High Performance Computing - Exercise 2 (C)

Mandelbot Set using MPI + OpenMP

Andrea Buscema

Introduction

In this report, will be discussed the results of the second exercise of the High Performance Computing assignment. The objective of this exercise is to implement the Mandelbrot Set using MPI and OpenMP, obtaining a single pgm image as output, and analysing the scalability of the implementation: OMP scaling: run with a single MPI task and increase the number of OMP threads; MPI scaling: run with a single OMP thread per MPI task and increase the number of MPI tasks; Hybrid scaling (such an extra analysis - not required for the assignment): run for different combinations of MPI and OMP threads.

As for the first exercise/assignment, the project was implemented to be used in the ORFEO HPC Cluster, using THIN nodes. The project was implemented in C language, using the MPI and OpenMP libraries, SLURM for job submission (file .sbatch), and Python for data analysis and plotting.

Description of the project

The Mandelbrot set is a fascinating and well-known construct in complex dynamics, generated by iterating a simple complex function on the complex plane $\mathbb{C}$. Specifically, this set is defined using the function $f_c(z)$ given by $f_c(z) = z^2+c$. Here, c is a complex number $c = x+iy$, and the iteration starts with $z = 0$, producing a series of complex numbers $z_0, z_1, z_2, ...$ defined by $z_0 = 0, z_1 = f_c(0), z_2 = f_c(z_1), \ldots, z_n = f_c^n(z_{n-1})$.

The Mandelbrot set $\mathcal{M}$ consists of all complex points c for which this series remains bounded. A key characteristic of the Mandelbrot set is that if any element z_i in the series has a magnitude greater that 2, the series will eventually become unbounded. Therefore, a point c is considered to be in the Mandelbrot set $\mathcal{M}$ if $|z_n| < 2$for all$n \leq I_{max}$, where I_{max} is a parameter that sets the maximum number of iterations, balancing the accuracy of the calculation with computational cost.

To visualise the Mandelbrot set, we can generate an image representing a portion of the complex plane. This portion is bounded by two corners: the bottom-left corner $c_L = x_L + iy_L$, and the top right corner $c_R = x_R + iy_R$. the image is composed on $n_x \times n_y$ pixels, each corresponding to a point c_i in the complex plane:

$$c_i = (x_L + \Delta x) + i(y_L + \Delta y)$$

where $\Delta x = \frac{x_R - x_L}{n_x}$, and $\Delta y = \frac{y_R - y_L}{n_y}$.

We define a 2D matrix M of integers where each entry $[j][i]$ corresponds to a pixel in the image. The value of each pixel is determined by whether the corresponding point c belongs to the Mandelbrot set $\mathcal{M}$. If c belongs to $\mathcal{M}$, the pixel value is 0. Otherwise, the pixel value is the iteration count n at which the magnitude of $z_n(c)$ exceeds 2, up to a maximum value of I_{max}.

This problem is inherently parallelisable since each point c_i can be computed independently. However, distributing the computational load evenly among concurrent processes or threads can be challenging due to the varying complexity of different regions of the Mandelbrot set. Inner points of $\mathcal{M}$ require more iterations to

determine their membership, whereas outer points are computationally simpler. The bundary between these regions, the "frontier", is particularly complex and requires careful consideration to avoid load imbalance in parallel implementations.

> **Note 1**: Mandelbrot set lives roughly in the circular region centered on $(-0.75, 0)$ with a radius of ~ 2.
>
> **Note 2:** the multiplication of 2 complex numbers is defined as $(x_1 + iy_1) \times (x_2 + iy_2) = (x_1x_2 - y_1y_2) + i(x_1y_2 + x_2y_1)$

With those notes in mind, we can expand basic Mandelbrot set computation to more accurately explore regions within the Mandelbrot set and potentially implement functionality that considers complex number multiplication.

Reason behind the choice of this project

"Bottomless wonders spring from simple rules which are repeated without end." - Benoît B. Mandelbrot, the father of fractal geometry (TED Talk, 02/2010).

The Mandelbrot set calculation project is compelling for several reasons. It visualizes complex dynamics, providing insights into fractal geometry and complex numbers. Generating high-resolution images is computationally demanding, making it ideal for exploring computational methods and performance optimization.

The Mandelbrot set is perfect for parallel computing since each point's calculation is independent. Using MPI (Message Passing Interface) and OpenMP (Open Multi-Processing) leverages both distributed and shared memory architectures, suitable for high-performance computing (HPC) clusters.

Testing the Mandelbrot set on an HPC cluster provides valuable metrics like speedup, efficiency, and scalability. Varying the number of nodes (MPI) and threads per node (OpenMP) helps identify bottlenecks and optimize performance.

The techniques and insights gained from this project have broader applications beyond fractal geometry. They are relevant to other scientific and engineering problems that require high-performance computing, such as climate modeling, molecular dynamics, and large-scale data analysis. Understanding how to efficiently utilize HPC resources is crucial for advancing computational capabilities in these fields. It is a practical and visually engaging way to test and improve computational techniques in a high-performance environment, with implications for a wide range of real-world applications.

Starting point

Since the Mandelbrot set resides within a circle centered around $(-0.75, 0)$ with a radius of ~ 2, we can set our program to generate a grid of points around this region to visualise the set. To do this efficiently, we start by computing a range of points that covers this area.

Implementation steps:

1. Setup a grid: Define a grid of complex numbers centered around $(-0.75, 0)$ with a radius of ~ 2.
2. Iterate over each point: For each point on this grid, determine whether it belongs to the Mandelbrot set using the iterative method.

For the first test, a simple script was implemented where the program writes the output to a file in ASCII mode (txt format).

Inside the local_test folder, a script was created to generate the Mandelbrot set in a txt file, which was then used to create a pgm image with different sizes and resolutions. This initial implementation was done to test and understand the structure of the Mandelbrot set.

Parallelisation

The next step was to parallelize the Mandelbrot set computation using MPI and OpenMP, aiming for a hybrid approach to leverage multiple processors and cores efficiently.

To use MPI (Message Passing Interface), we included MPI functions in the script and initialized MPI in the program. This allows the code to run across multiple processes, which may be distributed across different nodes in a cluster. OpenMP was used for shared-memory parallelism within each MPI process. This enables each process to further split its workload among multiple threads, effectively using multi-core architectures.

Our parallelization strategy involved dividing the grid of points among MPI processes so that each process is responsible for a specific part of the image. This typically involves splitting the image grid into segments (such as horizontal stripes) that each process computes. Once all processes complete their computations, the results must be gathered and assembled into the final PGM image. This can be handled in various ways, including using MPI I/O to write directly to the file in parallel or collecting the results in the root process and writing the final image from there.

For the hybrid script, we first integrated MPI to handle calculations on multiple nodes. Then, we integrated OpenMP to handle calculations on multiple threads within each node. Specific folders were created to store the results and the scripts used to run and obtain the results for each test. First, we created the MPI integration (mpi), then the hybrid with OpenMP integration (mpi_omp2), and finally, we conducted the tests and analysis described in the introduction of this report.

Strategies for the analysis

To analyze the results, we used the time command (/usr/bin/time -v) to measure execution time and report various resource usage statistics. The -v flag stands for "verbose" mode, providing detailed information about the program's execution, including:

- User CPU time (seconds): Total time spent in user mode, excluding kernel time.
- System time (seconds): Total time spent in system mode, including kernel operations like I/O.
- Percent of CPU this job got: Percentage of CPU allocated to the process. Values above 100% indicate the use of more than one CPU core.
- Elapsed (wall clock) time (seconds): Total execution time measured from start to finish.
- Maximum resident set size (kbytes): Maximum amount of physical memory used during execution.
- Major (requiring I/O) page faults: Number of page faults requiring I/O operations, indicating data loading from disk.
- Minor (reclaiming a frame) page faults: Number of page faults not requiring I/O operations, indicating data already available in memory.
- Voluntary context switches: Number of times the process voluntarily gave up the CPU, typically waiting for a resource.
- Involuntary context switches: Number of times the operating system forced the process to hand over the CPU to another process.

Using this data, we can analyze scalability, performance, identify bottlenecks, and measure the impact of parallelization. The chosen metrics help analyze:

- Efficiency: Comparing user time, system time, and total execution time for different numbers of MPI processes and OpenMP threads helps understand resource utilization efficiency.

- Bottlenecks: A high number of major page faults may indicate I/O-related bottlenecks, while a high number of involuntary context switches may indicate excessive competition for processor resources.
- Parallelisation Impact: Increasing CPU percentage and comparing execution times for different MPI and OpenMP configurations shows the effectiveness of parallelisation.

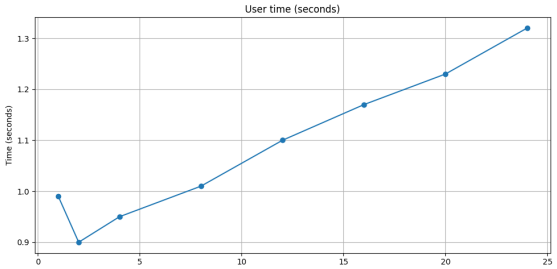
OpenMP Analysis

The script was run with a single MPI task, increasing the number of OpenMP threads. Using the same script, a specific sbatch file was created to run the script with different numbers of threads while collecting detailed timing and resource usage information. For this test, we requested one node, allocating 24 CPUs for the task (all cores on the THIN node).

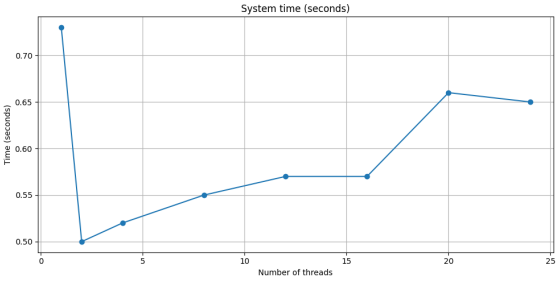
```
## Test for different number of OpenMP threads
for threads in 1 2 4 8 12 16 20 24
do
   export OMP_NUM_THREADS=$threads
   echo "Running with $threads OpenMP threads:"
   /usr/bin/time -v ./mandelbrot_scal_omp
done
```

Efficiency

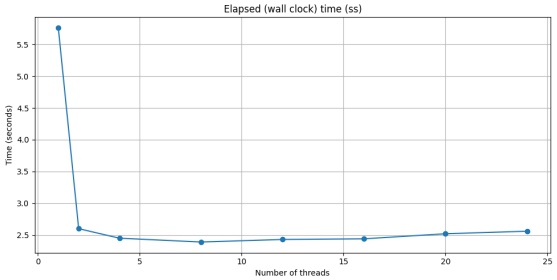
User time: The graph of user time shows an almost linear increase as the number of threads increases. This indicates that the program is performing more overall work with the addition of threads, which is expected if each thread performs a significant portion of the total work. However, a linear increase in user time with more threads could also suggest that there are portions of code that do not benefit from parallelism or that there are synchronizations or communication latencies that limit the effectiveness of parallelization. Ideally, we would like to see a plateau or even a decrease in user time per thread as the number of threads increases, indicating effective use of parallelism to reduce the load on each core.



System time: The system time initially shows a dramatic decrease, stabilizing later and increasing slightly as more threads are added. The initial decrease is positive, indicating that the system overhead per thread decreases when there are few threads. The later increase may reflect the increasing overhead associated with handling more threads. An increase in system time with many threads may indicate that the management of the threads themselves, including synchronization between them, is consuming system resources. Exploring methods of reducing this overhead, such as optimizing critical sections of code or using a different parallelism model, could be useful.



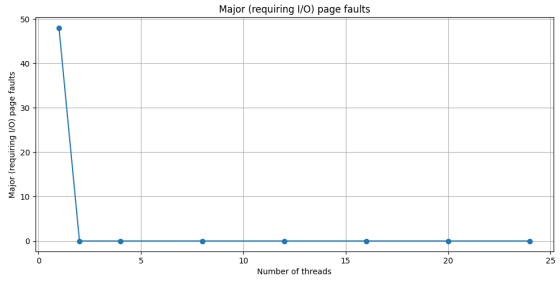
Elapsed (wall clock) time: This graph shows a significant improvement in total execution time as the number of threads increases from 1 to 12, after which the benefits stabilize. This indicates that the application benefits from parallelism up to a certain point, beyond which thread management overheads and resource saturation likely limit further improvements. The plateau in execution time suggests that there are physical limits or code limitations that prevent further improvements. Further optimization of the code or exploration of hardware limitations, such as memory bandwidth or cache conflicts, may be necessary to improve scalability.

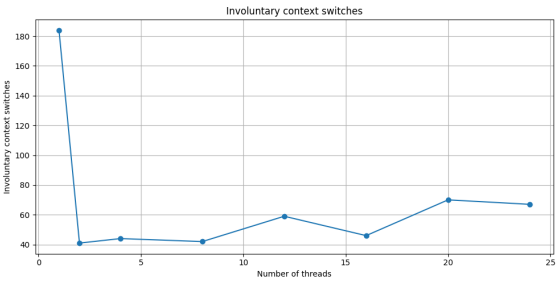


In summary, the results show that the program benefits from the addition of threads up to a certain number, after which the gains in execution time are reduced. This is typical of programs performing heavy calculations that can be effectively parallelised but only as long as no resource bottlenecks or architecture limitations are encountered.

Bottlenecks

Major (requiring I/O) page faults: The graph of major page faults shows a high number of faults when using a single thread, with a drastic decrease after introducing a few additional threads, then stabilizing at zero for subsequent thread increments. High major page faults initially may indicate that the process is accessing a large amount of data for the first time, which is not resident in memory and must be loaded from slower storage devices (such as disk). The rapid decrease with multiple threads suggests that once data is loaded into memory, access becomes more efficient and less likely to cause further page faults. This indicates that memory is efficiently shared and utilized between threads after the initial load.



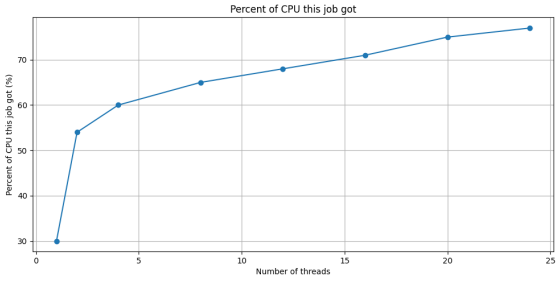


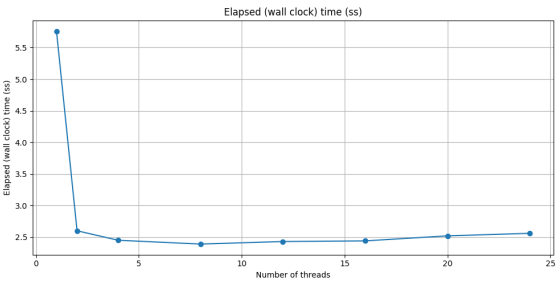
Involuntary context switches: Involuntary context switches show a high frequency with only one thread, followed by a marked decrease and slight fluctuations, but generally remaining at lower levels. A high number of initial involuntary context switches may be due to the operating system frequently managing the process, possibly due to resource waits. The decrease as threads increase indicates that the workload is more evenly distributed among threads, reducing the need for frequent operating system intervention. Fluctuations in later stages may reflect competition for system resources such as CPU or memory, particularly when the number of threads approaches the number of available physical cores.

The significant decrease in both page faults and involuntary context switches as threads increase is positive, indicating that the program scales well in terms of resource management as parallelism increases. However, it is also essential to monitor these metrics to assess whether other types of overhead or limitations, such as memory bandwidth saturation or cache limitations, occur, which may not be immediately apparent from execution times alone.

Parallelisation Impact

Percent of CPU this job got: This graph shows a marked increase in CPU usage as the number of threads increases, starting from a relatively low level with one thread and rising consistently to a plateau around 70 percent. The increase in CPU percentage indicates that adding more threads allows the program to utilize more processor resources simultaneously, suggesting good initial scalability. The plateau may indicate that the program is reaching the limits of scalability on this particular hardware architecture, possibly due to physical limitations such as the number of available cores, memory bandwidth saturation, or the workload no longer being effectively divided between additional threads.





Elapsed (wall clock) Time: The graph of execution time shows a sharp drop in execution time when increasing from 1 to several threads, then stabilizing at a low value after about 4 threads. This is a typical indicator of a good response to initial parallelisation. The rapid reduction in execution time as the first threads increase shows that the program benefits significantly from distributing the workload among multiple threads.

These results indicate that the program achieves good parallelism up to a certain number of threads, beyond which no significant benefits are gained. This information is crucial for resource allocation in a production environment and for optimizing further software development. Based on the results, it may be ideal to configure the program to use a number of threads close to the plateau point to maximize efficiency. Additionally, consider exploring optimizations beyond simply increasing the number of threads, such as improvements in the algorithm, memory management, and synchronization overhead reduction.

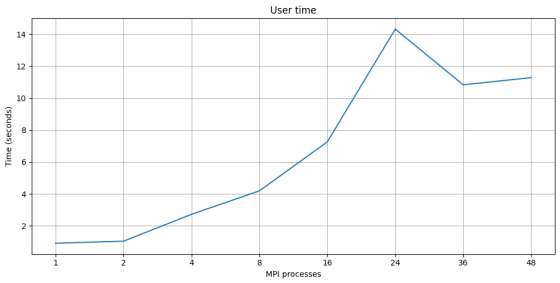
MPI Analysis

The script was run with a single OpenMP thread per MPI task and increasing the number of MPI tasks. For this test was requested 2 nodes, using all the cores available on the THIN nodes.

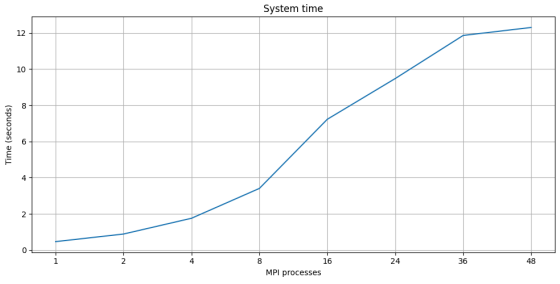
```
## Number of processes MPI to test for scalability
num_procs=(1 2 4 8 16 24 36 48)
## Run the program in loop for each number of processes
for procs in "${num_procs[@]}"
do
    echo "Running with ${procs} processes:"
    /usr/bin/time -v mpirun -np ${procs} ./mandelbrot_scal_mpi
done
```

Efficiency

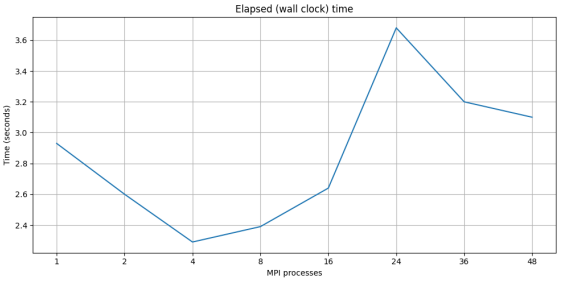
User time: The curve shows an increase up to 16 processes, then a peak at 24 processes before decreasing and finally increasing again with 48 processes. The increase in user time to 16 processes may indicate that the addition of MPI processes does indeed contribute to the computational workload. However, the spike to 24 processes and subsequent decrease may suggest inefficiencies, possibly due to communication congestion or sub-optimal workload distribution.



System time: The steady increase in system time with the increase in MPI processes suggests that there is an increasing system overhead, which may include handling communication between processes. The constant increase is an indicator that more processes result in higher communication or synchronisation costs at system level, which are not compensated for by a proportional increase in computing speed, leading to a decrease in overall efficiency.



Elapsed (wall clock) time: The graph initially shows a decreasing trend up to 4 processes, indicating good scalability. However, the time increases sharply at 16 processes, then increases dramatically at 24 before dropping to 48. The trend in execution time shows that there are peaks of inefficiency, especially at 24 processes. This could be due to increased waiting time for resources or inefficient handling of inter-process communication.

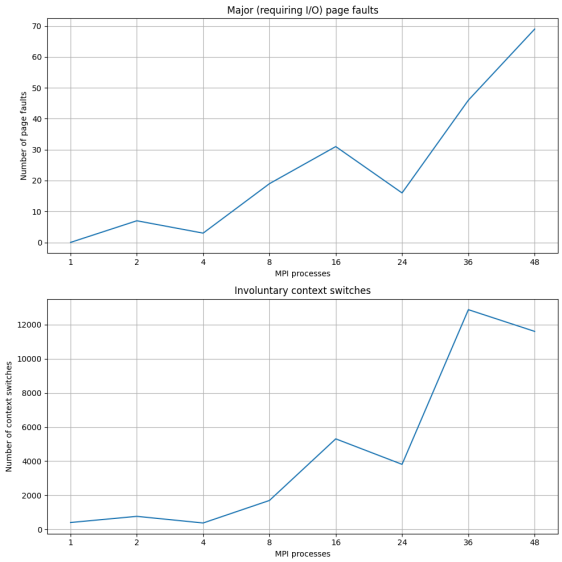


Overall, the scalability of the programme does not appear linear and shows signs of communication and system management

overheads that affect overall efficiency as the number of processes increases. This could be due the nature of the problem, lack of load balancing between processes, congestion in the network or communications, or inefficient utilisation of system resources due to the configuration of the computing topology.

Bottlenecks

Major (requiring I/O) page faults: Major page faults occur when the system has to read a page of memory from the disk because it is not present in RAM. This can happen if the process requires more memory than is physically available, forcing the system to swap pages between RAM and disk. The number of 'Major page faults' increases as the number of MPI processes increases, showing a significant peak at 16 processes, a decrease at 24, and then a further sharp increase to 48. This suggests that the system may be under memory pressure as the number of processes increases, forcing the operating system to resort to I/O operations that are significantly slower than memory operations.

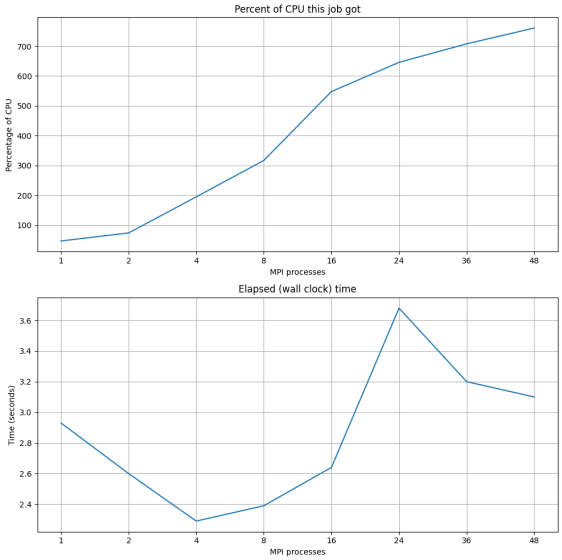


Involuntary context switches: The number of 'Involuntary context switches' initially increases as the number of processes increases, shows a peak at 16 processes, then decreases to 24, and increases dramatically to 36 before decreasing again to 48. This indicates that there are potential inefficiencies in process management and CPU scheduling, especially with a large number of processes.

The increasing frequency of 'major page faults' suggests that it might be worth optimising memory usage within the programme. High 'Involuntary context switches' indicate that the workload may not be distributed efficiently among processes, or that the waiting time for resources (such as memory or I/O) is adversely affecting performance.

Parallelisation Impact

Percent of CPU this job got: The percentage of CPU utilisation increases almost linearly with the increase in the number of MPI processes, indicating that the increase in processes leads to higher overall CPU utilisation. This is a positive sign suggesting good scalability in terms of the ability to handle parallel workloads. However, an increase above 100 per cent indicates that several cores are being used simultaneously, which is expected in a parallel environment such as MPI on a multicore system.



Elapsed (wall clock) Time: This graph shows the total elapsed time from the beginning to the end of programme execution, which is essential for assessing the effectiveness of parallelisation in terms of reducing execution time. The trend shows a significant improvement in execution time as the number of processes increases from 1 to 8, indicating that parallelisation is having a positive impact. However, there is an unexpected

peak at 24 processes, followed by a drop to 36 before a reduction to 48.

Analyses indicate that the programme benefits from parallelisation up to a certain number of processes, beyond which gains are reduced due to management overhead and communication complexity. These results suggest that there are key areas that can be optimised to further improve performance, such as memory management, workload balancing, minimising communication overheads and synchronisation.

Hybrid Analysis

In this extra analysis, it was run the script with different combinations of MPI and OpenMP threads to analyse the scalability of the hybrid implementation. The script was run with a varying number of MPI tasks and OpenMP threads per task. For this test was requested 2 nodes, using half of the cores available on the THIN nodes.

```
## Executing the program with different number of MPI processes and OpenMP threads
for procs in 2 4 8 16 24
do
   for threads in 1 2 4 8 12
   do
      export OMP_NUM_THREADS=$threads
      echo "Running with $procs MPI processes and $threads OpenMP threads:"
      /usr/bin/time -v mpirun -np $procs ./mandelbrot_hybrid
   done
done
```

Efficiency

The efficiency analysis shows that user time, system time, and elapsed time are crucial metrics. User time tends to increase with the number of MPI processes, with a significant rise when using higher OpenMP thread counts, particularly at 12 threads. System time increases linearly with MPI processes, indicating a proportional computational overhead. Elapsed time decreases significantly when moving from 2 to 4 MPI processes, demonstrating good initial scalability. However, beyond 4 MPI processes, the benefits diminish, and elapsed time either stabilises or slightly increases. The optimal configuration for efficiency is around 4-8 MPI processes with 4-8 OpenMP threads, avoiding configurations with 12 threads due to increased overhead.

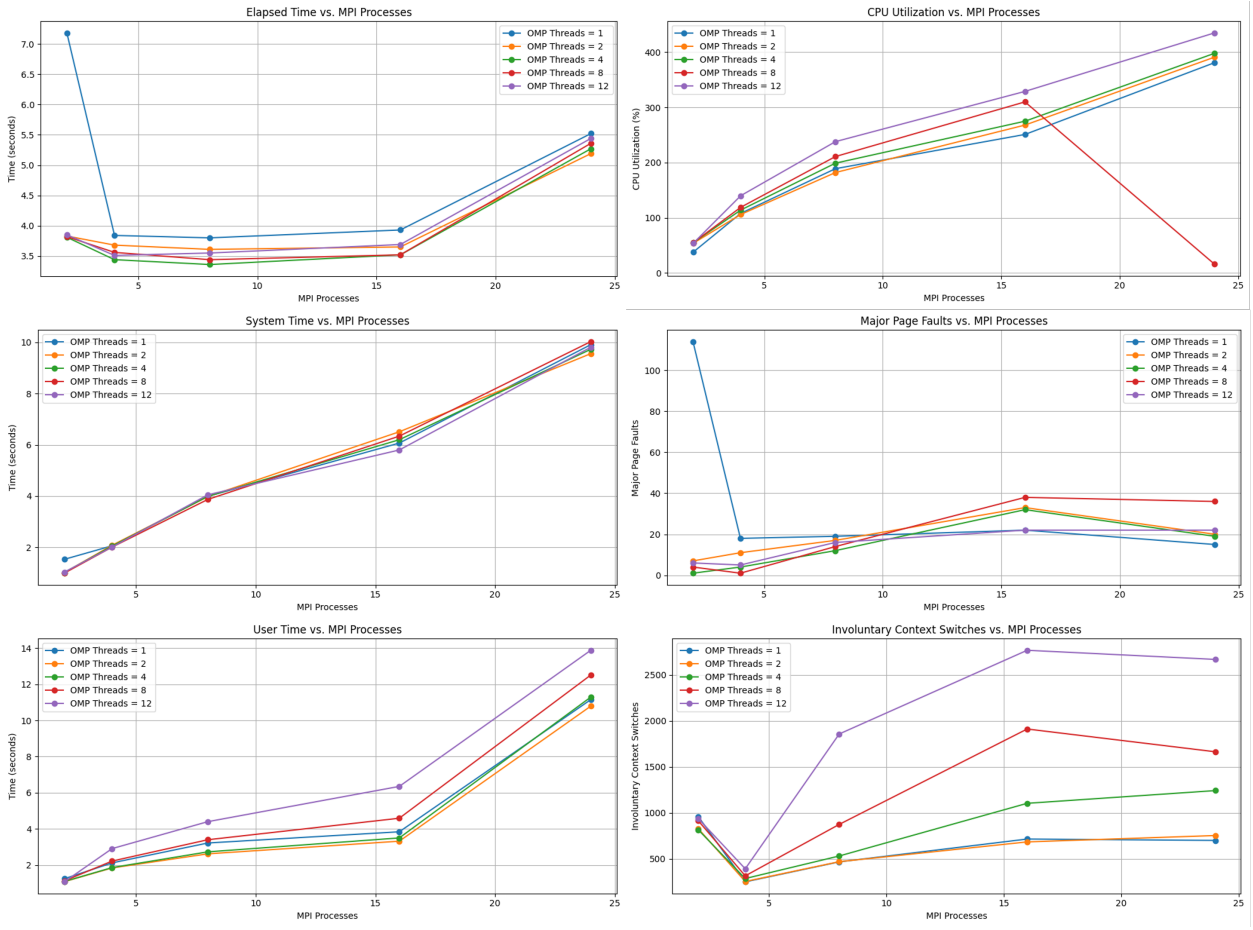
Bottlenecks

The bottleneck analysis focuses on major page faults and involuntary context switches. Involuntary context switches increase with the number of MPI processes, especially at higher OpenMP thread counts, indicating thread management overhead. Major page faults are initially high with 1 thread and 2 MPI processes but stabilize with higher process counts. Configurations with higher thread counts tend to have more page faults, suggesting memory contention or inefficiencies. Minimizing bottlenecks involves using moderate MPI processes (4-8) and OpenMP threads (4-8), avoiding the overhead introduced by high thread counts.

Parallelization Impact

The parallelization impact is assessed through CPU utilization and elapsed time. CPU utilization increases with MPI processes and higher OpenMP thread counts, indicating effective use of multicore resources. An anomaly with 8 OpenMP threads at 24 MPI processes shows a significant drop in CPU utilization, suggesting

resource contention or inefficiencies at higher concurrency levels. Elapsed time decreases significantly from 2 to 4 MPI processes, demonstrating good scalability, but stabilizes or increases slightly beyond this point. Lower thread counts (1-2) result in higher elapsed times compared to higher counts (4-12).



Optimal Configuration and Recommendations

The best performance for the Mandelbrot set computation is achieved with 4-8 MPI processes and 4-8 OpenMP threads. Extremely high thread counts (12) lead to higher CPU utilization but also increased overheads, negatively impacting efficiency.

Things that could be considered to further optimize performance:

- Investigating and optimizing thread management to reduce user time and context switches at high thread counts.
- Optimize memory access patterns to reduce major page faults and improve overall efficiency.
- Conduct additional tests with configurations around 4-8 MPI processes and 4-8 OpenMP threads to refine the optimal setup.
- Use profiling tools to identify and optimize specific functions or operations causing inefficiencies.

References

1. Mandelbrot Set - Wikipedia
2. Mandelbrot Set - Wolfram MathWorld
3. Message Passing Interface (MPI) - Official Website
4. Open Multi-Processing (OpenMP) - Official Website
5. High Performance Computing - ORFEO Cluster