at the main host.

```
double [] sendBuf = new double [] {sum} ;  // 1-element array containing sum
MPI.COMM_WORLD.Send(sendBuf, 0, 1, MPI.DOUBLE, 0, 0) ;
```

```
double [] recvBuf = new double [1] ;
for(int src = 1 ; src < P ; src++) {
  MPI.COMM_WORLD.Recv(recvBuf, 0, 1, MPI.DOUBLE, src, 0) ;
  sum += recvBuf [0] ;
}
```

The parallel version was tested in two primary configurations:

- **Multicore Mode**: The MPJ processes (MPI-like functions in Java) were executed as threads on the cores of a single node within the cluster.
- **Cluster Mode**: MPJ daemons were initiated on multiple nodes of the cluster. A machines file was created listing the hostnames of these nodes. The node would also have a machines file listing the main hostname. The mpjrun.sh command was then used with the -dev niodev -headnodeip mn01.soc MPJPi to launch the application across these distributed nodes.

Performance was evaluated by measuring the execution time for varying numbers of processes (P). Each configuration was run five times to account for system variability, and the average execution time was calculated. The parallel speedup was computed as the ratio of the sequential execution time to the parallel execution time for a given configuration.

$$parallel\ speedup = \frac{sequential\ time}{parallel\ time}$$

5.2.2 Result and Analysis

The performance for the parallel π calculator as well as the sequential benchmark calculation with N = 1,000,000,000 are systematically presented in the table below.

Nodes	1st (ms)	2nd (ms)	3rd (ms)	4th (ms)	5th (ms)	Average (ms)	Parallel Speedup	Efficiency (%)
Sequential Benchmark	6244	6214	6168	6239	6234	6216	1.00	100.00
2-Threads (Multicore)	3284	3287	3279	3274	3278	3280	1.90	95.00
2 Nodes (Cluster)	3510	3538	3488	3491	3574	3520	1.77	88.50
3 Nodes (Cluster)	2650	2565	2582	2586	2598	2596	2.39	80.00
4 Nodes (Cluster)	2193	1939	2125	2107	2047	2082	2.99	74.80
4-Threads with 2 Nodes (Multicore & Cluster)	2113	2155	2055	2067	2099	2098	2.96	74.00

Table 5.2.2.1: Benchmarking Results for Parallel π Calculation (N = 1,000,000,000)

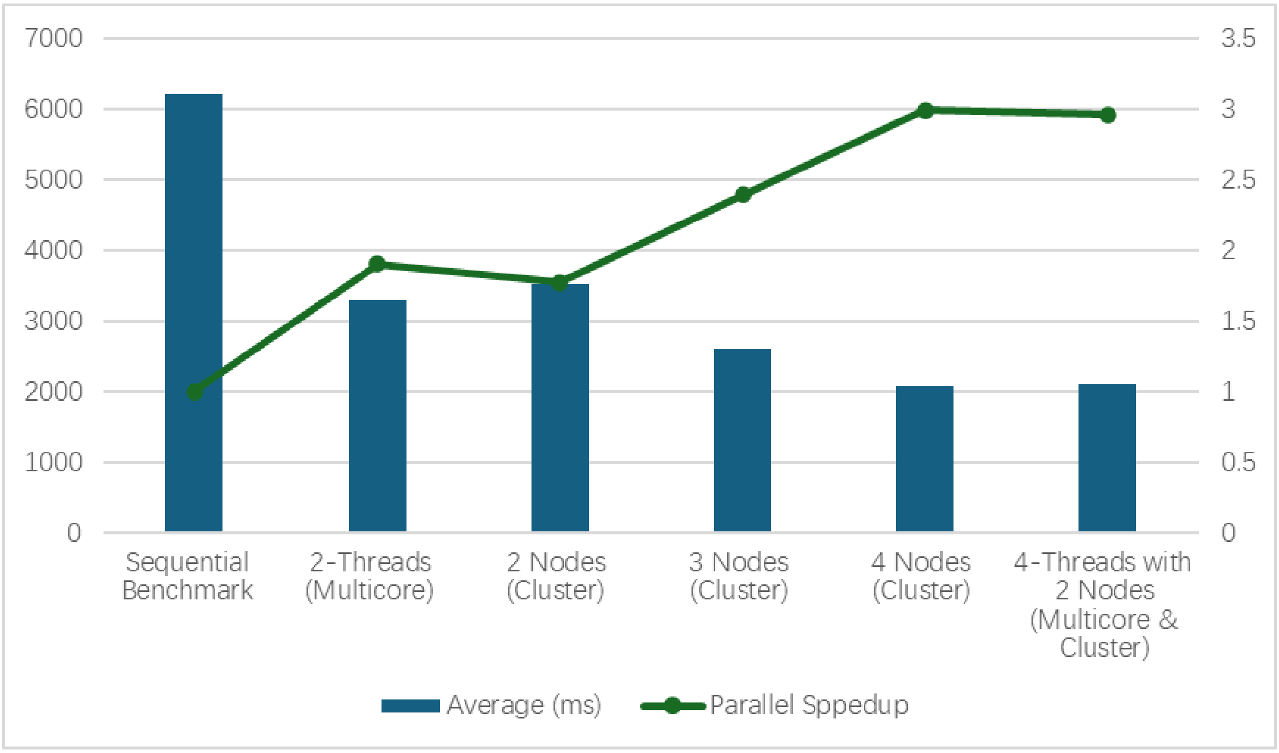


Figure5.2.2.1: Result of Parallel Programming with Cluster Mode and Multicore Mode with Sequential Benchmarking.

The trend of the parallel speedup is shown in the figure above. All parallel configurations achieved a reduction in execution time compared to the sequential baseline, confirming the effectiveness of the parallelisation strategy in both multicore mode and cluster mode. The speedup increased with the number of processes, demonstrating the scalability of the application for this problem size. The performance of 2 threads in multicore mode was very close to that of 2 processes distributed across 2 nodes in cluster mode (with parallel speedup of 1.90 and 1.77 respectively). This indicates that for compute-bound tasks like this, the overhead of inter-node communication via the network in cluster mode was not significantly detrimental compared to the lower-latency shared-memory communication in multicore mode at this scale.

The speedup improved notably when moving from 2 to 3 and then to 4 nodes, reaching a maximum speedup of approximately 3.0, although this is less than the ideal linear speedup of 4. The result for 4 threads distributed across 2 nodes also yielded a speedup of 2.96, showing that the efficiency is nearly as effective as using 4 nodes in the cluster mode.

5.3 Conclusion

This laboratory session successfully demonstrated the transition from shared-memory parallel programming to distributed-memory computing using MPJ Express. Through the implementation and benchmarking of a parallel π calculation algorithm, we gained practical experience with message-passing paradigms that form the foundation of large-scale distributed computing systems.

The experimental work revealed important insights into the performance characteristics of distributed computing. Operating MPJ Express in multicore mode provided an accessible testing environment that facilitated debugging and initial performance analysis on a single machine. The subsequent deployment in cluster mode across multiple networked computers illustrated both the potential and challenges of true distributed computing, where communication overhead between nodes becomes a critical factor affecting overall performance.

The π calculation benchmark effectively highlighted the trade-offs inherent in distributed programming. While parallelising the computational workload across multiple processes demonstrated measurable speedup, the performance gains were inevitably constrained by the communication costs associated with distributing work segments and aggregating partial results. The use of MPI.COMM_WORLD.Send() and MPI.COMM_WORLD.Recv() functions for inter-process communication emphasised the importance of minimising data transfer and optimising communication patterns in distributed applications.

6 Lab 06 - MPJ Communication

6.1 Introduction

This report investigates distributed memory parallel programming using MPJ Express through the implementation of a parallel Laplace Equation solver. Building on previous work with sequential and shared-memory computing, this lab explores the challenges of parallelising problems across a cluster where processes communicate via message passing.

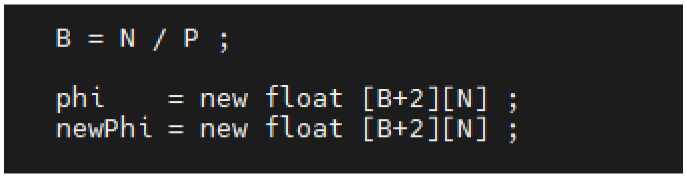
The primary objective was to examine how inter-process communication overhead, data decomposition strategies affect parallel efficiency. The implementation uses ghost regions and edge swap communications to maintain data consistency across distributed processes, following patterns like other distributed applications but with added complexity from fixed boundary conditions.

Experiments were conducted in both multicore mode and cluster mode to evaluate performance. Various optimisations were explored, including eliminating redundant boundary communications and investigating the relationship between problem size and scalability. The results provide insights into practical considerations for effective distributed memory parallel computing.

6.2 Methodology and Analysis

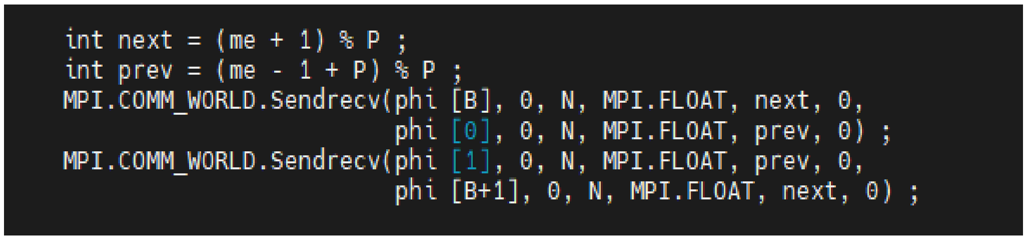
6.2.1 Methodology

The 2D Laplace domain of size N × N was divided among P processes along the i direction, so that each process handled a contiguous block of B = N/P rows. To allow stencil computations near block boundaries, two additional ghost rows were added to each block. These ghost rows stored boundary values exchanged with neighbouring processes, resulting in local arrays of size (B + 2) × N.

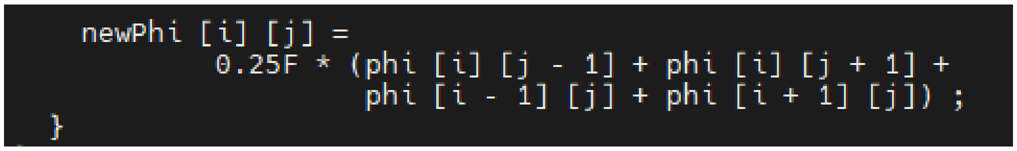


```
B = N / P ;

phi    = new float [B+2][N] ;
newPhi = new float [B+2][N] ;
```

During each iteration, nearby processes exchanged their boundary rows using MPI Sendrecv calls in order to update the ghost regions. Once this exchange was complete, interior grid points were updated using the standard five-point stencil. Fixed boundary conditions were maintained at the global edges, and for this reason process 0 and process P−1 restricted their update ranges to avoid altering boundary values.



```
int next = (me + 1) % P ;
int prev = (me - 1 + P) % P ;
MPI.COMM_WORLD.Sendrecv(phi [B], 0, N, MPI.FLOAT, next, 0,
                        phi [0], 0, N, MPI.FLOAT, prev, 0) ;
MPI.COMM_WORLD.Sendrecv(phi [1], 0, N, MPI.FLOAT, prev, 0,
                        phi [B+1], 0, N, MPI.FLOAT, next, 0) ;
```



```
  newPhi [i] [j] =
          0.25F * (phi [i] [j - 1] + phi [i] [j + 1] +
                   phi [i - 1] [j] + phi [i + 1] [j]) ;
}
```

For visualisation purposes, each process transmitted its computed block, excluding ghost rows, to process 0. The root process assembled these blocks into a single global array and displayed the solution using a Swing based graphical interface.

Execution time was measured using wall clock timing around the main iteration loop. To minimise the impact of input/output overhead on performance measurements, the frequency of graphical updates was reduced once after 1000 iterations during timing experiments.

6.2.2 Result and Analysis

The benchmarking results reveals a stark contrast between multicore and cluster mode performance, underscoring the dominance of communication costs in distributed environments.

Node	**1st (ms)**	**2nd (ms)**	**3rd (ms)**	**4th (ms)**	**5th (ms)**	**Ave (ms)**	**Parallel Speedup**	**Efficiency (%)**
1	9938	9186	9804	9619	9738	9656	1.00	100.00
2	8250	8365	7639	7963	8644	8172	1.18	59.00
3	7061	6331	7430	6842	7033	6939	1.39	46.30
4	7054	6966	7346	7789	7680	7367	1.31	32.80
8	14126	13947	14826	14362	13175	14087	0.69	8.60

Table6.2.2.1: Performance of MPJLaplace (N=256) in Multicore Mode

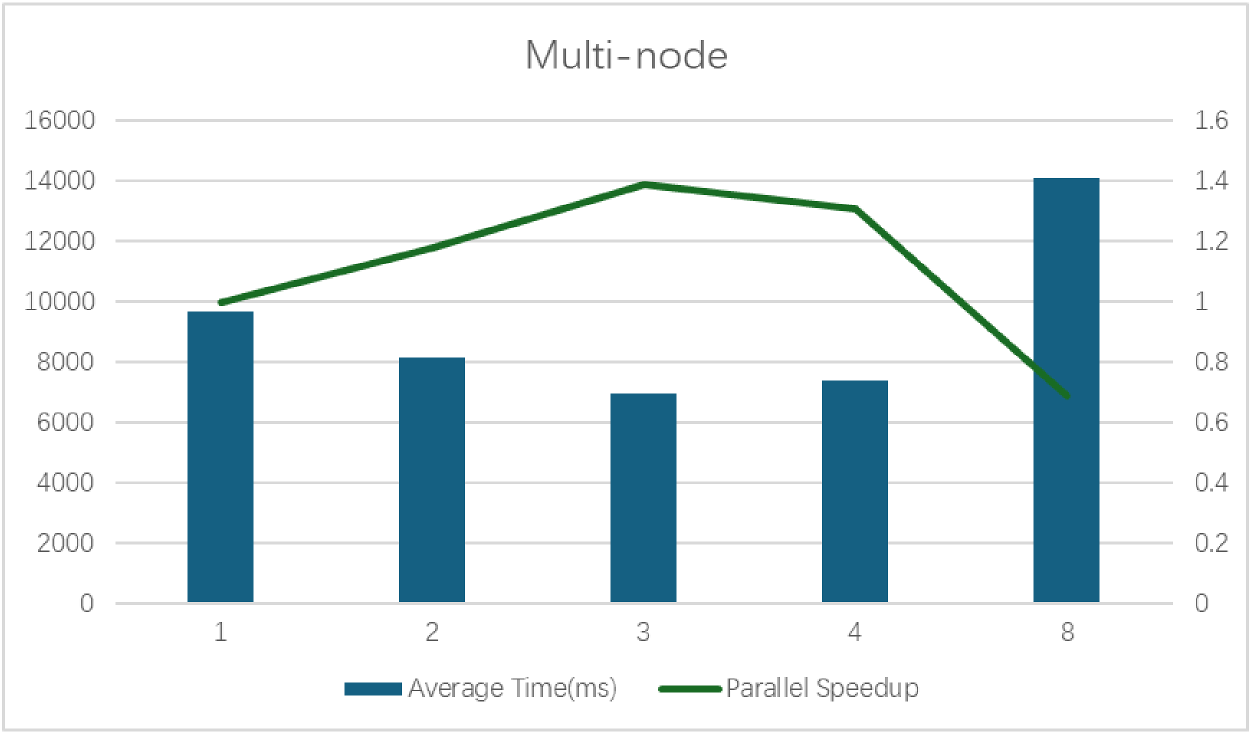


Figure6.2.2.1: Performance of MPJLaplace (N=256) in Multicore Mode and Trend of Parallel Speedup.

In multicore mode, linear speedups were partially achieved for P=2 and 3. Efficiency declined significantly as P increased, indicating growing overheads from thread management and memory contention. The severe slowdown at P=8 suggests resource oversubscription on the node's 12-core processor, leading to cache thrashing and scheduling contention.

Node	**1st (ms)**	**2nd (ms)**	**3rd (ms)**	**4th (ms)**	**5th (ms)**	**Ave (ms)**	**Parallel Speedup**	**Efficiency (%)**
1	9938	9186	9804	9619	9738	9656	1.00	100.00
2	44735	44825	44972	45037	44516	44817	0.22	11.00
3	43947	44024	43475	43869	43759	43815	0.22	7.30
4	41942	41812	41626	41482	41734	41719	0.23	5.80
8	38646	38398	38461	38426	38345	38455	0.25	3.10

Table6.2.2.2: Performance of MPJLaplace (N=256) in Cluster Mode.

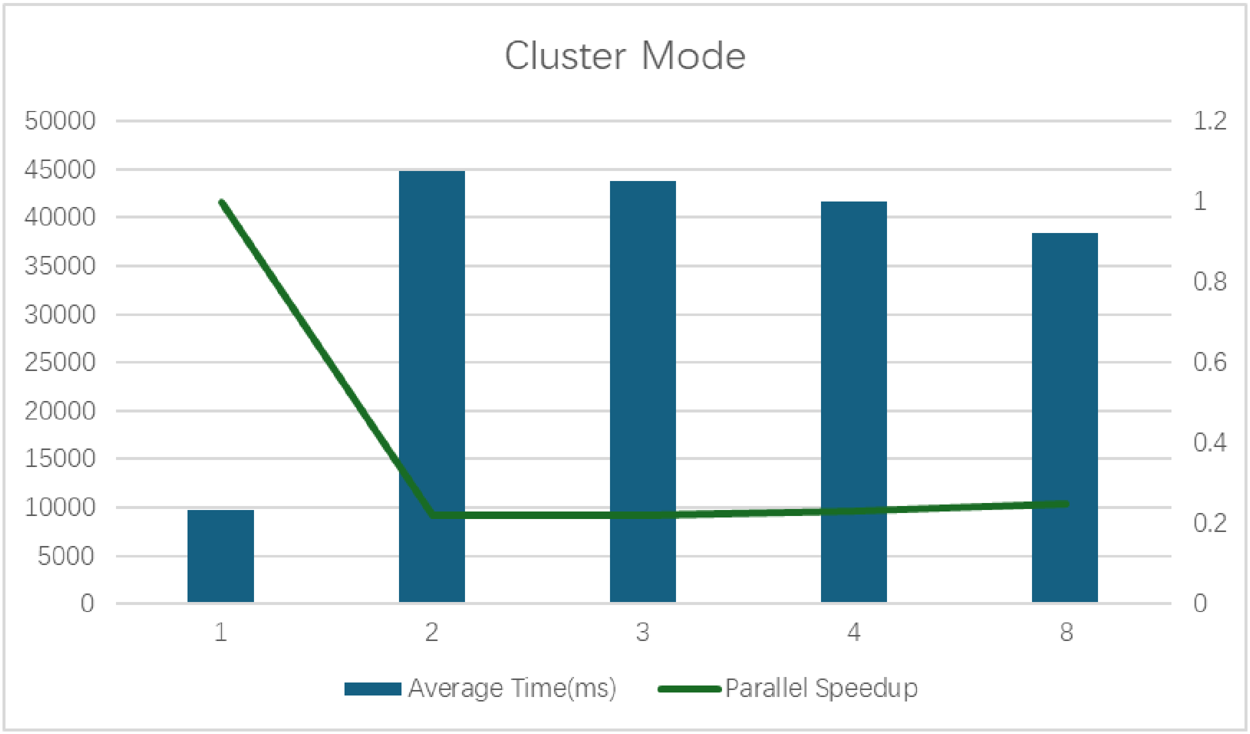


Figure6.2.2.2: Performance of MPJLaplace (N=256) in Cluster Mode and Trend in Parallel Speedup.

The cluster mode results are illuminating. A parallel slowdown was observed, with the 2-process configuration begin approximately 4.6 times slower than the sequential version. This is a direct result of high communication latency. For N=256, the local sub-grid per process is only 64x256 elements. The computation per iteration for this small domain is minimal, but the cost of performing two Sendecv operations (for the ghost regions) in every iteration across the network is exceedingly high. The total communication volume is substantial: for P=2 and NITER=100,000, there are 200,000 point-to-point messages. This results in a very low computation-to-computation ratio, causing communication costs to utterly dominate the total runtime.

Node	**1st (ms)**	**2nd (ms)**	**3rd (ms)**	**4th (ms)**	**5th (ms)**	**Ave (ms)**	**Parallel Speedup**	**Efficiency (%)**
1	8766	8729	8571	8597	8483	8629	1.00	100.00
2	4710	4944	5134	4663	4804	4851	1.78	89.00
3	3038	3094	3057	3169	3107	3093	2.79	93.00
4	2216	2192	2334	2237	2344	2264	3.81	95.30

Table6.2.2.3: Cluster Mode Performance with Communication Overhead Removed.

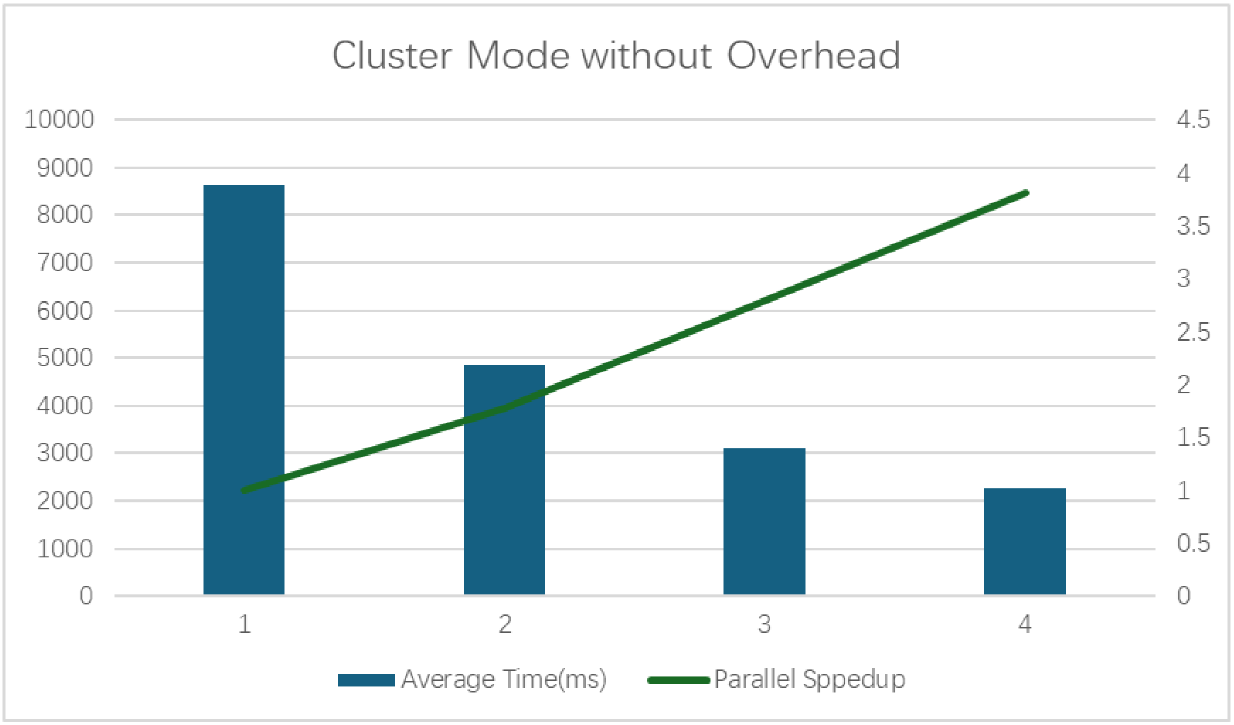


Figure6.2.2.3: Cluster Mode Performance with Communication Overhead Removed and Trend of Parallel Speedup

If we remove the edge-swap code validates this hypothesis, as shown in the Table 6.2.2.3, when communication is removed, a linear speedups and high efficiencies are achieved. This demonstrates that the data decomposition strategy is correctly implemented and that the computational load is well-balanced across nodes. The results in Table6.2.2.2 and Table6.2.2.3 show that the communication function is the main reason for the significant slowdown in cluster mode.

6.3 Conclusion

This lab discusses how static decomposition can lead to load imbalance when different parts of a problem require different amounts of computation and explains that dynamic scheduling is better suited to irregular workloads. The results from this lab directly contrast the limitations of static approaches by showing how a task farm (master–worker) model overcomes these issues in practice.

With small task sizes, the task farm dynamically redistributes work so that faster workers receive more tasks, significantly reducing idle time. This leads to strong scaling and consistently high efficiency as the number of processes increases. These results highlight a clear advantage over the static partitioning strategies outlined in this lab, where uneven workloads would cause processors to wait for the slowest task to finish.

However, the lab also demonstrates that dynamic scheduling is not automatically optimal. When task granularity is too large, the benefits of dynamic load balancing are reduced, and performance degrades due to imbalance and overhead with the communication function. One valuable lesson in this lab is that effective parallel performance depends on choosing an appropriate decomposition strategy and task size, particularly for problems with non-uniform computational cost.

7 Lab 07 - An MPJ Task Farm

7.1 Introduction

This lab focuses on the task farm model in parallel programming using MPJ Express. In earlier labs, parallelism was achieved by dividing a problem into fixed regions that were assigned to processes at the start of execution. While effective for some applications, this static approach can lead to load imbalance when different parts of the problem require different amounts of computation.

The task farm model addresses this issue by distributing work dynamically. A central master process assigns small units of work to worker processes as they become available. Once a worker finishes its task, it returns the result and immediately requests more work. This approach is particularly appropriate for embarrassingly parallel problems.

To demonstrate this model, the Mandelbrot set computation is used. Although the problem is inherently parallel, the workload is artificially increased by using a very large iteration cut-off value, making it suitable for evaluating parallel performance on multicore systems and distributed clusters.

7.2 Approaches and Analysis

7.2.1 Methodology

The experimental methodology was based on the implementation and execution of a master-worker task farm from the Mandelbrot set calculation. In this model, process 0 acts as the master, responsible for distributing tasks and collating results, while other processes act as workers. Therefore, for np processes, there are np-1 workers.

In this model, we will be using a few advanced features from MPI to help the implementation of the Mandelbrot set calculation. TAG_HELLO, TAG_TASK, TAG_RESULT, TAG_GOODBYE were used to differentiate between message types, enabling a single communication channel to handle multiple purposes. The master process used MPI.ANY_SOURCE and MPI.ANY_TAG in its Recv call to accept the next available message from any worker, whether a request for work or a completed result. The Status object returned by Recv was interrogated to determine the actual source (status.source) and message tag (status.tag), allowing the master to respond appropriately.

This lab will also test two different decompose strategies:

- **Vertical Strips**: This will be tested first, where each task comprised a contiguous block of BLOCK_SIZE rows of the image.
- **Square Blocks**: An alternative strategy where tasks were defined as square sub-regions of the image with block_edge = 64.

Benchmarking was conducted. The problem size was fixed at N=1024 and the CUTOFF=100,000. Each configuration was run five times to get an average execution time. Parallel speedup (Sp) and efficiency were calculated. The master process will also be calculated in the efficiency as part of the overall system overhead.

7.2.2 Results and Analysis

The benchmarking results provide a comprehensive view of the task farm's performance characteristics, revealing strengths and a critical performance pathology.

Nodes	**1st (ms)**	**2nd (ms)**	**3rd (ms)**	**4th (ms)**	**5th (ms)**	**Ave (ms)**	**Parallel Speedup**	**Efficiency (%)**
2	27881	28120	28921	28121	28581	28325	1.00	50.00
3	14353	14374	14531	14461	14407	14425	1.96	65.30
4	9925	9716	9910	9696	9800	9809	2.89	72.30
5	7502	7368	7369	7364	7386	7398	3.83	76.60
6	6065	6314	6069	6053	5982	6097	4.65	77.50
7	5118	5155	5040	5123	5043	5096	5.56	79.40
8	4425	4477	4437	4394	4408	4428	6.40	80.00
9	3869	3937	3919	4018	3967	3942	7.19	79.90
10	3660	3675	3611	3559	3557	3612	7.84	78.40
11	3453	3372	3234	3406	3247	3342	8.47	77.00
12	3076	3164	2952	3015	3150	3071	9.22	76.80

Table7.2.2.1: Multicore Performance Results for Mandelbrot Set Calculation (N=1024, CUTOFF=100,000, Block Size=4)

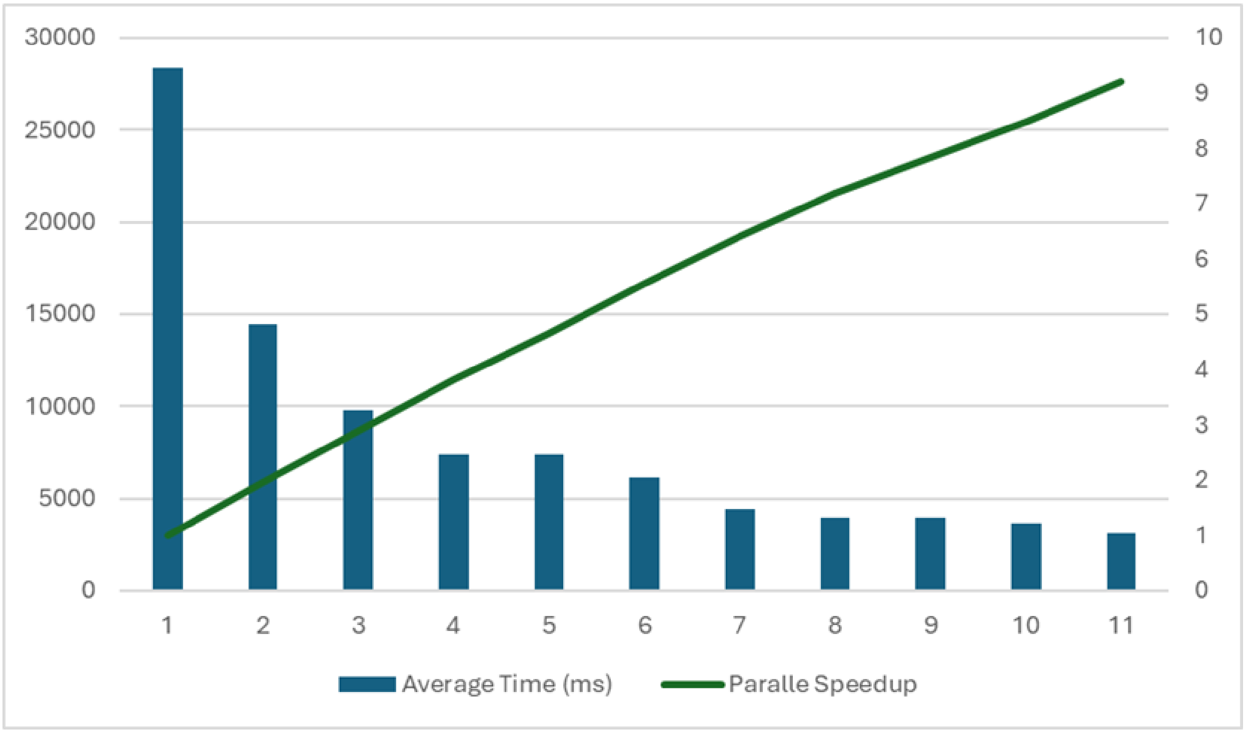


Figure7.2.2.1: Multicore Trend for Mandelbrot Set Calculation (N=1024, CUTOFF=100,000, Block Size=4)

The multicore results show strong scaling as workers(np) increase. Average runtime drops from 28325ms (np=2) down to 3071ms (np=12), with speedup rising to 9.22 and efficiency staying roughly at 77% to 80% across larger process counts.

The task farm model is doing its job, dynamic scheduling keeps workers busy, so efficiency remains relatively high even as P grows. Speedup becomes less-than-ideal at higher P due to unavoidable overheads (master coordination, communication, contention), but the overall trend is still strongly positive.

Nodes	**1st (ms)**	**2nd (ms)**	**3rd (ms)**	**4th (ms)**	**5th (ms)**	**Ave (ms)**	**Parallel Speedup**	**Efficiency (%)**
2	26721	26732	26692	26676	26646	26693	1.00	50.00
3	13645	13693	13739	13955	13660	13738	1.94	64.70
4	9293	9322	9302	9660	9293	9374	2.85	71.30
5	7172	7245	7299	7144	7147	7201	3.71	74.20
6	5802	5840	5882	5857	5814	5832	4.58	76.30

Table 7.2.2.2: Cluster Performance Results for Mandelbrot Set Calculation (N=1024, CUTOFF=100,000, Block Size=4)

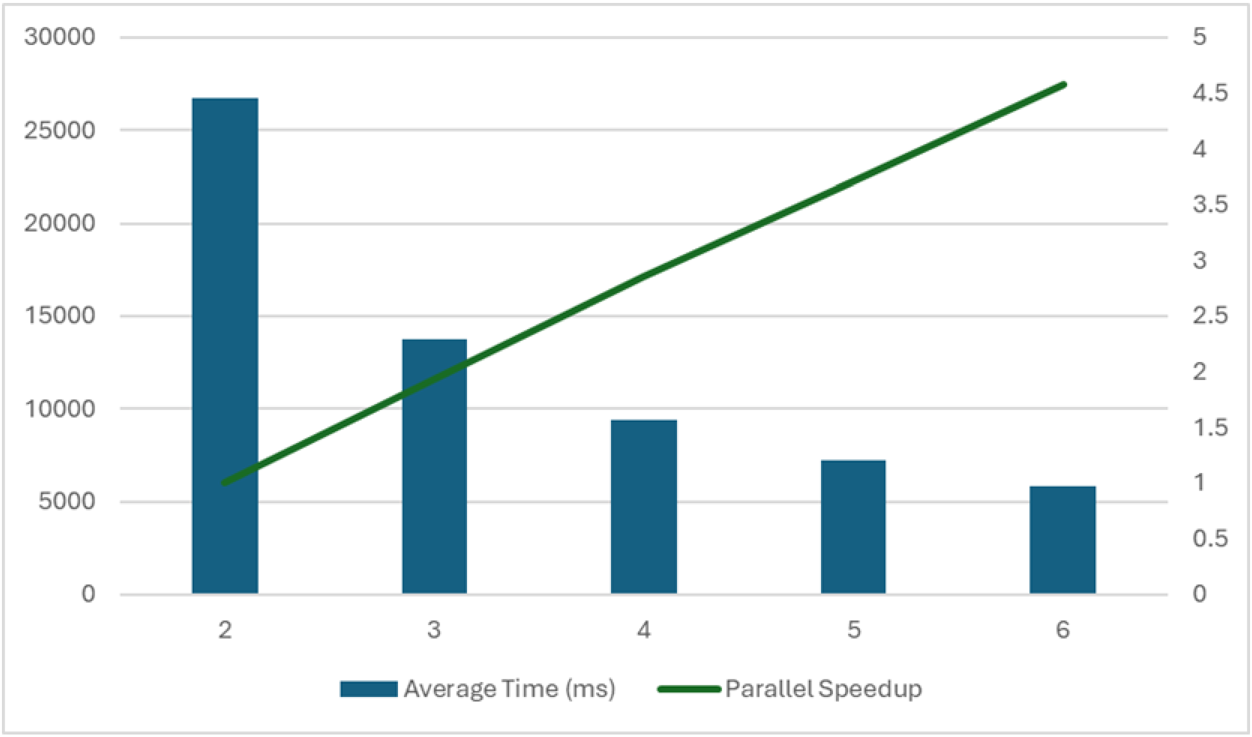


Figure7.2.2.2: Cluster Trend Performance Results for Mandelbrot Set Calculation (N=1024, CUTOFF=100,000, Block Size=4).

The vertical strip farm performed effectively in cluster mode, as shown in Table 2. While efficiencies were slightly lower than in multicore mode indicating the added latency of network communication, they remained above 71% for up to 6 total processes. This confirms that for problems with a very high computation-to-communication ratio, the task farm paradigm can leverage distributed resources efficiently, as the time spent computing dwarfs the time spent communicating tasks and results.

Nodes	**1st (ms)**	**2nd (ms)**	**3rd (ms)**	**4th (ms)**	**5th (ms)**	**Ave (ms)**	**Parallel Speedup**	**Efficiency (%)**
2	26721	26732	26692	26676	26646	26693	1.00	50.00
3	14398	14178	14339	14102	14171	14238	1.87	62.30
4	9357	9497	9487	9510	9537	9478	2.82	70.50
5	7345	7635	7213	7415	7481	7424	3.60	72.00
6	6045	6012	5998	5943	5968	5993	4.45	74.20

Table7.2.2.3: Multicore-Cluster Hybrid Performance Results for Mandelbrot Set Calculation (N=1024, CUTOFF=100,000, Block Size=4)

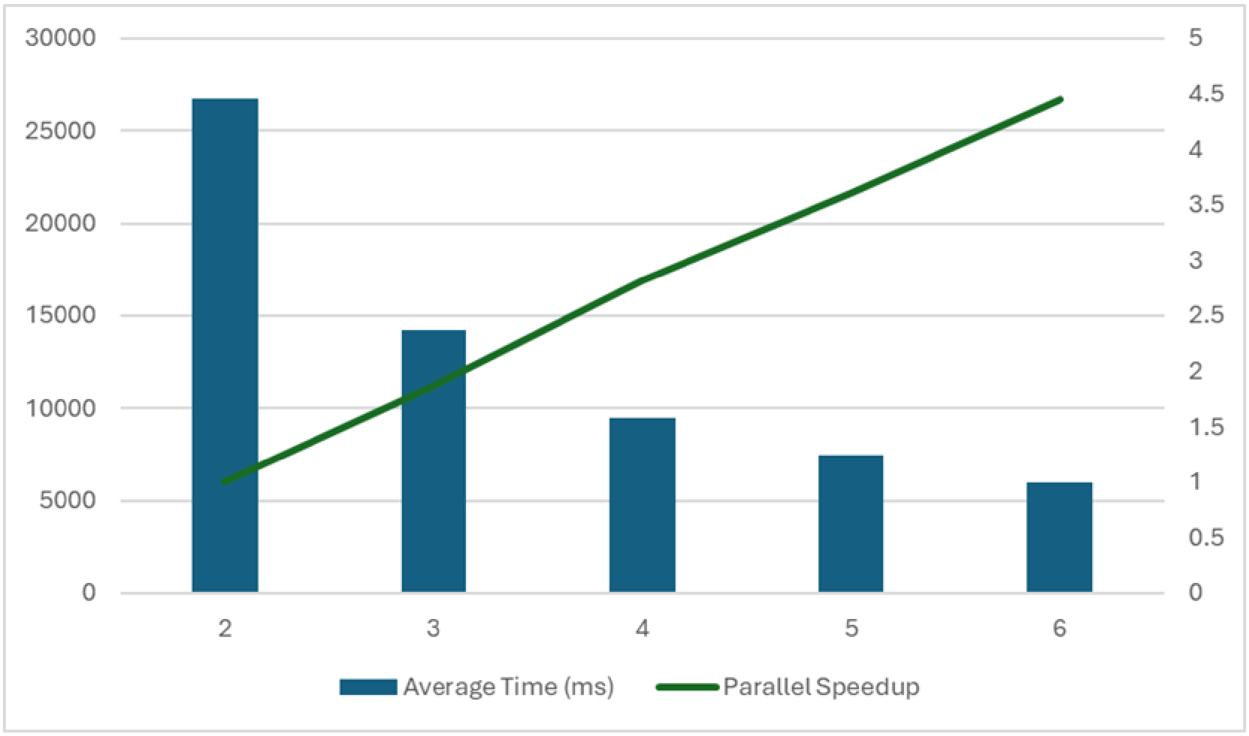


Figure7.2.2.3: Multicore-Cluster Hybrid Trend Performance Results for Mandelbrot Set Calculation (N=1024, CUTOFF=100,000, Block Size=4).

For the same block size and comparable process counts, cluster mode (np=2-6) and the multicore-cluster setup achieve similar scaling trends, Cluster mode reaches speedup 4.58 at np=6 (avg 5832ms), while Multicore-cluster reaches speedup 4.45 at np=6 (avg 5993ms). With sufficiently compute-heavy tasks (high CUTOFF) and small blocks, the farm can scale well even when workers are distributed, because computation per task helps amortise message overhead.

Nodes	**1st (ms)**	**2nd (ms)**	**3rd (ms)**	**4th (ms)**	**5th (ms)**	**Ave (ms)**	**Parallel Speedup**	**Efficiency (%)**
2	34161	32682	32054	32624	32512	32807	1.00	50.00
3	17056	16970	16859	17148	17159	17038	1.93	64.30
4	11798	12349	12215	11935	11703	12000	2.73	68.30
5	9483	9343	9269	9186	9367	9330	3.52	70.40
6	7655	7565	7806	7758	8021	7761	4.23	70.50

Table7.2.2.4: Multicore Performance Results for Mandelbrot Set Calculation (N=1024, CUTOFF=100,000, Block Size=64)

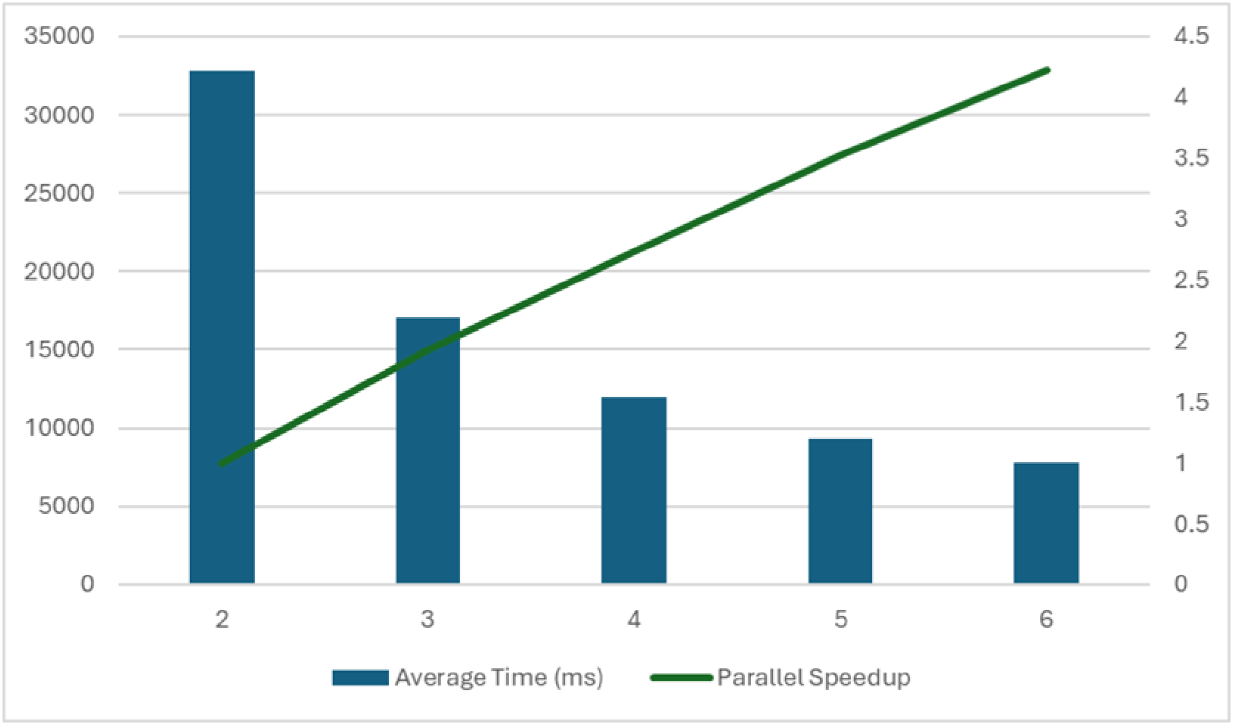


Figure7.2.2.4: Multicore Trend Performance Results for Mandelbrot Set Calculation (N=1024, CUTOFF=100,000, Block Size=64).

Nodes	**1st (ms)**	**2nd (ms)**	**3rd (ms)**	**4th (ms)**	**5th (ms)**	**Ave (ms)**	**Parallel Speedup**	**Efficiency (%)**
2	71283	71460	72024	71409	72095	71654	1.00	50.00
3	52680	52778	52821	52930	52473	52736	1.36	45.30
4	46652	46874	46919	46942	46813	46840	1.53	38.20
5	43958	44080	44102	44010	44081	44046	1.63	32.50
6	42352	42305	42118	42092	42329	42239	1.70	28.30

Table7.2.2.5: Cluster Performance Results for Mandelbrot Set Calculation (N=1024, CUTOFF=100,000, Block Size=64)

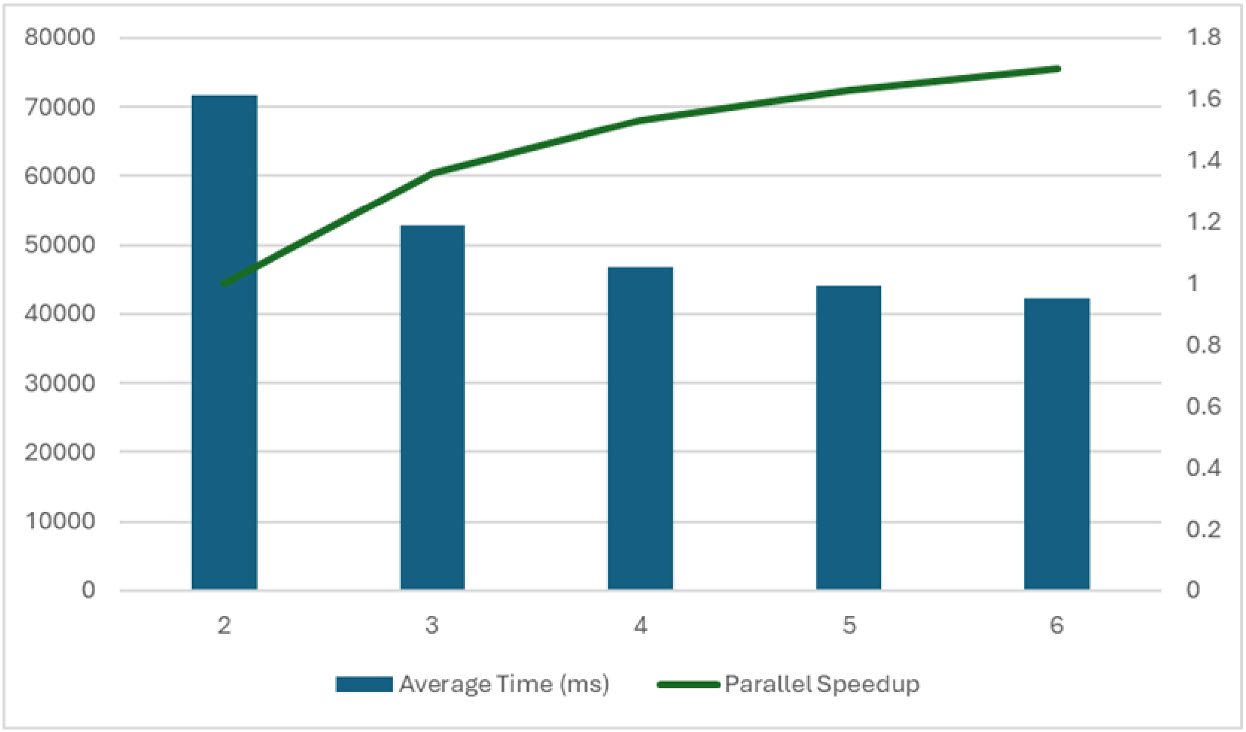


Figure7.2.2.5: Cluster Performance Results for Mandelbrot Set Calculation (N=1024, CUTOFF=100,000, Block Size=64)

Changing block size had a noticeable impact, Multicore block size = 64 is consistently slower than block size = 4 at the same np (e.g., at np=6: 7761ms vs 6097ms).

This suggests that larger blocks reduce the scheduler's ability to "smooth out" uneven work across the fractal (i.e., less effective load balancing), which is exactly what task farming is meant to solve. Cluster block size = 64 performs dramatically worse (e.g., np=2 average 71,654ms, and only 1.70 speedup at np=6).

This indicates large blocks were not suitable for the cluster setup in this dataset.

7.3 Conclusion

The task farm model helps reduce load imbalance by distributing work dynamically. It is highly effective, achieved excellent speedup, as seen in the Mandelbrot set computation. On multicore systems, the results showed good scaling behaviour, with efficiency staying around 76% to 80% even when the number of processes increased. In contrast, performance on a cluster was more dependent on the chosen block size and decompose strategies. Smaller blocks, such as size four under vertical strips, achieved better efficiency by allowing computation and communication to overlap, whereas larger blocks, such as size 64 under square blocks, caused a noticeable drop in performance, particularly in distributed settings. Overall, these findings indicate that the task farm approach is effective for embarrassingly parallel problems with a high ratio of computation to communication, although careful adjustment of task size is required to achieve good performance on different hardware platforms.

8 Lab 09 - Aparapi

8.1 Introduction

This lab presents a comprehensive evaluation of parallel computing for calculations of `matrix`multiplications, comparing the performance characteristics of CPU-based and GPU-based approaches. The experiments systematically investigate how computational efficiency scales with problem size and parallelisation strategies. Through controlled benchmarking using single-threaded CPU execution, optimised multi-threaded CPU implementations, and GPU-accelerated processing via the Aparapi framework, this study quantifies the performance between different parallel computing architectures. The analysis focuses on execution time measurements and computational throughput (GFLOPS) across varying grid dimensions (N = 1024, 2048, and 4096) to determine the optimal computational approach for large-scale calculations.

8.2 Approaches and Analysis

8.2.1 Methodology

A standard sequential matrix multiplication algorithm was implemented in Java using three nested loops. Two matrices, a and b, were initialised with deterministic values and multiplied to produce matrix c. Execution time was measured using system clock values, and performance was calculated using:

$$\text{GFLOPS} = \frac{2 \times N^3}{10^6 \times time(ms)}$$

The experiment was first conducted with N = 1024 and then repeated with larger matrix sizes to observe performance behaviour.

To experiment the effect of cache utilisation, matrix b was transposed into a temporary array bt. The modified algorithm was executed with larger matrix sizes (e.g. N=2048) and performance results were recorded for comparison with the naive sequential approach.

GPU acceleration was experimented using Aparapi. A Maven Java project was created and the Aparapi dependency was added. The matrix multiplication was expressed as a kernel where each work item computed one element of the output matrix. The kernel was executed over a two-dimensional range, and Aparapi automatically selected the most suitable available device. Execution time and performance were recorded for larger matrix sizes.

8.2.2 Results and Analysis

The performance results for all three implementations are presented in the following tables. The performance of the sequential CPU implementation serves as the benchmark for the experiment. The results in Table8.2.2.1 reveals a significant performance degradation as the problem size increases. The data of the performance results for all three implementations are presented in the following tables. The performance of the sequential CPU implementation serves as the benchmark for the experiment. The results in Table8.2.2.1 reveals a significant performance degradation as the problem size increases. The data shows that there is a considerable variable in execution time for N=1024, with the first run being slowest (4859 ms). This is very likely due to just-in-time compilation overhead in the Java runtime environment. As size increases to N=2048, the performance dropped significantly, to only one third of GFLOPS than only half of the size ones. This confirms that there is a performance drop as the problem size increases. One of the reasons is the overused CPU's memory as the problem size increases. The storage size for N=2048 is approximately 4 times more than storage size for N=1024 for the calculation of the three matrices with the storage size at O(N^3). This would overwhelm the CPU's memory, could result in a high-rate cache missed and forced CPU to read and use the slower main memory.

Matrix Size (N)	1st (ms)	2nd (ms)	3rd (ms)	4th (ms)	5th (ms)	Average (ms)	GFLOPS
1024	4859	2629	2995	2735	2584	3160	0.68
2048	76268	60611	62177	62819	62845	64944	0.26

Table8.2.2.1: Performance of the Sequential CPU (Baseline) Implementation.

The optimised version, which pre-computes the transpose of matrix B to enable contiguous memory accesses, demonstrates a different trend as shown in Table8.2.2.2. The results shown the consistency and efficiency. This implementation maintains a nearly constant 1.97 GFLOPS for both problem sizes N=1024 and N=2048. The optimisation successfully neutralises the memory bottleneck. The improvement for N=2024 is 7.4 times faster than the benchmark version.

Matrix Size (N)	1st (ms)	2nd (ms)	3rd (ms)	4th (ms)	5th (ms)	Average (ms)	GFLOPS
1024	1088	1110	1100	1088	1074	1092	1.97
2048	8955	8672	8653	8735	8676	8738	1.97

Table8.2.2.2: Performance of the CPU Implementation with Modified Algorithm.

The implementation of GPU execution with the Aparapi-based approach did not go as planned. Firstly, the experiment output was significantly lower than the optimised algorithm on the CPU code. The device type, ALT, indicates that the framework did not execute the kernel on the GPU. Instead, it fell back to a secondary mode, which is typically a Java thread pool running on the CPU cores. Therefore, this performance data does not represent GPU capability but rather the performance of a multi-threaded CPU execution. The results are displayed in Table 8.2.2.3. Although this failure does not relate to the GPU, it revealed that the performance of the multi-threaded CPU's performance. The Aparapi fallback mechanism incurs overhead from managing Java threads, and the kernel abstraction could also incur some extra time during the experiments. Although the GFLOPS values are smaller compared to the modified algorithm's version at both N=1024 and N=2048, the performance of N=1024 with GFLOPS at 1.14, and N=2048 with GFLOPS at 0.78 shows that multi-threaded CPU performance still outperforms the single-threaded CPU performance with GFLOPS values of 0.68 and 0.26, respectively, in the benchmark.

Matrix Size (N)	**1st (ms)**	**2nd (ms)**	**3rd (ms)**	**4th (ms)**	**5th (ms)**	**Average (ms)**	**GFLOPS**
1024	1969	1892	1673	1948	1919	1880	1.14
2048	23412	21960	19945	22489	21837	21929	0.78
4096	303474	286133	291677	294683	277978	290789	0.47

Table8.2.2.3: Performance of the Multi-Threaded CPU Implementation.

After the correction to the environment, the Apirapi kernel successfully executed on the GPU, as confirmed by the terminal output: Device type = GPU. The performance results for this successful GPU execution are summarised in Table8.2.2.4. The data shows the definitive performance advantage of the GPU for this type of parallel task, especially when the problem scale is sufficiently large. For N=1024, THE GPU's performance is only faster than the single-threaded benchmark, and the performance of the modified algorithm is slightly faster than GPU implementations. This could be due to the fixed overhead of launching a GPU kernel and transferring data.

However, for N=2048 and N=4096, the GPU demonstrates its capability in parallel programming. The GFLOPS jumped to 9.22 and 40.89, respectively. This is about 4 times and 21 times faster than the modified algorithm implementation for the same problem sizes. This is mainly due to the kernel launches N2 concurrent threads (over 16 million for N=4096), which are scheduled across hundreds of processing elements within the GPU's streaming Multiprocessors (SMs). The ability of GPU GDDR memory in high memory bandwidth and rapidly switching between thousands of active threads also contributes to the GPU's higher computational throughput than a CPU for compute-intensive parallel programming operations.

Matrix Size (N)	**1st (ms)**	**2nd (ms)**	**3rd (ms)**	**4th (ms)**	**5th (ms)**	**Average (ms)**	**GFLOPS**
1024	1651	1579	1741	1775	1723	1694	1.27
2048	1746	1812	1715	2141	1899	1863	9.22
4096	3430	3368	3208	3340	3461	3361	40.89

Table8.2.2.4: Performance of the GPU Implementation.

8.3 Conclusion

The experimental results demonstrate that GPU acceleration provides substantial performance advantages for large-scale parallel computations in matrix multiplication. While single-threaded CPU execution established baseline performance, the optimised multi-threaded CPU implementation showed moderate improvements, achieving up to 0.78 GFLOPS for N=2048. However, the GPU implementation dramatically outperformed both CPU approaches for larger problem sizes, delivering 40.89 GFLOPS for N=4096, approximately 21 times faster than the modified algorithm version's implementation. This performance stems from the GPU's ability to execute thousands of concurrent threads across its streaming multiprocessors and leverage high-bandwidth GDDR memory. The findings confirm that for compute-intensive parallel operations with sufficient problem scale, GPU computing offers a compelling solution that significantly reduces execution time and maximises computational throughput, making it the preferred architecture for large-scale calculations.

9 Development Project - Parallelisation in Matrices Calculation

9.1 Introduction

Matrix computations form the core of many scientific, engineering, and data-intensive applications, including simulations, optimisation problems, and machine learning. As matrix sizes increase, sequential execution becomes computationally expensive, motivating the use of parallel computing techniques to reduce execution time. This project investigates the performance impact of parallelisation on matrix-based algorithms. The primary focus is on matrix multiplication as well as LU decomposition. The implementations include sequential algorithms and parallel versions of multicore mode and cluster mode using MPJ Express and Aparapi. Performance is evaluated by comparing execution times, speedup, and scalability across different matrix sizes and GFLOPS as mentioned in Chapter8.

9.2 Methodologies

The experiments are designed with the increasing sequence of matrix sizes of 512, 1024, 2048, 3072, 4096. This selection was due to several technical issues. One of the main reasons is the GPU architectural constraints. The GPU is designed so that to achieve maximum performance, the problem size needs to be aligned with the warp size (usually 16, or 32 threads). Hence, the problem sizes are set to be multiples of 32.

Also, with the computational complexity of $O(n^3)$ for both matrix multiplication and LU decomposition, and in the sequential experiment, we discovered that with the size of 5,000, it will take about 30 minutes to finish one multiplication between two matrices, and with the size of 8,192 will take nearly 350 minutes to do the calculations. Hence, the size is kept below 5,000 and need to be multiples of 32.

We selected matrix multiplication and LU decomposition as target operations due to some of their natures. Matrix Multiplication exhibits inherent parallelism with minimal data dependencies. LU decomposition contains inherent sequential dependencies in factorisation steps. Both operations have the computational complexity of $O(n^3)$, that can demonstrate the best of the parallelism with a perfect benchmark for evaluation rather than other operations that only has $O(n)$. Matrix multiplication is a canonical example of a compute-intensive problem that exhibits high data parallelism, making it well suited to shared-memory parallelisation when memory access patterns are carefully managed (Grama et al., 2003).

The project tasks are distributed into the following parts: the sequential implementation that work out as the benchmark for our parallel method evaluation, multicore parallelisation that using Java and shared memory locally, MPJ cluster mode implementation that using distributed memory, and the Aparapi API that uses GPU. Each configuration was executed with five independent trails and then obtained the average value to ensure reliable and stable data.

9.2.1 Sequential Implementation

The sequential implementations served as performance baselines and validation references for the parallel programming parts. Both algorithms employed conventional mathematical formulations optimised for clarity and numerical stability.

In matrix multiplication, the sequential method implemented the standard triple nested loop formulation. The computational complexity of $O(n^3)$ with $2n^3$ floating point operations.

For LU decomposition, the core algorithm implements a standard Gaussian elimination process to factorise the matrix A in to lower (L) and upper (U) triangular matrices such that:

$$A = L \times U$$

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}$$

The sequential process iterates through each column k (from 0 to N – 1). For each step, k, the element at A_{kk} is selected as the pivot. The algorithm computes the column of L below the diagonal by dividing elements A_{ik} by pivot A_{kk} (for i > k).

The algorithm also ignored the pivoting method in this implementation for simplicity. The code will check the absolute value of current pivot A_{kk} is approximately zero or not. If so, the program terminates the row rather than attempting to permute rows to find a viable pivot.

The linear system Ax = b (or LUx = b) is solved using a two-stage substitution process. The algorithm first solves the lower triangular system Ly=b for the intermediate vector y. Since L is implicit with a unit diagonal, the update formula can be simplified to the following equation:

$$y_i = b_i - \sum_{j=0}^{i-1} L_{ij} y_j$$

This process is performed sequentially from the first row to the last. Then, the algorithm solves the upper triangular system Ux = y to determine the final solution vector x. This is performed in reverse order from (N-1 down to 0). The diagonal element of U serves as the divisors in this step:

$$x_i = \frac{1}{U_{ii}} \left(y_i - \sum_{j=i+1}^{N-1} U_{ij} x_j \right)$$

The code will check here to ensure no diagonal element in U is vanishingly small before division.

9.2.2 Multicore

Two square matrices A and B of size N*N were generated using pseudo-random double-precision values. A fixed random seed was used to ensure reproducibility across runs. To improve cache performance, matrix B was transposed once before multiplication. This allows the innermost loop to access memory contiguously, reducing cache misses and improving execution time.

The Parallel implementation for Matrix multiplication was performed using Java threads and a row-wise decomposition strategy, furthermore

- The result matrix C is divided into contiguous blocks of rows.
- Each thread computes a distinct subset of rows.
- Threads write to independent rows of the output matrix, eliminating race conditions.
- The main thread waits for all worker threads using join().

The number of threads was varied between 2 and 6. Three warm-up runs were performed to allow the Java JIT compiler to optimise execution, five timed runs were recorded, the average execution time was calculated.

Parallelisation of LU decomposition is significantly more challenging than matrix multiplication due to strong data dependencies between successive elimination steps. At iteration k, the updates for iteration k+1 cannot begin until all updates for iteration k are complete. A row-wise decomposition strategy was employed for the update of the trailing submatrix. At each step k, rows k+1 to N-1 of the trailing submatrix were divided into contiguous blocks, and each block was assigned to a separate thread.

Initially, a naive thread-based implementation was attempted, in which threads were created and joined at each iteration. This approach resulted in significant overhead and poor performance. To reduce this overhead, a thread pool (ExecutorService) was introduced. The thread pool creates a fixed number of worker threads once and reuses them throughout the entire LU factorisation. At each iteration k, tasks corresponding to row blocks are submitted to the pool, and a synchronisation barrier is enforced before proceeding to the next iteration.

9.2.3 MPJ

Both algorithms were implemented in Java and executed on a distributed memory cluster using multiple processes. The aim was to analyse how execution time and speedup change with increasing matrix size and number of processes.

For matrix multiplication, rows of matrix A were distributed evenly across all processes using MPI.Scatter, while matrix B was broadcast to every process using MPI.Bcast. Each process computed its assigned block of the result matrix independently, after which the partial results were gathered back to the root process using MPI.Gather. This approach minimises communication during computation and allows most operations to proceed in parallel.

For LU decomposition, the matrix was also distributed row-wise across processes. The algorithm followed a simplified LU factorisation without pivoting. At each iteration, the process owning the pivot row extracted the row and broadcast it to all other processes. Each process then updated its local rows before a global barrier synchronisation ensured that all processes completed the iteration before proceeding. Execution time was measured only for the decomposition phase, excluding matrix initialisation.

9.2.4 Using Aparapi

The Aparapi allows developers to write native Java code capable of being executed directly on a GPU by converting Java byte code to an OpenCL kernel dynamically at runtime (Syncleus, 2021). Matrix operations were parallelised using a two-dimensional decomposition strategy, where each GPU thread computed a single element of the matrix.

For matrix multiplication, the work group configuration uses 16 x 16 threads per work group as set by default, aligning with typical GPU warp sizes with n × n matrices as global variable.

For LU decomposition, a hybrid approach was necessitated by the algorithm's nature properties. A sequential coordination in the outer loop iterations managed by CPU with GPU executing parallelisable inner loops.

With the modern GPU's power, the primary constraint for matrix operation typically resides in memory bandwidth rather than GPU's computational capacity with the working sizes we have.

9.3 Result and Analysis

9.3.1 Multicore

Sequential

N	1st (ms)	2nd (ms)	3rd (ms)	4th (ms)	5th (ms)	Average (ms)
512	192	176	170	170	166	175
1024	1852	1938	1855	1859	1816	1864
2048	11743	11615	11577	11775	13129	11968
3072	118769	126249	126737	126368	125384	124702
4096	290978	298139	294487	273736	280276	287523

Table9.3.1.1: Sequential performance for Matrix Multiplication

N=512

Nodes	1st (ms)	2nd (ms)	3rd (ms)	4th (ms)	5th (ms)	Ave (ms)	Parallel Speedup
2	109.48	116.29	113.55	114.75	116.93	114.19	1.52
3	93.13	99.97	100.80	100.78	113.19	100.54	1.73
4	79.19	84.12	80.86	81.71	80.25	81.27	2.14
5	71.74	73.72	74.26	73.32	69.87	72.35	2.41
6	59.26	63.87	63.18	58.86	60.70	61.22	2.85

N=1024

Nodes	1st (ms)	2nd (ms)	3rd (ms)	4th (ms)	5th (ms)	Ave (ms)	Parallel Speedup
2	1051.55	1123.79	1053.12	1211.60	1224.83	1132.47	1.64
3	794.82	892.16	836.36	774.08	735.90	807.64	2.31
4	590.31	540.24	609.89	585.72	605.80	586.86	3.17
5	569.05	576.35	539.91	497.20	568.41	550.85	3.38
6	446.01	464.14	540.24	434.65	427.53	462.23	4.18

N=2048

Nodes	**1st (ms)**	**2nd (ms)**	**3rd (ms)**	**4th (ms)**	**5th (ms)**	**Ave (ms)**	**Parallel Speedup**
2	7666.02	7835.89	7692.36	7916.76	7711.60	7764.53	1.53
3	5603.60	5613.22	5887.81	5695.29	5659.08	5691.80	2.10
4	3729.17	3366.20	3533.43	3091.31	3384.90	3421.00	3.49
5	2271.67	2191.29	2115.86	2374.19	2285.86	2247.77	5.32
6	1988.73	2029.67	2046.30	1986.86	1994.73	2009.26	5.95

N=3072

Nodes	**1st (ms)**	**2nd (ms)**	**3rd (ms)**	**4th (ms)**	**5th (ms)**	**Ave (ms)**	**Parallel Speedup**
2	56123.64	61258.53	64089.70	58666.47	58565.69	59740.81	2.08
3	39088.11	35324.15	36645.19	35568.63	34305.58	36186.33	3.44
4	28612.21	29478.71	29848.28	30540.15	36680.72	31032.01	4.01
5	23958.83	23736.68	23079.72	24255.39	24255.39	22462.45	5.55
6	19970.48	19971.41	19966.25	19864.29	20901.26	20134.65	6.19

N=4096

Nodes	**1st (ms)**	**2nd (ms)**	**3rd (ms)**	**4th (ms)**	**5th (ms)**	**Ave (ms)**	**Parallel Speedup**
2	141764.35	143876.67	143854.74	143621.66	143557.85	143335.05	2.00
3	96567.23	96275.63	96378.33	96361.14	95631.92	96242.85	2.98
4	71739.79	71937.77	71074.89	71640.67	71739.09	71626.44	4.01
5	57907.67	56964.01	56899.35	57134.87	57468.25	57274.83	5.02
6	47987.79	47972.57	47983.96	47986.78	46563.68	47698.96	6.02

Table9.3.1.2: Parallel Performance for Matrix Multiplication

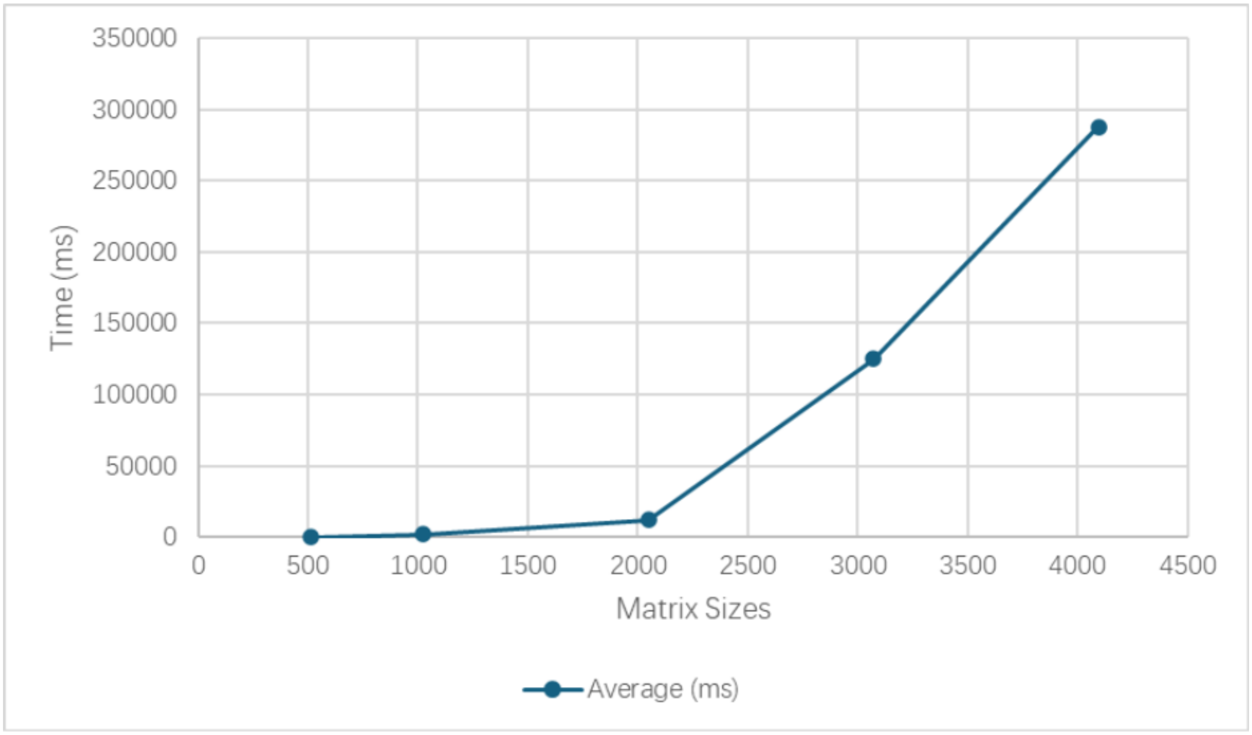


Figure9.3.1.1: Average execution time of sequential matrix multiplication as a function of matrix size N.

Figure 9.3.1.1 illustrates the average execution time of the sequential matrix multiplication as the matrix size increases. The results demonstrate a rapid increase in execution time consistent with the O (N^3) computational complexity of the classical matrix multiplication algorithm. The sub-linear speedup observed as thread count increases can be attributed to memory bandwidth contention and cache coherence overheads, which commonly limit scalability in multicore matrix computations (Pacheco, 2011).

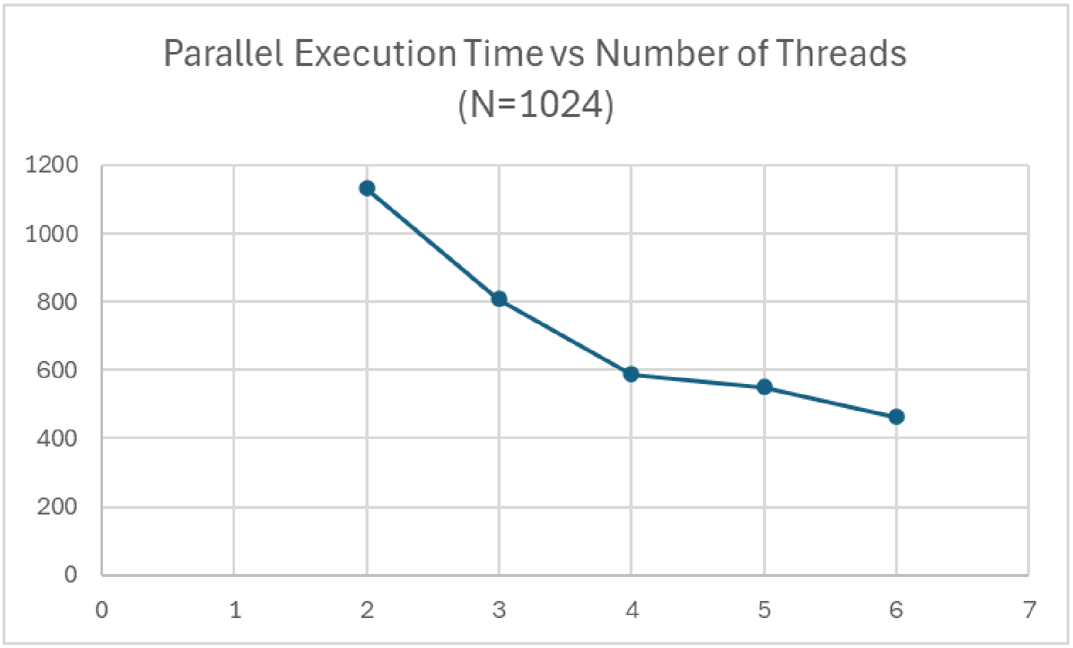


Figure9.3.1.2: Parallel execution time for matrix multiplication with N=1024 as a function of thread count.

FIGURE9.3.1.2 shows the effect of increasing the number of threads on parallel execution time. For all tested matrix sizes, increasing the number of threads significantly reduces execution time. The reduction is more pronounced for larger matrices, where computation dominates thread management overhead.

SEQUENTIAL (LU)

N	1st (ms)	2nd (ms)	3rd (ms)	4th (ms)	5th (ms)	AVG (ms)
512	21.40	27.72	22.47	24.42	27.52	24.71
1024	130.04	125.52	128.38	122.17	107.52	122.73
2048	3789.71	3410.10	3150.00	3511.68	3506.69	3473.64
3072	11455.31	12424.74	11622.50	11429.18	11493.62	11685.07
4096	22455.83	21680.04	21188.11	21895.11	22815.83	22006.98

Table9.3.1.3: Sequential performance for LU Decomposition.

The sequential LU decomposition results exhibit the expected cubic growth in execution time as matrix size increases. This behaviour aligns with the theoretical O(N^3) complexity of LU factorisation. For small matrix sizes, execution time is relatively low, while for large matrices the computational cost increases rapidly.

PARALLEL (LU)

N=512

Nodes	1st (ms)	2nd (ms)	3rd (ms)	4th (ms)	5th (ms)	Ave (ms)	Parallel Speedup
2	25.4	30.3	30.8	34.2	21.2	28.3	0.8
3	35.06	31.3	30.7	28.87	33.2	31.8	0.7
4	32.9	48.3	44.8	34.3	38.0	39.6	0.6
5	38.6	47.8	32.3	32.1	41.0	38.3	0.6
6	40.0	53.3	32.1	41.2	36.6	40.6	0.6

N=1024

Nodes	1st (ms)	2nd (ms)	3rd (ms)	4th (ms)	5th (ms)	Ave (ms)	Parallel Speedup
2	152.1	118.5	137.9	147.8	132.6	137.7	0.89
3	130.2	184.3	123.1	169.3	144.7	150.3	0.81
4	186.9	137.3	130.4	181.8	172.6	161.8	0.75
5	187.7	200.8	185.7	207.3	187.1	193.7	0.63
6	190.2	162.8	242.3	153.7	192.5	188.3	0.65

N=2048

Nodes	1st (ms)	2nd (ms)	3rd (ms)	4th (ms)	5th (ms)	Ave (ms)	Parallel Speedup
2	2751.3	2664.9	2732.7	2592.9	2658.9	2233.78	1.55
3	2614.5	2573.8	2568.4	2634.42	2591.5	2164.27	1.60
4	2579.8	2679.3	2444.2	2626.6	2696.0	2171.65	1.59
5	2570.4	2593.5	2602.7	2659.2	2676.4	2184.53	1.59
6	2510.8	2526.6	2646.3	2632.8	2845.3	2194.63	1.58

N=3072

Nodes	1st (ms)	2nd (ms)	3rd (ms)	4th (ms)	5th (ms)	Ave (ms)	Parallel Speedup
2	9904.6	9974.9	9822.1	10164.1	9831.4	9939.4	1.17
3	10092.0	9924.2	9956.5	9807.4	9847.4	9925.5	1.17
4	9991.2	9915.3	10103.3	9918.5	10045.7	9994.8	1.17
5	10140.0	10529.7	9906.1	10036.0	10018.2	10126.0	1.15
6	11335.7	10976.6	10601.0	10472.5	10299.8	10737.1	1.08

N=4096

Nodes	**1st (ms)**	**2nd (ms)**	**3rd (ms)**	**4th (ms)**	**5th (ms)**	**Ave (ms)**	**Parallel Speedup**
2	22479.6	21751.4	22084.9	22579.8	22577.7	18579.23	1.18
3	22236.4	23724.0	23511.6	22820.9	23724.0	19336.65	1.13
4	22521.5	22040.1	21975.9	22562.9	22161.9	18544.38	1.18
5	22533.8	22271.0	22881.1	22214.7	22122.1	18671.28	1.17
6	23451.6	22451.6	22346.7	22763.5	22912.1	18988.58	1.16

Table9.3.1.4: Parallel performance for LU Decomposition

The parallel LU implementation achieved limited speedup compared to the sequential version. For small matrix sizes (N=512, N=1024), parallel execution often resulted in slower performance than the sequential baseline. This behaviour is due to thread management and synchronisation overhead dominating the computation when the workload per iteration is small.

For medium matrix sizes (N=2048), modest speedups were observed, with peak speedups of approximately 1.5. For larger matrix sizes (N=3072 and N=4096), speedup diminished again, typically remaining near 1.1 to 1.2. This reduction is attributed to memory bandwidth limitations and increased cache contention between threads.

Unlike matrix multiplication, which is embarrassingly parallel and achieved strong scaling using row-wise decomposition, LU decomposition exhibits limited scalability. LU factorisation contains inherent sequential dependencies between elimination steps, requiring global synchronisation at each iteration. Additionally, the amount of parallel work decreases as the algorithm progresses, further limiting achievable speedup.

9.3.2 MPJ

N = 512	**1st (ms)**	**2nd (ms)**	**3rd (ms)**	**4th (ms)**	**5th (ms)**	**AVG (ms)**	**Speedup**
Sequential	334	315	330	316	328	324.6	--
2	258	260	266	273	272	265.8	1.22
4	178	191	189	189	182	185.8	1.75
8	180	180	178	185	177	180	1.80
16	273	240	249	250	262	254.8	1.27

Table 9.3.2.1: Results for parallel matrix multiplication.

N = 1024	1st (ms)	2nd (ms)	3rd (ms)	4th (ms)	5th (ms)	AVG (ms)	Speedup
Sequential	2507	2058	2104	2047	2088	2160.8	--
2	1235	1248	1239	1248	1264	1246.8	1.73
4	887	754	936	814	777	833.6	2.59
8	683	727	700	757	741	721.6	2.99
16	867	782	873	815	870	841.4	2.57

Table 9.3.2.2: Results for parallel matrix multiplication.

N = 2048	1st (ms)	2nd (ms)	3rd (ms)	4th (ms)	5th (ms)	AVG (ms)	Speedup
Sequential	40947	40557	40551	40361	40721	40627.4	--
2	19138	19072	18744	18802	18809	18913	2.15
4	10215	9738	10149	9946	9945	9998.6	4.06
8	6642	6508	6501	6338	6342	6466.2	6.28
16	4675	4997	5101	4642	4740	4831	8.41

Table 9.3.2.3: Results for parallel matrix multiplication.

N = 3072	1st (ms)	2nd (ms)	3rd (ms)	4th (ms)	5th (ms)	AVG (ms)	Speedup
Sequential	160325	163198	160335	158400	160368	160525.2	--
2	81876	81122	81403	82707	81300	81681.6	1.97
4	40101	41623	41900	41780	41767	41434.2	3.87
8	21704	21897	22709	22564	22909	22356.6	7.18
16	14730	14531	14320	14663	15746	14798	10.85

Table 9.3.2.4: Results for parallel matrix multiplication.

N = 4096	1st (ms)	2nd (ms)	3rd (ms)	4th (ms)	5th (ms)	AVG (ms)	Speedup
Sequential	367066	380418	382485	380628	364832	375085.8	--
2	192866	185503	193096	185480	193955	190180	1.97
4	98681	98804	98185	98181	98082	98386.6	3.81
8	52201	52301	52173	52956	51880	52302.2	7.17
16	41215	43790	40852	43121	40066	41808.8	8.97

Table 9.3.2.5: Results for parallel matrix multiplication.

The results for parallel matrix multiplication show a clear improvement in performance as the number of processes increases, particularly for larger matrix sizes.

N	Sequential (ms)	2 Threads (ms)	4 Threads (ms)	8 Threads (ms)	16 Threads (ms)
512	324.6	265.8	185.8	180.0	254.8
1024	2160.8	1246.8	833.6	721.6	841.4
2048	40627.4	18913.0	9998.6	6466.2	4831.0
3072	160525.2	81681.6	41434.2	22356.6	14798.0
4096	375085.8	190180.0	98386.6	52302.2	41808.9

Table 9.3.2.6: Average Execution Time for MPJ Matrix Multiplication (ms).

N	2 Threads (ms)	4 Threads (ms)	8 Threads (ms)	16 Threads (ms)
512	1.22	1.75	1.8	1.27
1024	1.73	2.59	2.99	2.57
2048	2.15	4.06	6.28	8.41
3072	1.97	3.87	7.18	10.85
4096	1.97	3.81	7.17	8.97

Table 9.3.2.7: Speedup for MPJ Matrix Multiplication.

The performance improvement occurs because matrix multiplication involves a large number of independent arithmetic operations and very little synchronisation. Once data is distributed, each process performs computation almost entirely independently, allowing MPJ Express to exploit parallelism efficiently. For smaller matrices, communication overhead becomes more noticeable, which explains the limited gains and slight performance degradation at higher process counts.

In contrast, the LU decomposition results reveal significantly different behaviour.

N = 512	1st (ms)	2nd (ms)	3rd (ms)	4th (ms)	5th (ms)	AVG (ms)	Speedup
Sequential	73	85	73	65	70	73.20	--
2	793	799	807	790	839	805.60	0.08
4	857	872	817	852	811	841.80	0.08
8	1038	1098	1101	1034	1038	1061.80	0.06
16	1293	1325	1286	1280	1319	1300.60	0.05

Table 9.3.2.8: Results for LU decomposition.

N = 1024	1st (ms)	2nd (ms)	3rd (ms)	4th (ms)	5th (ms)	AVG (ms)	Speedup
Sequential	327	321	346	321	322	327.40	--
2	1722	1702	1755	1640	1704	1704.60	0.192
4	1741	1670	1670	1775	1698	1710.80	0.191
8	2191	2206	2149	2214	2231	2198.20	0.148
16	2638	2642	2689	2629	2669	2653.40	0.123

Table 9.3.2.9: Results for LU decomposition.

N = 2048	1st (ms)	2nd (ms)	3rd (ms)	4th (ms)	5th (ms)	AVG (ms)	Speedup
Sequential	2675	2662	2629	2654	2648	2653.6	--
2	4787	4873	4703	4971	4716	4810	0.5615
4	4844	5015	5085	5019	4839	4960.4	0.535
8	5950	6050	6006	6077	5992	6015	0.4412
16	6811	6893	6830	6830	6857	6844.2	0.3877

Table 9.3.2.10: Results for LU decomposition.

N = 3072	1st (ms)	2nd (ms)	3rd (ms)	4th (ms)	5th (ms)	AVG (ms)	Speedup
Sequential	10320	9924	10009	9901	10376	10106.00	--
2	11249	11590	11387	11666	11292	11436.80	0.9189
4	11181	10944	10584	10653	10879	10848.2	0.9538
8	12524	12125	12342	12692	12526	12441.8	0.8284
16	13891	14055	14016	14086	14054	14020.4	0.7383

Table 9.3.2.11: Results for LU decomposition.

N = 4096	1st (ms)	2nd (ms)	3rd (ms)	4th (ms)	5th (ms)	AVG (ms)	Speedup
Sequential	24892	24579	24889	24944	24913	24843.4	--
2	24883	24538	25727	24934	24325	24881.4	1.0242
4	18470	18727	18217	18471	18444	18465.8	1.3507
8	21328	20443	20958	21488	21081	21059.6	1.1818
16	23810	23944	23796	23737	23878	23833.0	1.0433

Table 9.3.2.12: Results for LU decomposition.

N	Sequential	2 Threads	4 Threads	8 Threads	16 Threads
512	73.2	805.6	841.8	1061.8	1300.6
1024	327.4	1704.6	1710.8	2198.2	2653.4
2048	2653.6	4810.0	4960.4	6015.0	6844.2
3072	10106.0	11436.8	10848.2	12441.8	14020.4
4096	24843.4	24881.4	18465.8	21059.6	23833.0

Table 9.3.2.13: Average Execution Time for LU Decomposition (ms).

N	2 Threads	4 Threads	8 Threads	16 Threads
512	0.08	0.08	0.06	0.05
1024	0.19	0.19	0.14	0.12
2048	0.56	0.53	0.44	0.38
3072	0.91	0.95	0.82	0.73
4096	1.02	1.35	1.18	1.04

Table 9.3.2.14: Speedup for LU Decomposition.

Comparing average Execution Time for LU Decomposition clearly shows that, for most matrix sizes, the parallel LU decomposition takes significantly longer than the sequential version. The speedups further illustrates that speedup remains below 1 for most configurations, indicating a slowdown rather than an improvement.

This behaviour is a direct consequence of the algorithm's structure. LU decomposition progresses column by column, and each step depends on the results of the previous iteration. As a result, only a limited portion of the computation can be parallelised. Furthermore, at every iteration, a pivot row must be broadcast to all processes, followed by a global barrier synchronisation. For large matrices, this results in thousands of communication and synchronisation operations, which dominate the execution time.

For matrix sizes up to N=3072, the amount of computation per iteration is insufficient to offset the cost of communication, making the parallel version consistently slower than the sequential one. However, at N=4096, a different trend emerges. The increased computational workload begins to outweigh communication overhead when using 2 and 4 processes, resulting in observable speedup. This indicates that the problem size has reached a point where parallel computation becomes beneficial.

The performance then declines again for 8 and 16 processes. At higher process counts, communication overhead increases substantially, while the amount of work performed by each process decreases. This leads to increased idle time and network contention, causing overall performance to degrade.

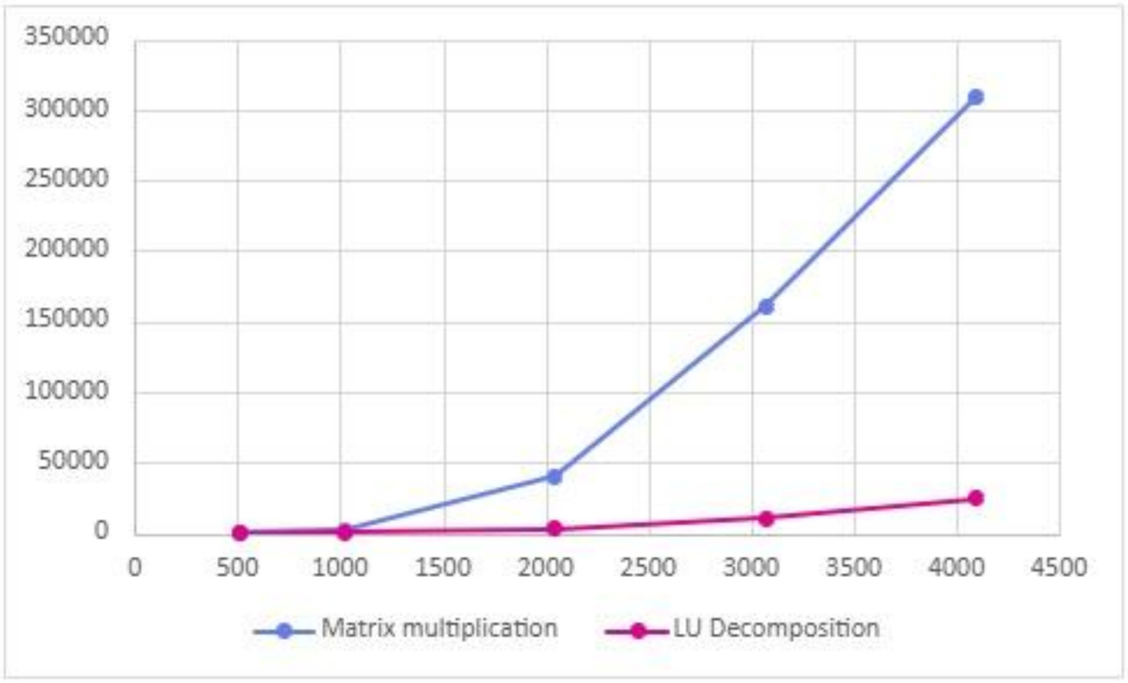


Figure9.3.2.1: Sequential Matrix Multiplication vs LU Decomposition.

The results show that execution time increases much faster for sequential matrix multiplication than for sequential LU decomposition as matrix size grows. In contrast, LU decomposition grows more gradually and remains significantly faster in the sequential case, reflecting its lower arithmetic workload despite its more complex algorithmic structure.

9.3.3 GPU Aparapi

The performance of GPU implementation using the Aparapi framework was evaluated against sequential benchmark data as well. The following tables present the consolidated experimental results, with the execution times measured in milliseconds and computational throughput in GFLOPS.

Nodes	**1st (ms)**	**2nd (ms)**	**3rd (ms)**	**4th (ms)**	**5th (ms)**	**Ave (ms)**	**GFLOPS**
512	192	176	170	170	166	175	1.54
1024	1852	1938	1855	1859	1816	1864	1.15
2048	11743	11615	11577	11775	13129	11968	1.44
3072	118769	126249	126737	126368	125384	124702	0.46
4096	290978	298139	294487	273736	280276	287523	0.48

Table9.3.3.1: Baseline Performance of Sequential Matrix Multiplication

Nodes	1st (ms)	2nd (ms)	3rd (ms)	4th (ms)	5th (ms)	Ave (ms)	GFLOPS
512	22	22	22	23	24	23	11.77
1024	181	192	188	186	189	187	9.45
2048	1453	1460	1497	1499	1461	1474	11.66
3072	6319	5911	6005	6200	6044	6096	9.51
4096	12683	12670	12801	12824	12575	12710	10.81

Table9.3.3.2: Performance of Aparapi API Matrix Multiplication

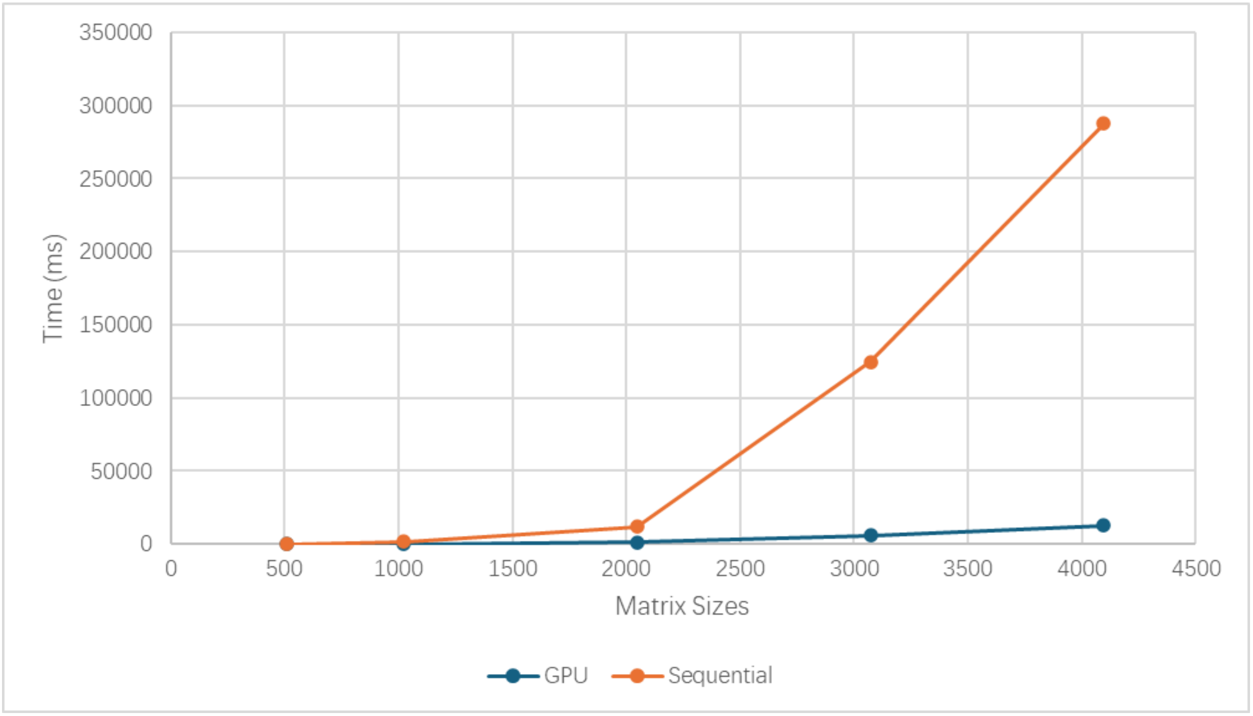


Figure9.3.3.1: Comparison of GPU and Sequential Matrix Multiplication Average Run Time (ms).

The performance shows a strong and scalable advantage when execute on the GPU. The parallel speedup exhibits a generally increasing trend with problem size, escalating from 7.66 for 512x512 matrix to 22.62 for a 4096x4094 matrix. We can see an increasing trend in parallel speedup with a few suboptimal. The factor dips to 8.12 for N=2048 before rising sharply for the larger dimensions. This could be due to memory bandwidth with memory access with specific problem size. Despite the fluctuation, we can see that as workload increases with the sizes, other factors such as kernel launch latency and data transfer are amortised (Kirk and Hwu, 2016).

The performance of GPU is around 9.5 to 11.8 GFLOPS for our experiments. This consistency suggests that the performance is primarily constrained by the sustainable memory bandwidth of the GPU or it could already reach its peak theoretical floating-point capability.

Nodes	**1st (ms)**	**2nd (ms)**	**3rd (ms)**	**4th (ms)**	**5th (ms)**	**Ave (ms)**	**GFLOPS**
512	21	28	22	24	28	25	10.86
1024	130	126	128	122	108	123	17.50
2048	3790	3410	3150	3512	3507	3474	4.95
3072	11455	12425	11623	11429	11494	11685	4.96
4096	22456	21680	21188	21895	22816	22007	6.25

Table9.3.3.3: Sequential Baseline Performance for LU Decomposition.

Nodes	**1st (ms)**	**2nd (ms)**	**3rd (ms)**	**4th (ms)**	**5th (ms)**	**Ave (ms)**	**GFLOPS**
512	23	25	24	24	25	23	11.51
1024	184	184	186	183	183	184	11.69
2048	1406	1400	1405	1401	1410	1404	12.23
3072	5880	5903	5890	5903	5899	5895	9.84
4096	12074	12031	12038	12029	12061	12047	11.41

Table9.3.3.4: Aparapi GPU Performance for LU Decomposition.

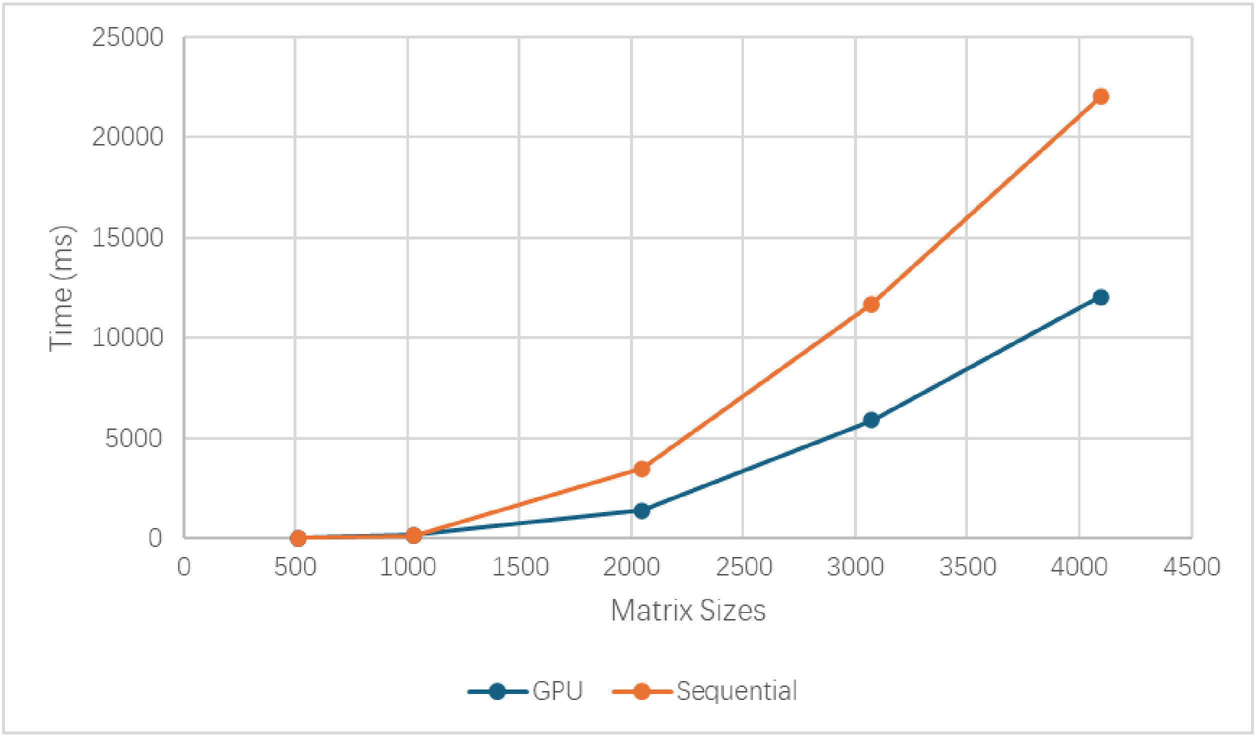


Figure9.3.3.2: Comparison of GPU and Sequential LU Decomposition Average Run Time (ms).

The performance for LU decomposition shows a more complex and nuanced pattern. Opposite to the matrix multiplication's trend, the sequential LU decomposition implementation shows a high performance with smaller matrices at beginning with 10.86 GFLOPS for N=512, and 17.50 GFLOPS for N=1024. Hence, the parallel speedup for smaller problem sizes failed to outperform the sequential part. This could be due to kernel launch latency with CPU running in the outer loop consume more times than the just uses CPU to runs with both outer and inner loops. However, a crossover point is reached at N= 2048, where the size is sufficient for the time of these overheads to be a smaller fraction of the cost, and let the calculation take the dominance. So that the GPU's performance beat the sequential implementation. The GPU performance ranging from 9.84 to 12.23 GFLOPS during the LU decomposition experiment. This suggests a similar conclusion for the matrix multiplication that the GPU's capability might be limited by the bandwidth of memory shared speed, or it could already reach its peak theoretical floating-point capability.

For both operations and their parallel versions, we can find some common behaviours. The performance for both algorithms has a similar range (around 9 to 12 GFLOPS), strongly indicating that the performance is bound by the end GPU's memory subsystem bandwidth or its peak floating-point capacity. Furthermore, a performance threshold is evident, below which the fixed costs of GPU execution negate any benefit from parallelisation. For smaller problem sizes, the fixed overheads associated with the programming framework take a larger portion of the running time.

Particularly, kernel launch and data transfer become decisive factors in GPU implementation (Gupta, 2013). The situation is more significant for LU decomposition than the matrix multiplication operation. Yet, this issue is negligible when the problem size increases.

9.3.4 Comparison

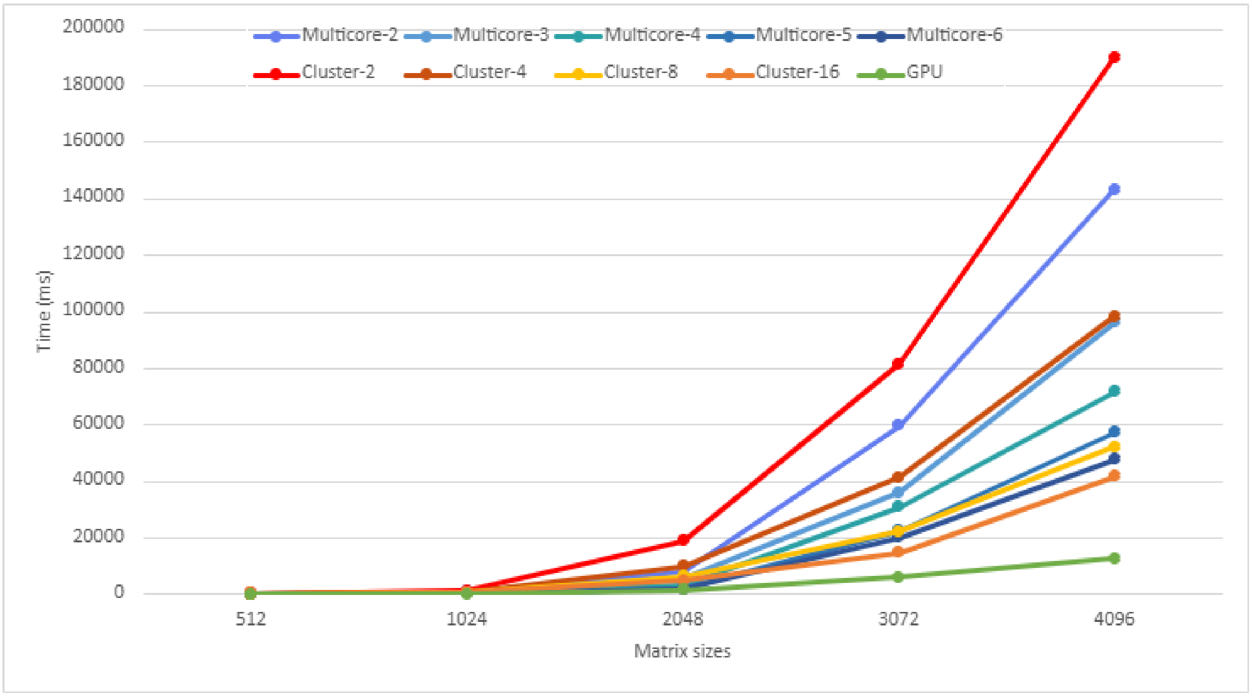


Figure9.3.4.1: Results of Multicore, Cluster Mode and Using GPU to Run Matrix Multiplication under Different Matrix Sizes.

The Figure9.3.4.1 demonstrates that parallel performance is strongly problem-size dependent. While multicore and cluster approaches offer incremental improvements, GPU acceleration delivers the highest performance and best scalability, particularly as matrix sizes grow large. This reinforces the conclusion that for large-scale numerical workloads, GPUs are the most efficient parallel platform, whereas multicore and cluster approaches are better suited to smaller or moderately sized problems.

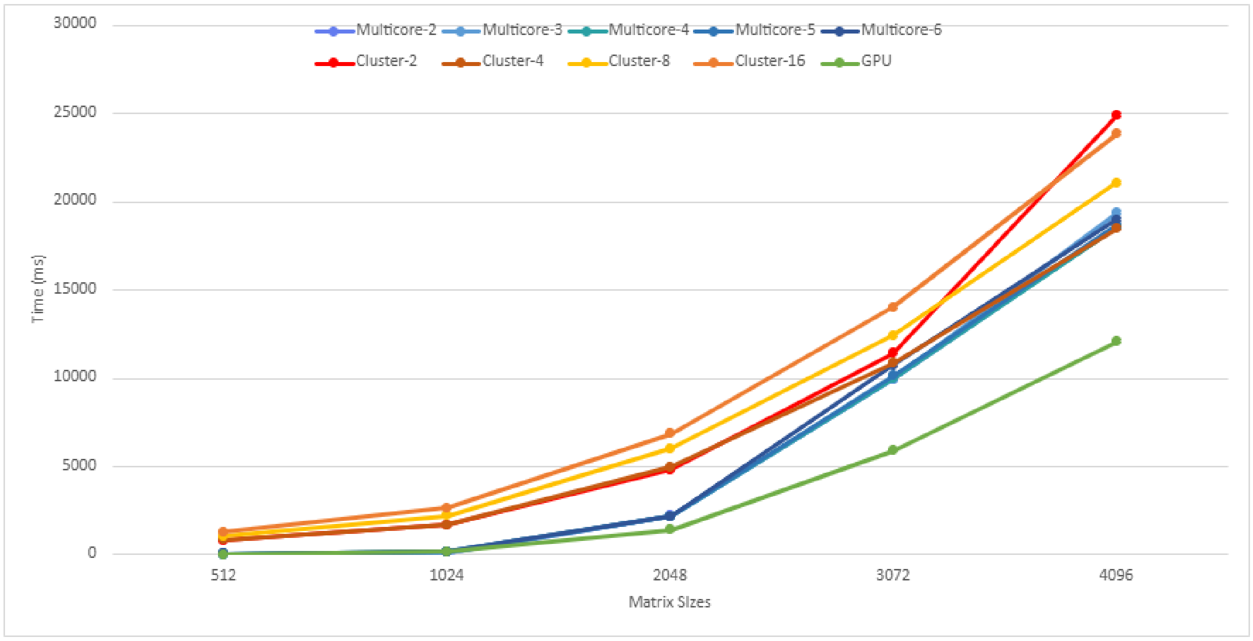


Figure9.3.4.2: Results of Multicore, Cluster Mode and Using GPU for LU Decomposition under Different Matrix Sizes.

For LU decomposition while the GPU still offers the best performance at large matrix sizes as can be seem in the Figure9.3.4.2, the gains are constrained by the algorithm's inherent sequential structure. Multicore implementations provide moderate improvements, whereas MPJ cluster implementations suffer from excessive communication and synchronisation overhead. Overall, LU decomposition favours shared-memory and GPU-based approaches over distributed-memory clustering, especially as problem size increases.

9.4 Conclusion

This project shows that the best parallel method depends on the algorithm's dependency structure and the platform overheads. Matrix multiplication benefits strongly from parallelism because most computations are independent, so all three approaches improve performance, with GPU using Aparapi API providing the largest gains at large matrix sizes and MPJ/multicore offering solid improvements depending on communication and core limits.

In contrast, LU decomposition contains intrinsic sequential dependencies that force synchronisation at each factorisation step, which limits scalability on multicore and makes MPJ cluster performance particularly poor due to repeated communication. GPU acceleration helps mainly when the matrix is large enough to outweigh kernel/coordination overhead, reinforcing the key lesson: parallel speedup is driven as much by algorithm structure and overheads (memory bandwidth, synchronisation, communication) as by raw hardware capability.

References

G Amdahl, G.M. (1967) Validity of the single processor approach to achieving large scale computing capabilities. AFIPS Conference Proceedings. rama,

Ananth. (2003). Introduction to parallel computing. 636.

Kirk, David., & Hwu, W. (2017). Programming massively parallel processors : a hands-on approach.

Pacheco, P. S. . (2011). An introduction to parallel programming. 370.

Syncleus. (2021, July 12). GitHub - Syncleus/aparapi: The New Official Aparapi: a framework for executing native Java and Scala code on the GPU. GitHub. https://github.com/Syncleus/aparapi

Gupta, K. G., Agrawal, N., & Samrit Kumar Maity. (2013). Performance analysis between aparapi (a parallel API) and JAVA by implementing sobel edge detection Algorithm. 1–5. https://doi.org/10.1109/parcomptech.2013.6621408