High-Performance Computing Project

Mandelbrot Parallelization

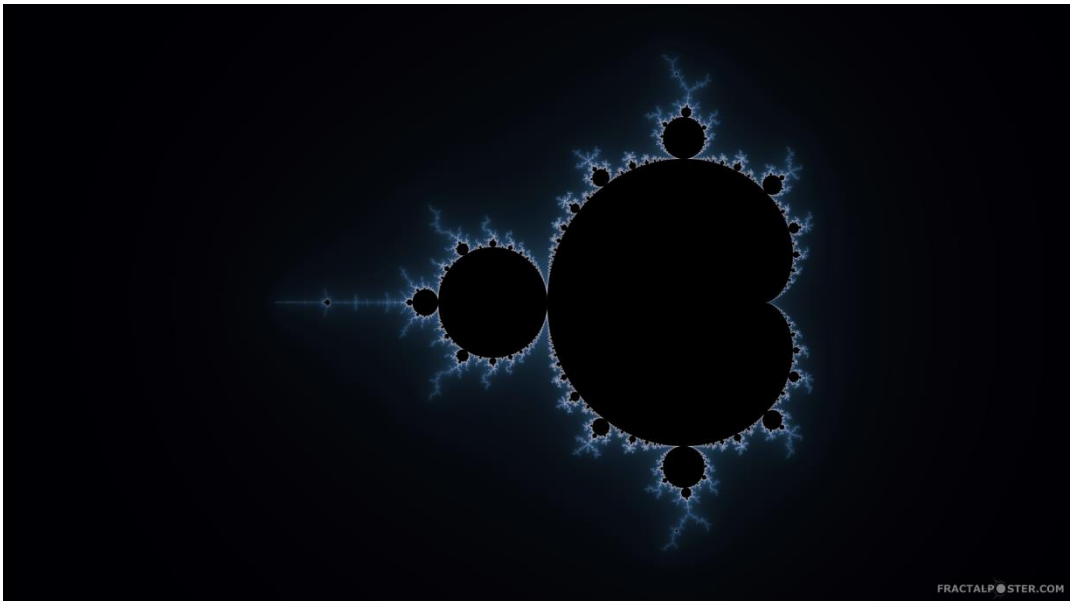
Problem Statement

The Mandelbrot set is a complex and beautiful example of a fractal in mathematics, discovered by Benoît Mandelbrot. It is defined in the complex plane, which is the set of complex numbers. A complex number has the form $a + b\,i$, where a and b are real numbers, and i is the square root of -1.

The Mandelbrot set is generated by the iterative formula:

$$z_{n+1} = z_n^2 + c$$

Here, c is a complex number in the complex plane, and z is a complex variable that starts at zero. For each value of c, the sequence of z values is calculated by repeatedly applying this formula. If the sequence of z values remains bounded (does not go to infinity) no matter how many times the formula is applied, then the complex number c is part of the Mandelbrot set. Visually, the Mandelbrot set is often displayed as a plot on the complex plane, where each point represents a complex number c. Points that are part of the Mandelbrot set are usually coloured black, and the points that are not part of the set (where the sequence goes to infinity) are coloured in various colours depending on the rate at which the sequence diverges. The beauty of the Mandelbrot set truly shines when we visualize it, offering a mesmerizing gateway into the complex and infinite world of fractal geometry as shown in the images down.



Mandelbrot sequential code

This code is a C++ program designed to generate a representation of the Mandelbrot set by mapping each point on a complex plane to a pixel in an image, based on the iteration count at which the sequence defined by $z_{n+1} = z_n^2 + c$ diverges.

It outputs this representation to a file in a Txt format, creating a visualizable dataset of the fractal. When the sequence encounters a number whose absolute value exceeds 2, it is stopped, and the pixel's value is set to the number of iterations performed to reach that point. Conversely, if the sequence remains bounded (its absolute value does not exceed 2) even after the predefined maximum of 1000 iterations (ITERATION), the pixel's initial value, set to 0 at the start, indicates that the point is within the Mandelbrot set, as it did not diverge within the iteration limit.

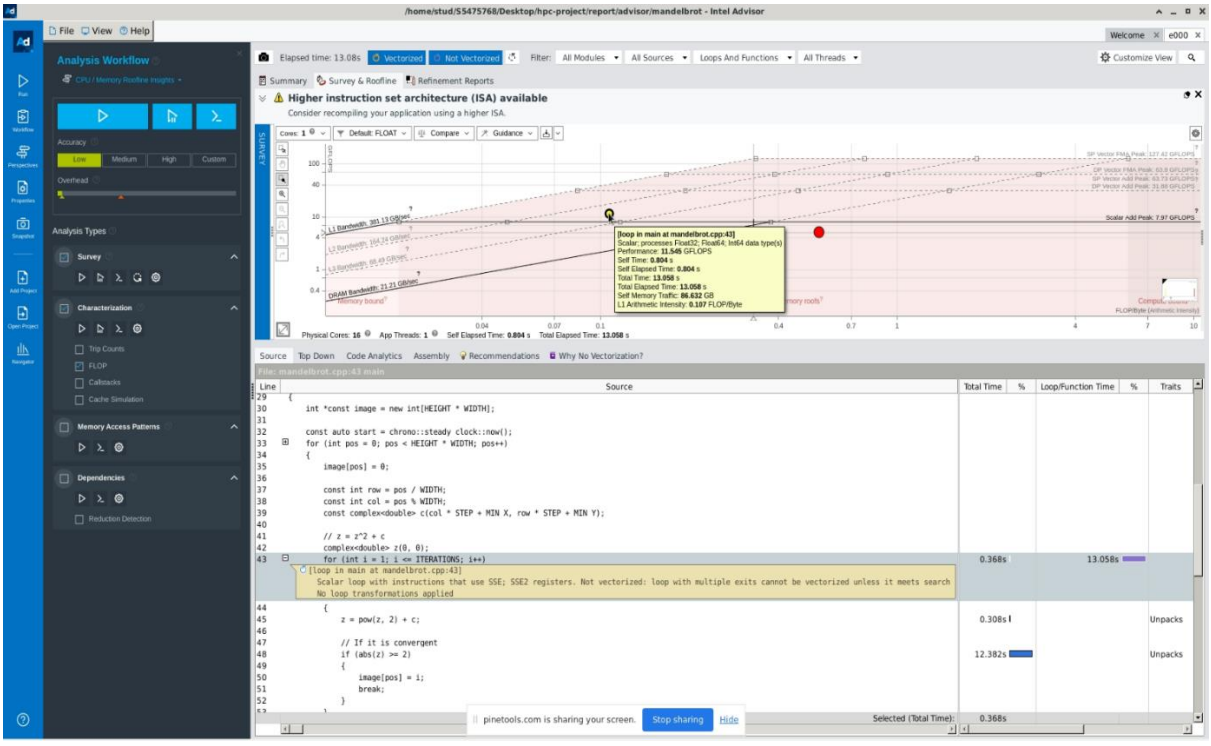
The code operates sequentially, meaning it processes each pixel one after the other in a linear fashion. This method ensures that the computation for each point is completed before moving on to the next. However, this sequential processing approach, while reliable, may not be the most time-efficient, especially for generating high-resolution images or complex fractal calculations, as it does not leverage the potential for parallel processing to compute multiple points simultaneously.

Hot spots

In our case, hot spots could be spotted manually by reviewing the code, since the code is small and the heavy computation hot spot is easy to find.

```cpp
const auto start = chrono::steady_clock::now();
for (int pos = 0; pos < HEIGHT * WIDTH; pos++)
{
    image[pos] = 0;

    const int row = pos / WIDTH;
    const int col = pos % WIDTH;
    const complex<double> c(col * STEP + MIN_X, row * STEP + MIN_Y);

    // z = z^2 + c
    complex<double> z(0, 0);
    for (int i = 1; i <= ITERATIONS; i++)
    {
        z = pow(z, 2) + c;

        // If it is convergent
        if (abs(z) >= 2)
        {
            image[pos] = i;
            break;
        }
    }
}
const auto end = chrono::steady_clock::now();
```

Also, we compiled the sequential code with the *-g* flag and then used *Intel Advisor* to produce the *roofline* report, which matched our manual spotting evaluation.



our manual hot spotting and the intel advisor roofline report, show that the loop iterating to find the Mandelbrot set is the hotspot.

The identified hotspot is a high potential candidate for parallelization, since the computation for each pixel/position inside the image is an independent calculation from the other pixels, while the pixel calculation itself is a dependent process where the output of each iteration during the calculation is used in the next iteration. Parallelization could be implemented on the level of pixel computation, which means that we can compute each pixel separately and then gather all the pixel calculations to form the final image.

Checking results

Before implementing any parallelization method to our sequential code, we need to find a standard way of measuring the errors, since we know that we can obtain a fast and incorrect parallelized implementation “no one is interested in a fast wrong answer” – during lectures.

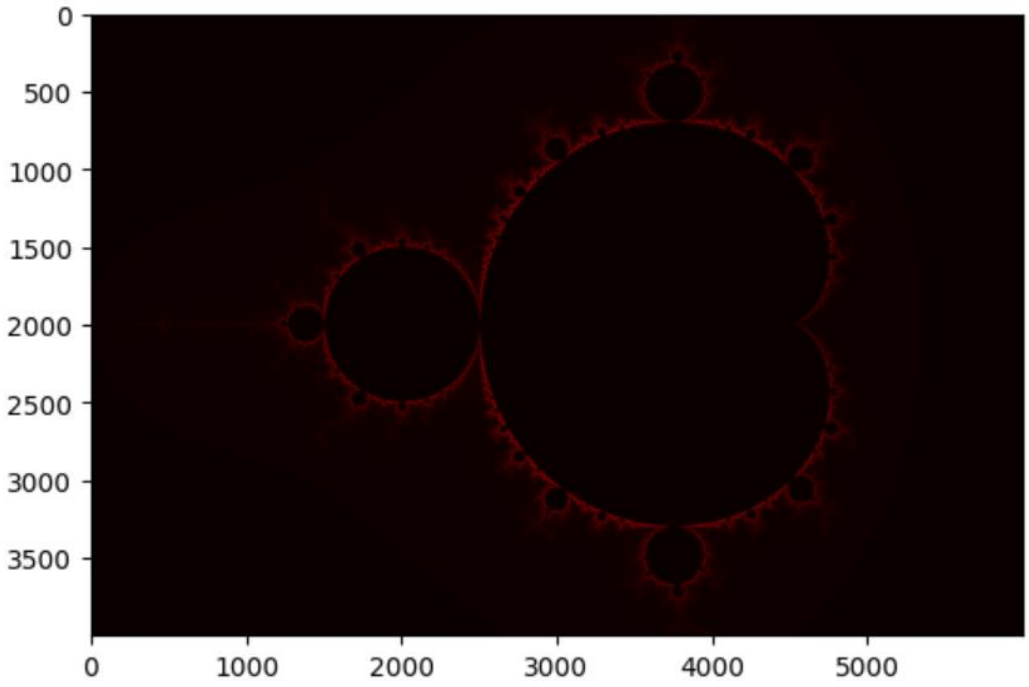
Quantitative method-RMSE(Root Mean Square Error)

We implemented this metric which takes the serial code output as a reference and the parallelized output to compute the root mean square error. While holding different experiments on multiple collections of resolution and iterations, it is mandatory to generate different reference images for different configurations.

$$RMSE = \sqrt{\frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{n}}$$

Visualization method

We have implemented a method to visualize the generated images, this method is not so accurate to measure how precise the parallelization is, but it gave us a rough view of the result's correctness. For visualizing the generated images we developed a Python script by which we utilized matplotlib to visualize the generated image matrix.



Resources

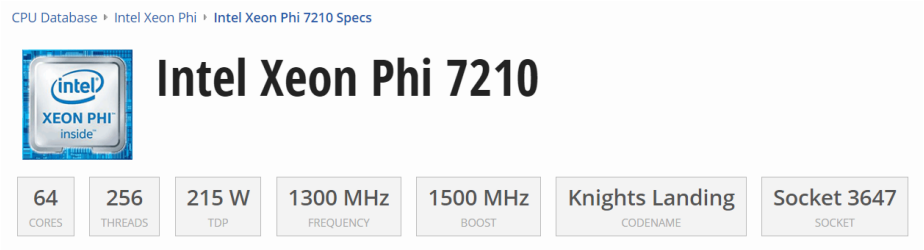
210 Software Room PC/CPU - used for Baseline & OpenMP

```
File  Edit  View  Search  Terminal  Help
S5475768@wk008:~$ lscpu
Architecture:                    x86_64
CPU op-mode(s):                  32-bit, 64-bit
Byte Order:                      Little Endian
Address sizes:                   46 bits physical, 48 bits virtual
CPU(s):                          24
On-line CPU(s) list:             0-23
Thread(s) per core:              1
Core(s) per socket:              16
Socket(s):                       1
NUMA node(s):                    1
Vendor ID:                       GenuineIntel
CPU family:                      6
Model:                           151
Model name:                      12th Gen Intel(R) Core(TM) i9-12900K
Stepping:                        2
CPU MHz:                         3200.000
CPU max MHz:                     5200.0000
CPU min MHz:                     800.0000
BogoMIPS:                        6374.40
Virtualization:                  VT-x
L1d cache:                       384 KiB
L1i cache:                       256 KiB
L2 cache:                        10 MiB
NUMA node0 CPU(s):               0-23
```

By reviewing the exact specs of this CPU we can find it contains 16 cores, 8 of them are p-cores which means it has 24 threads, since Intel Hyper-Threading technology is available and effectively doubles the core-count of the P-Cores, to a total of 24 threads. also, it is noticeable the high frequency of the CPU.

INFN hpcocapie01 - used for MPI

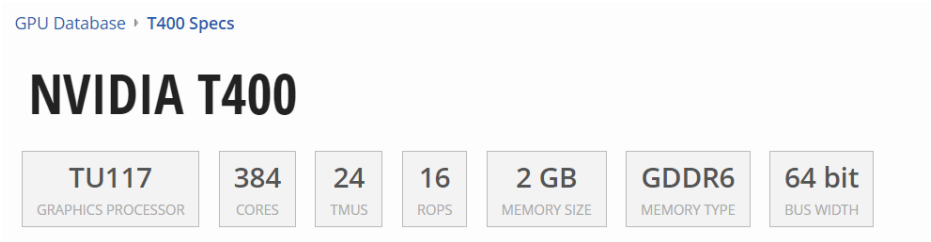
```
[ialtufayli@hpcocapie01 ~]$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                256
On-line CPU(s) list:   0-255
Thread(s) per core:    4
Core(s) per socket:    64
Socket(s):             1
NUMA node(s):          1
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 87
Model name:            Intel(R) Xeon Phi(TM) CPU 7210 @ 1.30GHz
Stepping:              1
CPU MHz:               1000.441
BogoMIPS:              2600.03
L1d cache:             32K
L1i cache:             32K
L2 cache:              1024K
NUMA node0 CPU(s):     0-255
```



It is worth noting here the high number of threads while the CPU frequency is low in comparison with the lab CPU. these specs are a design decision taken by the manufacturer to address the best throughput of the CPU based on the use case.

210 Software Room PC/GPU - used for Cuda

```
S5475768@wk008:~$ nvaccelinfo

CUDA Driver Version:           11040
NVRM version:                  NVIDIA UNIX x86_64 Kernel Module

Device Number:                 0
Device Name:                   NVIDIA T400
Device Revision Number:        7.5
Global Memory Size:            1967259648
Number of Multiprocessors:     6
Concurrent Copy and Execution: Yes
Total Constant Memory:         65536
Total Shared Memory per Block: 49152
Registers per Block:           65536
Warp Size:                     32
Maximum Threads per Block:     1024
Maximum Block Dimensions:      1024, 1024, 64
Maximum Grid Dimensions:       2147483647 x 65535 x 65535
Maximum Memory Pitch:          2147483647B
Texture Alignment:             512B
Clock Rate:                    1425 MHz
Execution Timeout:             Yes
Integrated Device:             No
Can Map Host Memory:           Yes
Compute Mode:                  default
Concurrent Kernels:            Yes
ECC Enabled:                   No
Memory Clock Rate:             5001 MHz
Memory Bus Width:              64 bits
L2 Cache Size:                 524288 bytes
Max Threads Per SMP:           1024
Async Engines:                 3
Unified Addressing:            Yes
Managed Memory:                Yes
Concurrent Managed Memory:     Yes
Preemption Supported:          Yes
Cooperative Launch:            Yes
Default Target:                cc75
```



Two important information are the wrap size and the number of threads per block, that we need to know while developing CUDA kernels.

Baseline

we need to calculate the baseline of the problem by which we can calculate later the speed up and efficiency for each parallelization technique, for this matter we will use the software 201 PC for measuring the time of the sequential code execution with different resolution and iteration configurations. First, we need to apply some optimizations that could speed the serial code execution like *vectorization* and compiling the code with some *optimization flags*. The Intel C++ compiler(icc) was used for the compilation process.

Vectorization

Vectorization is a process performed by compilers to optimize code for execution on SIMD (Single Instruction, Multiple Data) architectures like modern CPUs. Essentially, it transforms sequential scalar operations into parallel vector operations, where a single instruction operates on multiple data elements simultaneously.

Here we examine the vectorization report generated by the icc compiler after compiling with the following command:

```
icc -std=c++11 -O2 -qopenmp -qopt-report -qopt-report-
file=./reports/report -qopt-report-phase=vec ./code/mandelbrot.cpp -o
./builds/exc.o
```

```
Intel(R) Advisor can now assist with vectorization and show optimization
  report messages with your source code.
See "https://software.intel.com/en-us/intel-advisor-xe" for details.


Begin optimization report for: main(int, char **)

    Report from: Vector optimizations [vec]


LOOP BEGIN at mandelbrot.cpp(33,5)
   remark #15541: outer loop was not auto-vectorized: consider using SIMD directive

   LOOP BEGIN at mandelbrot.cpp(43,9)
      remark #15520: loop was not vectorized: loop with multiple exits cannot be vectorized unless it meets search loop idiom criteria   [ mandelbrot.cpp(48,13) ]
   LOOP END
LOOP END

LOOP BEGIN at mandelbrot.cpp(78,9)
   remark #15333: loop was not vectorized: exception handling for a call prevents vectorization   [ mandelbrot.cpp(80,24) ]
LOOP END


Non-optimizable loops:


LOOP BEGIN at mandelbrot.cpp(76,5)
   remark #15333: loop was not vectorized: exception handling for a call prevents vectorization   [ mandelbrot.cpp(80,24) ]
LOOP END
=================================================================
```

The compiler was not able to vectorize loops inside the code due to multiple reasons listed in the vectorization report.

- Iteration loop at line:48 (the hot spot)

  Not vectorized, because of the “break” inside the loop causing multiple exits in the loop, which prevents the vectorization.

- Image writing loops

  Not vectorized because of the potential IO exception while writing images to the desk.

So the execution does not get any benefit from vectorization.

Optimization flags - Intel compiler icc

Optimization in the context of compiling code refers to the process by which a compiler attempts to improve the efficiency and performance of the resulting binary without altering its functionality.

The Intel C++ Compiler icc, is a highly optimized C and C++ compiler that offers advanced optimization capabilities, targeting Intel architectures. It is designed to maximize performance on Intel processors by taking advantage of the unique features and instruction sets of these CPUs.

When compiling code with icc, various optimization flags can be used to control the level and type of optimizations applied.

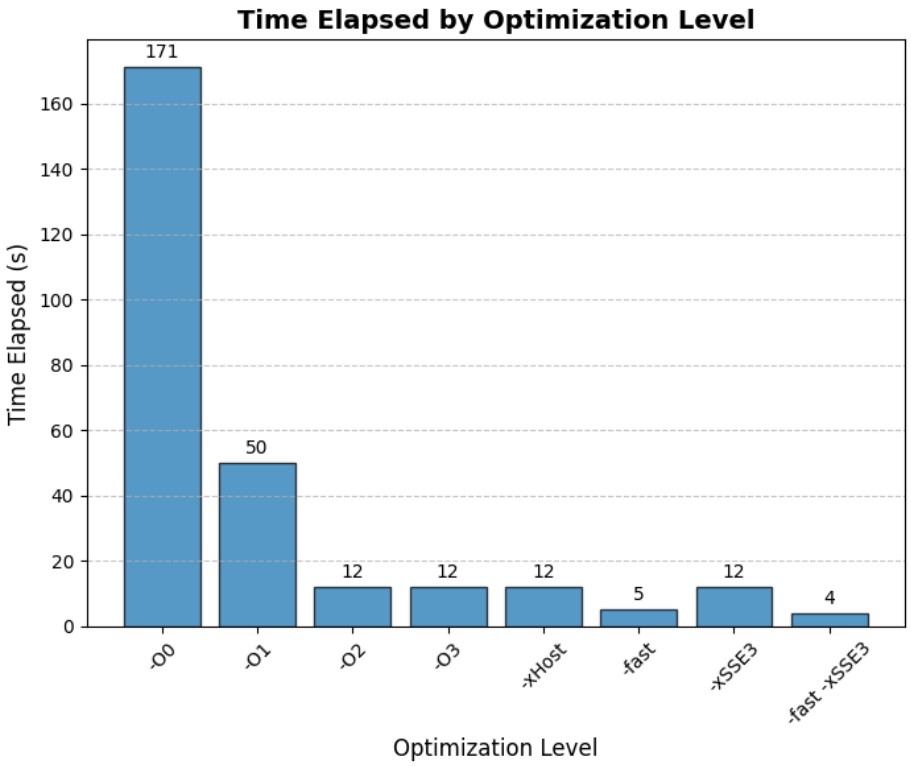
Here is a closer look at the optimization flags mentioned and their significance:

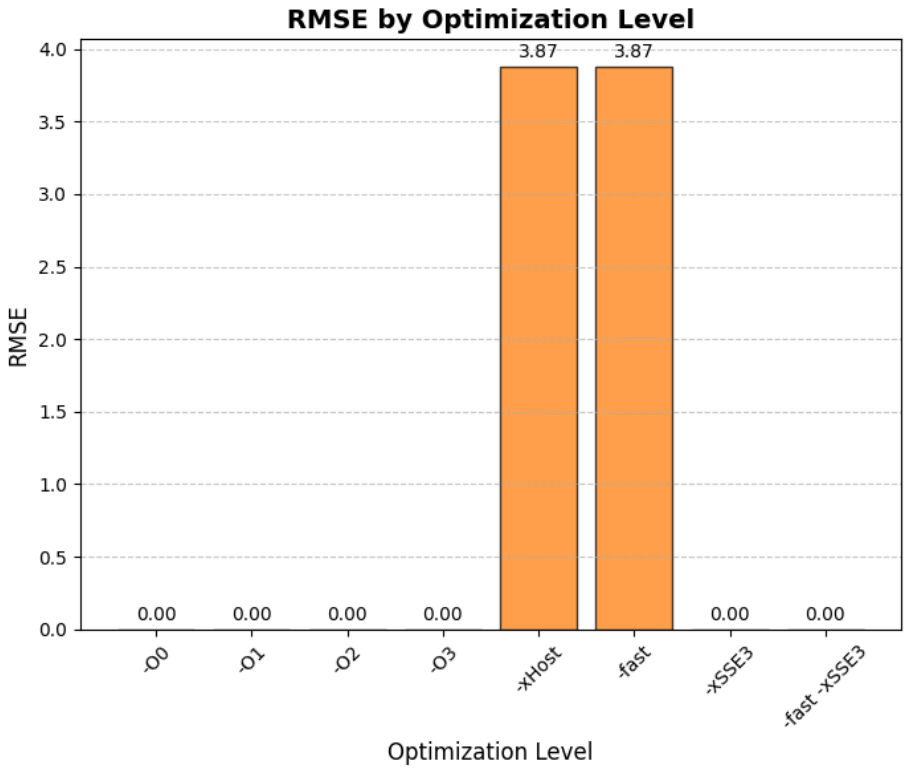
- `O0`: This level turns off most optimizations, resulting in a faster compilation time but potentially slower execution time.
- `O1`, `O2`, `O3`: These flags represent increasing levels of optimization. `O1` provides a basic optimization balance without significantly increasing compile time. `O2` introduces a broader set of optimizations that may slightly lengthen compilation but can lead to better performance. `O3` applies aggressive optimizations, including those that may substantially increase compilation time but aim to achieve the maximum execution speed. Each level builds upon the previous.
- **`xHost`**: This flag enables optimizations specific to the architecture of the machine on which the code is compiled. It allows the compiler to generate code that utilizes the most advanced instruction sets available.
- **`fast`**: This combines several optimization strategies to enhance performance aggressively. It generally includes `O3` alongside other optimizations that target quick execution.
- **`xSSE3`**: Specifies the compiler to use the SSE3 instruction set for optimizations. SSE3 instructions are designed to improve performance in certain computational tasks, such as those involving floating-point operations.

- **Combining flags** like `fast` with `xSSE3` tailors the optimization process further, aiming to extract the best possible performance for specific processor capabilities.

After running The previous optimization flags on Lab PC we got the following results:

Flag	Resolution & Iterations	Time Elapsed	RMSE
-O0	1000	171	0
-O1	1000	50	0
-O2	1000	12	0
-O3	1000	12	0
-xHost	1000	12	3.87298
-fast	1000	5	3.87298
-xSSE3	1000	12	0





Analysis

- Optimization Levels -O0 to -O3:
    - As optimization levels increase from -O0 to -O3, we observe a significant reduction in time elapsed. The highest optimization level without architecture-specific flags (-O3) provides the same performance in terms of time as -O2, indicating that the additional optimizations in -O3 may not contribute further to this program's performance.

    - The RMSE remains at 0 for these optimization levels, indicating that the optimizations have not affected the accuracy of the generated image compared to the reference image ”computed without any optimization flag”.

- Architecture-Specific Optimization -xHost:
  Using -xHost does not improve the time performance compared to -O2 and -O3, which suggests that the general optimizations already maximize performance for this specific host architecture. However, it has an RMSE of 3.87298, which indicates a deviation from the expected output, possibly due to the use of different instruction sets or aggressive optimizations that affect floating-point calculations.

- Optimization Level -fast:

The -fast optimization significantly reduces the time to only 5 seconds. However, it also results in an RMSE of 3.87298, indicating a loss in accuracy.

- Architecture-Specific Optimization -xSSE3:
  The time elapsed with -xSSE3 is the same as with -O2 and -O3, with no loss in accuracy (RMSE of 0).

- Combination of -fast and -xSSE3:
  This combination yields the best time performance of all the tested flags, reducing the execution time to 4 seconds, and does not impact the accuracy (RMSE is 0). It indicates that this mix of fast optimizations and SSE3 instructions is highly effective for this program.

Although the combination of *-fast* and *-xSSE3* is the best regarding the execution time and accuracy, we will use *-O2* for calculating the sequential time baseline for different resolution and iteration combinations since we will use the *INFN servers* for MPI where *-fast* flag is not supported, and to make our analysis agnostic from the device architecture we execute our code on.

Sequential time baseline / -O2 & different configurations

Resolution	Iterations	Time Elapsed in sec
1000	1000	12
1000	2000	25
1000	3000	38
1000	4000	50
1000	5000	63
2000	1000	52
2000	2000	102
2000	4000	202
2000	5000	254
3000	1000	116
3000	2000	231

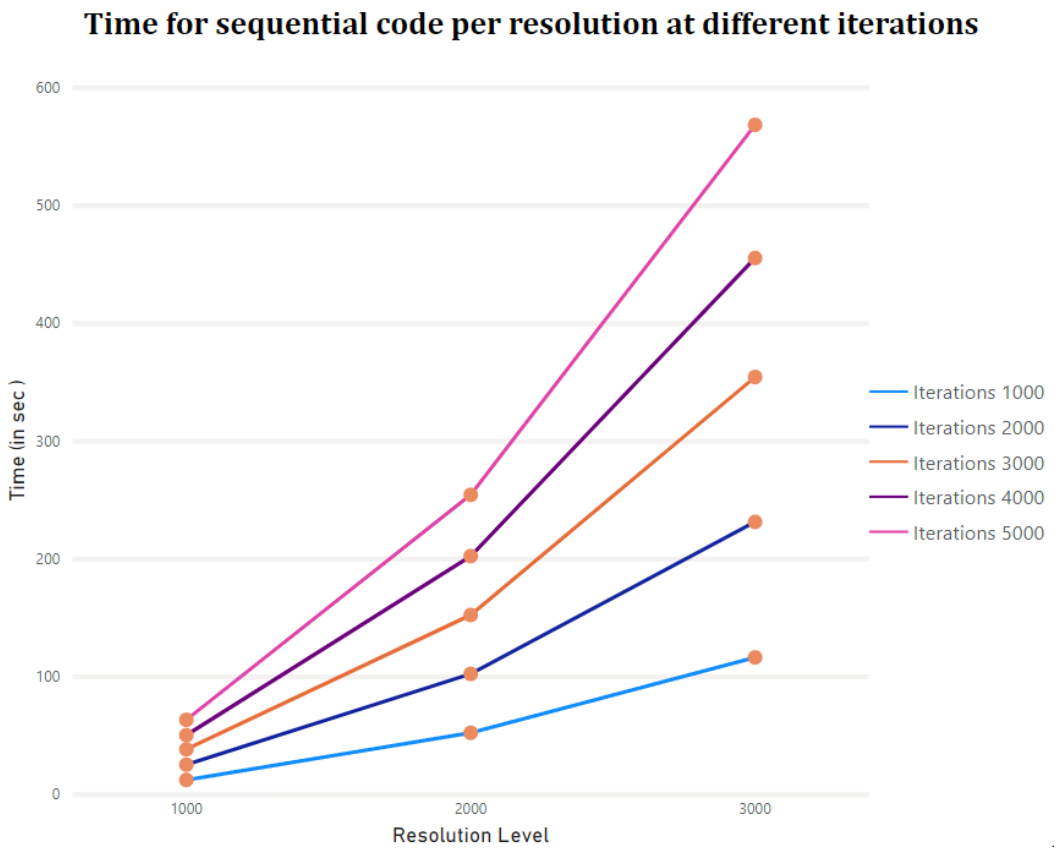
3000	3000	354
3000	4000	455
3000	5000	568

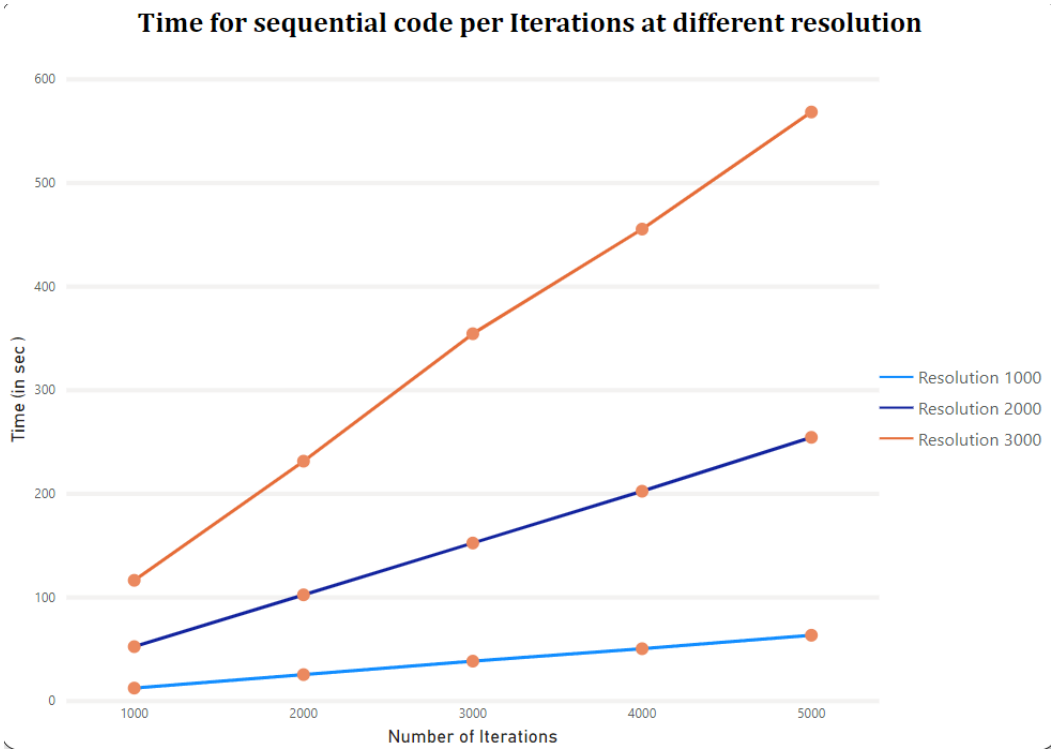
Images generated by these configurations will be used as reference images to calculate RMSE later in different parallelization methods.

The relation between image resolution and time elapsed is *quadratic* because increasing resolution affects the amount of pixels in the two dimensions causing a quadratic increase in the number of pixels to compute.

The relation between iterations and time elapsed is almost linear, actually it is not easy to interpret, but linearity here means that the number of pixels that get out of the bound with increasing the iterations are few, at least with the chosen iterations, so by increasing the iterations time complexity increases linearly.

The plots below show the *quadratic* and *linear* relations mentioned above.





OpenMP

OpenMP can be used to parallelize the computation and take advantage of multiple threads available on the machine. Here we will parallelize the computation of image pixels(pixels computation loop) where each row of pixels will be computed separately.

The number of threads is a parameter we pass before running the executable and set using *opm_set_num_threads* function. Threads in OpenMP share the same memory and OpenMP will handle threads duty distribution.

```
omp_set_num_threads(nThreads);
std::cout<<"Threads Requested: "<<nThreads<<std::endl;


#pragma omp parallel shared(image) for schedule(dynamic)
for (int row = 0; row < HEIGHT; row++)
{
  for (int col = 0; col < WIDTH; col++)
    {
       int pos = row * WIDTH + col;
       if(pos == 0)
         std::cout<<"Threads
Reserved:"<<omp_get_num_threads()<<std::endl;

        image[pos] = 0;
        const complex<double> c(col * STEP + MIN_X, row * STEP +
```

```
MIN_Y);

          // z = z^2 + c
          complex<double> z(0, 0);
          for (int i = 1; i <= ITERATIONS; i++)
          {
              z = pow(z, 2) + c;

              // If it is convergent
              if (abs(z) >= 2)
                {
                  image[pos] = i;
                  break;
                 }
           }
         }
    }
```

In OpenMP, we can define different scheduling criteria(static, dynamic and guided) for parallelizing "for" loops. We used the three scheduling criteria with different resolution and iteration configurations to compare the *time elapsed*, *speed up* and *thread efficiency*.

Each scheduling criteria has it is own specs that characterize it from others, that could fit the hot spot/problem nature or not.

$$S(Speedup) = \frac{Ts(Time\ Sequential)}{Tp(Time\ Parallel)}$$

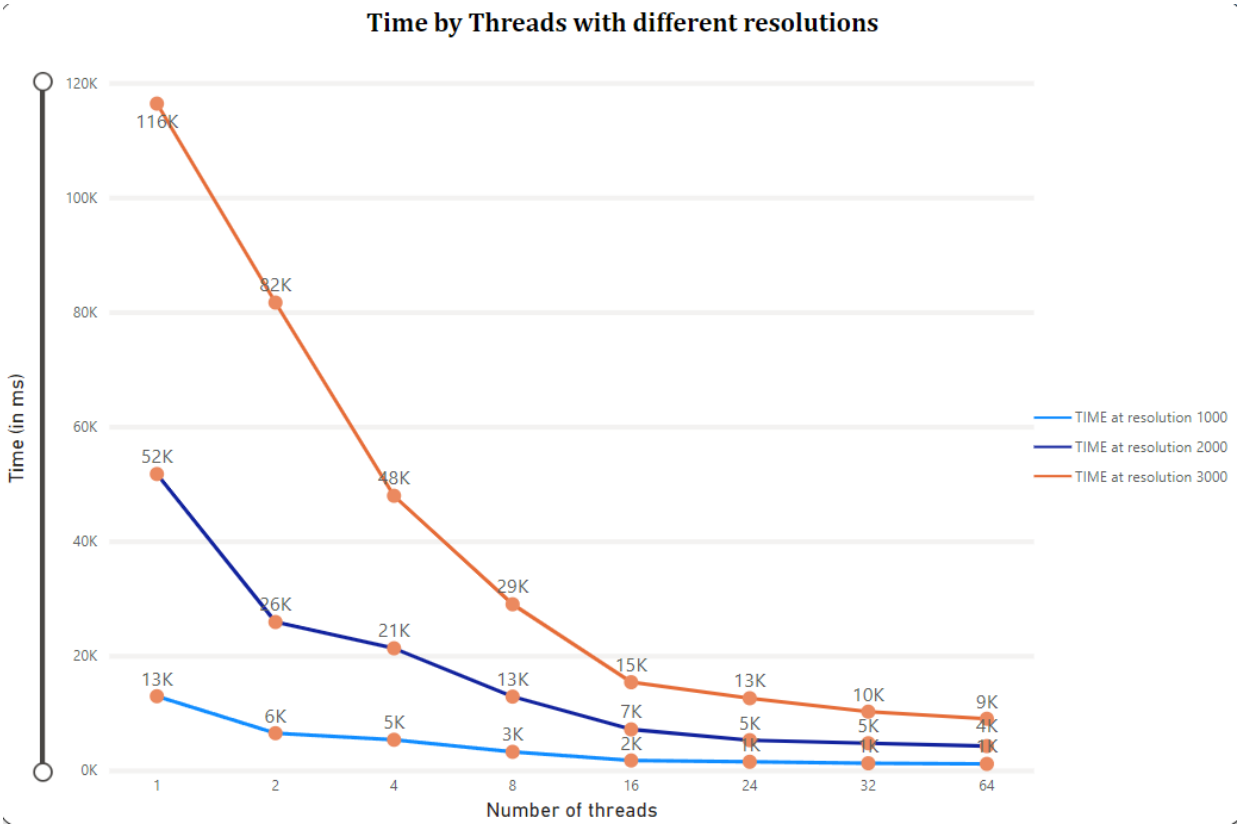
$$E(Efficiency) = \frac{S}{N(Number\ of\ threads)}$$

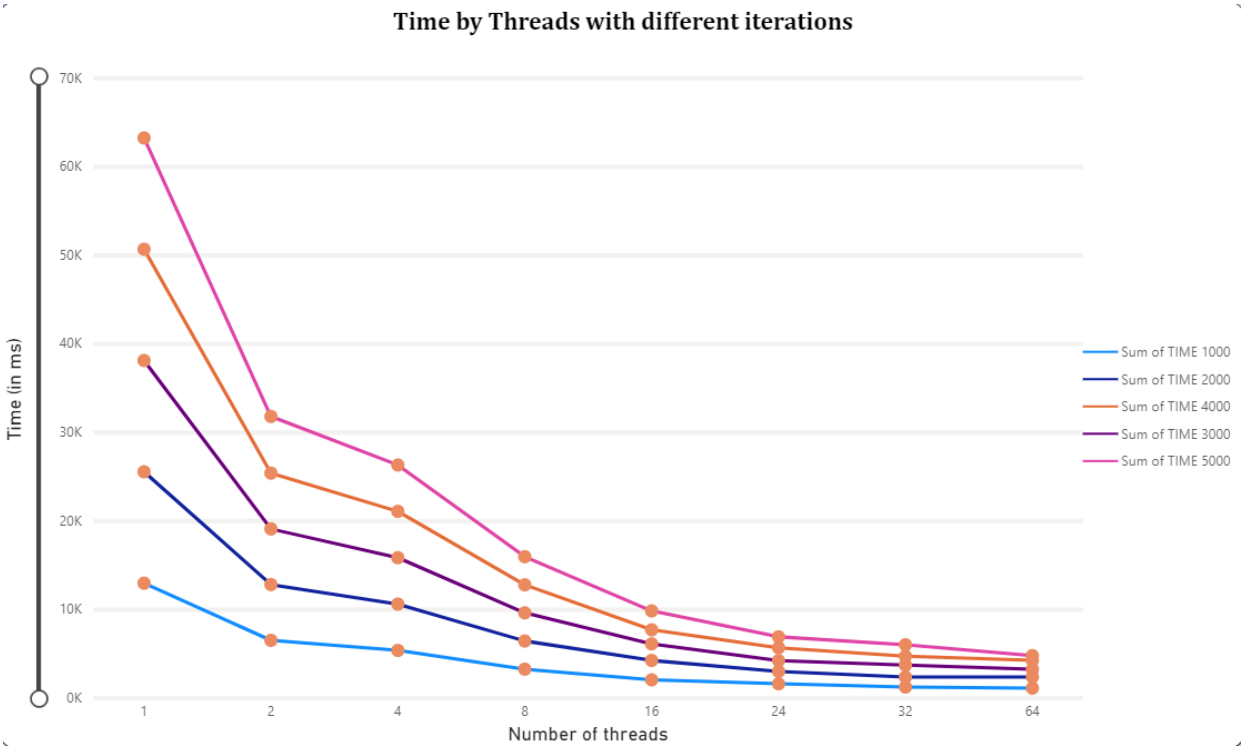
Static Scheduling Results

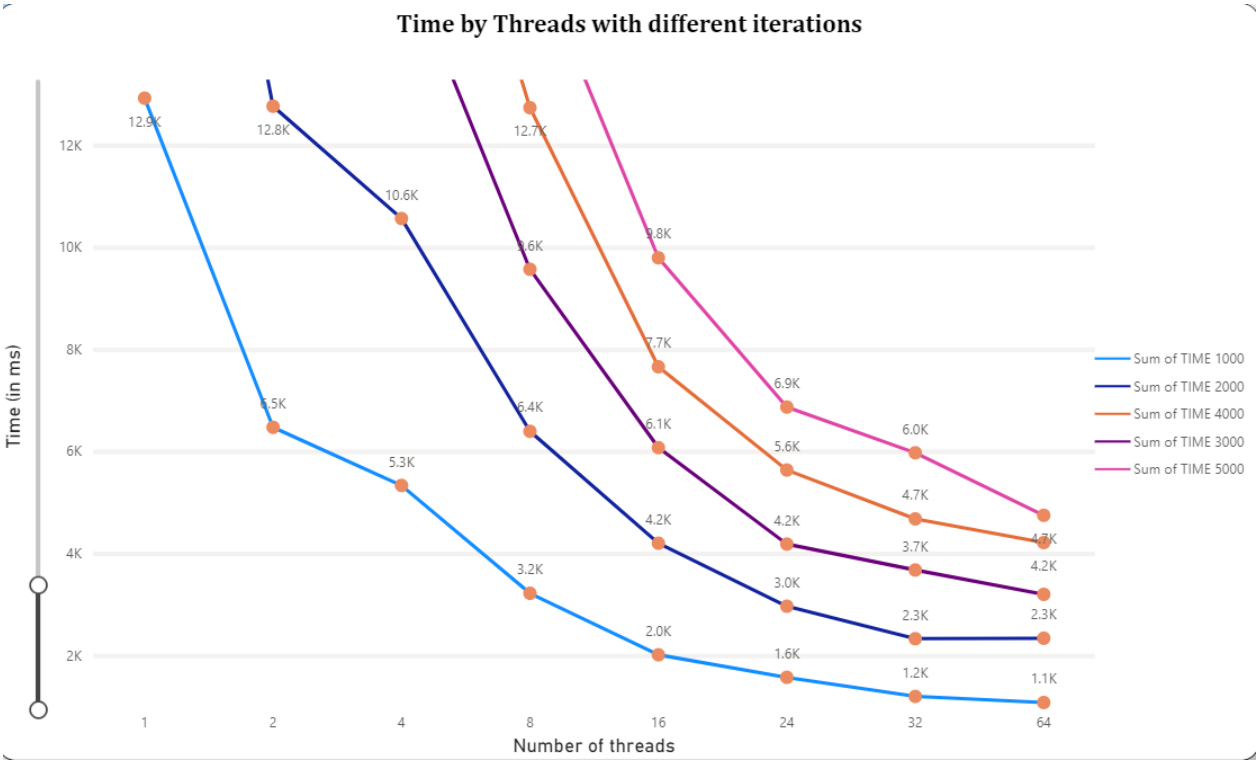
#	RES 1000 – ITR 1000			RES 2000 – ITR 1000			RES 3000 – ITR 1000		
N	T/MS	S	E	T/MS	S	E	T/MS	S	E
64	1116	10.75	0.17	4227	12.30	0.19	8994	12.90	0.20
32	1227	9.78	0.31	4719	11.02	0.34	10232	11.34	0.35
24	1486	8.08	0.34	5257	9.89	0.41	12577	9.22	0.38
16	1713	7.01	0.44	7136	7.29	0.46	15372	7.55	0.47
8	3218	3.73	0.47	12858	4.04	0.51	28979	4.00	0.50

4	5327	2.25	0.56	21295	2.44	0.61	47921	2.42	0.61
2	6474	1.85	0.93	25881	2.01	1.00	81643	1.42	0.71
1	12927	0.93	0.93	51716	1.01	1.01	116378	1.00	1.00

#	ITR 1000 – RES 1000			ITR 3000 – RES 1000			ITR 5000 – RES 1000		
N	T/MS	S	E	T/MS	S	E	T/MS	S	E
64	1116	10.75	0.17	3196	11.89	0.19	4745	13.28	0.21
32	1227	9.78	0.31	3673	10.35	0.32	5971	10.55	0.33
24	1486	8.08	0.34	4181	9.09	0.38	6867	9.17	0.38
16	1713	7.01	0.44	6072	6.26	0.39	9793	6.43	0.40
8	3218	3.73	0.47	9568	3.97	0.50	15915	3.96	0.49
4	5327	2.25	0.56	15799	2.41	0.60	26261	2.40	0.60
2	6474	1.85	0.93	19047	2.00	1.00	31745	1.98	0.99
1	12927	0.93	0.93	38062	1.00	1.00	63185	1.00	1.00







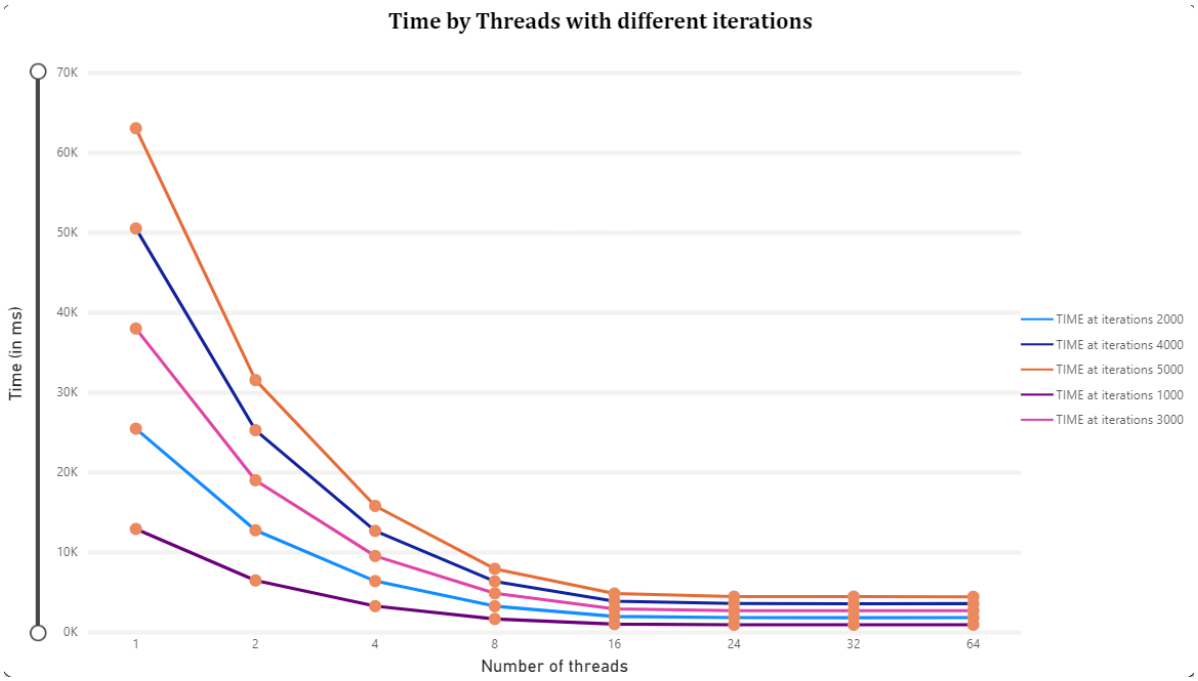
We can see how increasing the thread number enhances the execution time *gradually* with different resolution & iteration configurations.

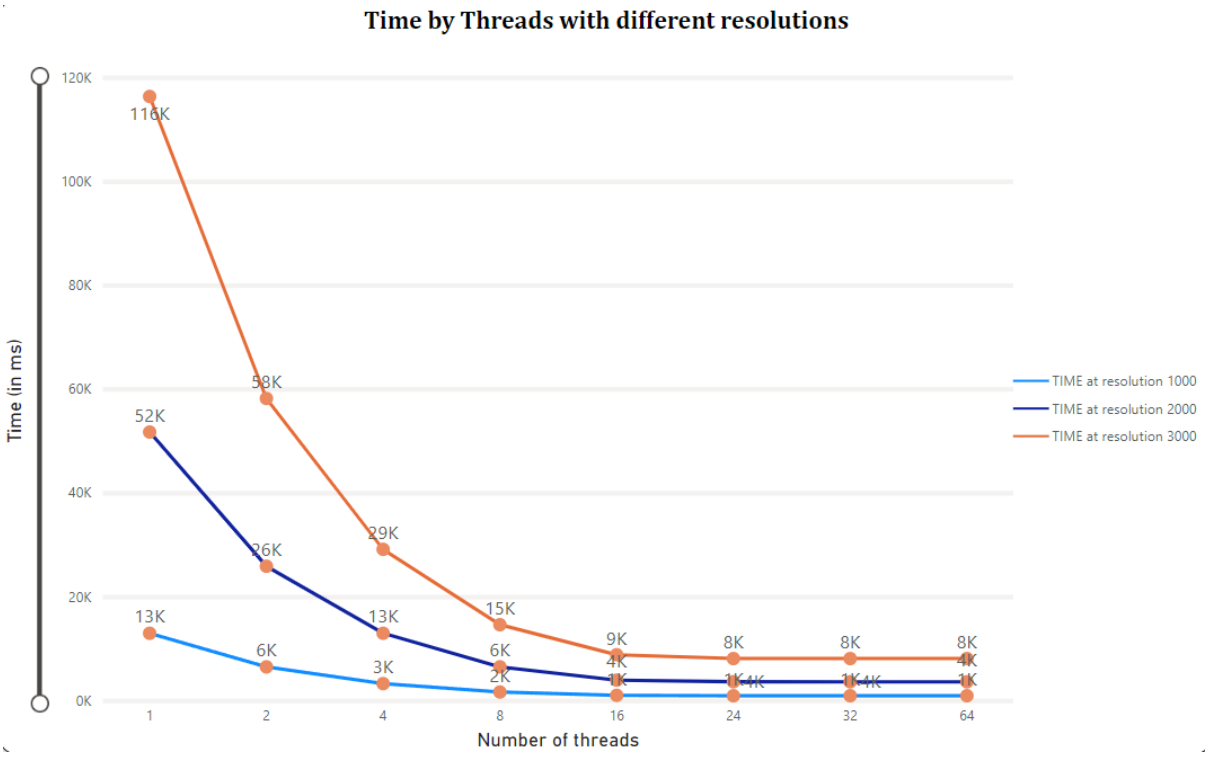
Another observation, the time elapsed still decreases even with a number of threads greater than the amount of the threads in the host machine.

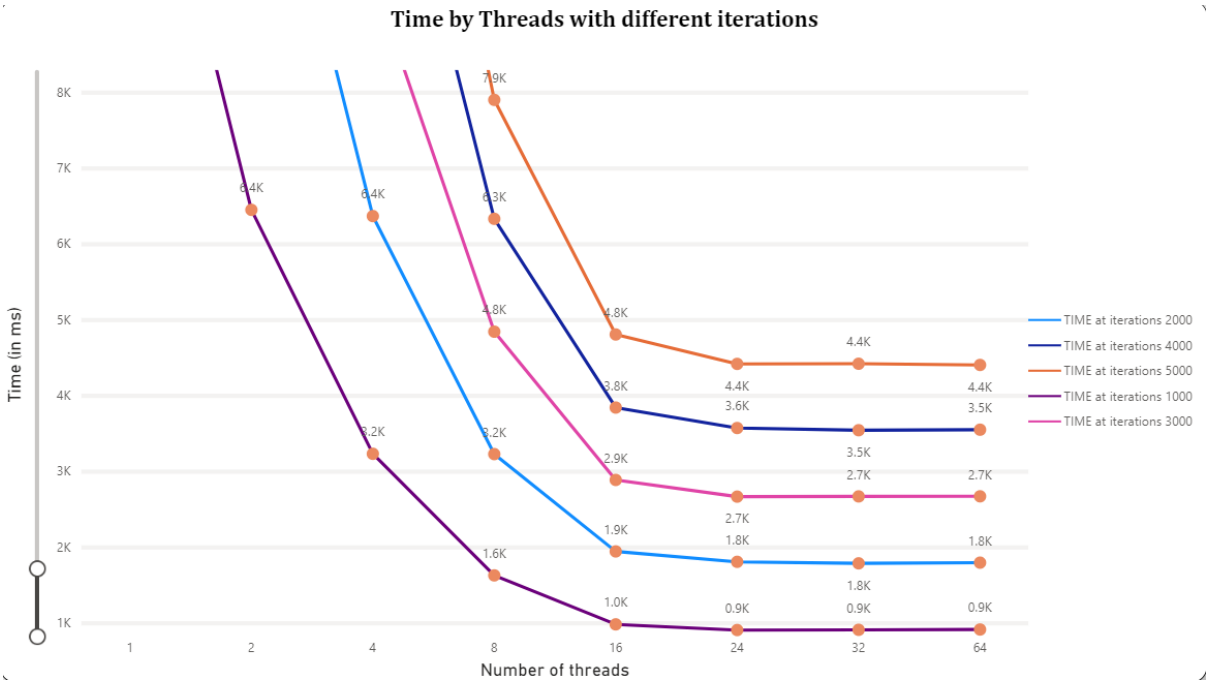
Dynamic Scheduling Results

#	RES 1000 – ITR 1000			RES 2000 – ITR 1000			RES 3000 – ITR 1000		
N	T/MS	S	E	T/MS	S	E	T/MS	S	E
64	909	13.20	0.21	3596	14.46	0.23	8070	14.37	0.22
32	908	13.22	0.41	3595	14.46	0.45	8060	14.39	0.45
24	902	13.30	0.55	3605	14.42	0.60	8062	14.39	0.60
16	982	12.22	0.76	3909	13.30	0.83	8800	13.18	0.82
8	1622	7.40	0.92	6470	8.04	1.00	14580	7.96	0.99
4	3237	3.71	0.93	12933	4.02	1.01	29097	3.99	1.00
2	6462	1.86	0.93	25843	2.01	1.01	58151	1.99	1.00
1	12929	0.93	0.93	51682	1.01	1.01	116308	1.00	1.00

#	ITR 1000 – RES 1000			ITR 3000 – RES 1000			ITR 5000 – RES 1000		
N	T/MS	S	E	T/MS	S	E	T/MS	S	E
64	909	13.20	0.21	2668	14.24	0.22	4401	14.31	0.22
32	908	13.22	0.41	2667	14.25	0.45	4417	14.26	0.45
24	902	13.30	0.55	2664	14.26	0.59	4415	14.27	0.59
16	982	12.22	0.77	2884	13.18	0.82	4803	13.12	0.82
8	1622	7.40	0.92	4840	7.85	0.98	7898	7.98	1.00
4	3237	3.71	0.93	9503	4.00	1.00	15771	3.99	1.00
2	6462	1.86	0.93	18981	2.00	1.00	31514	2.00	1.00
1	12929	0.93	0.93	37967	1.00	1.00	63029	1.00	1.00


Time by Threads with different iterations
Time (in ms)
70K
60K
50K
40K
30K
20K
10K
0K
1
2
4
8
16
24
32
64
Number of threads
TIME at iterations 2000
TIME at iterations 4000
TIME at iterations 5000
TIME at iterations 1000
TIME at iterations 3000


Time by Threads with different resolutions
Time (in ms)
120K
100K
80K
60K
40K
20K
0K
116K
58K
52K
29K
26K
15K
13K
13K
9K
8K
8K
8K
6K
6K
4K
4K
4K
4K
3K
2K
1K
1K
1K
1K
1
2
4
8
16
24
32
64
Number of threads
TIME at resolution 1000
TIME at resolution 2000
TIME at resolution 3000



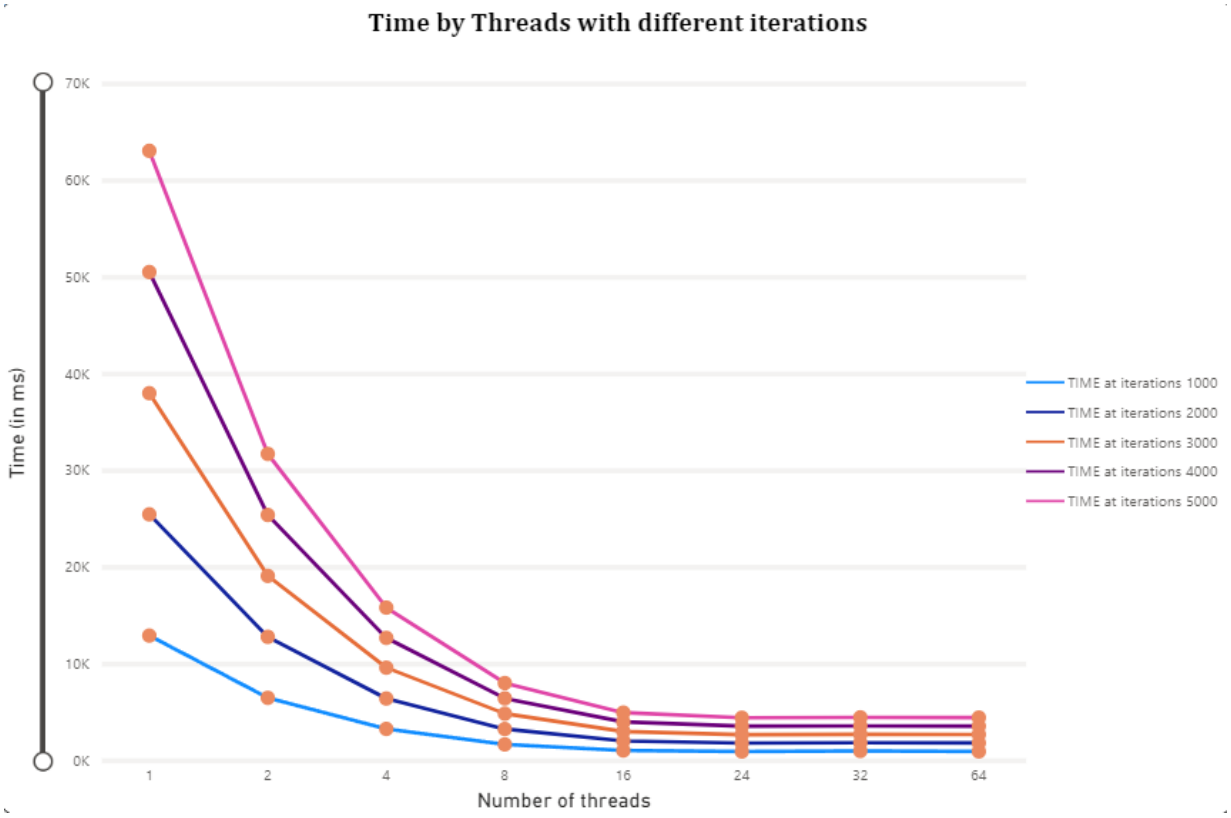
We can see that the time decreases as we increase the threads number in a nearly linear way, till reaching the maximum count of threads in the host machine, which is /24/ threads.

After reaching the maximum /24/ we can observe a plateau in the time elapsed.

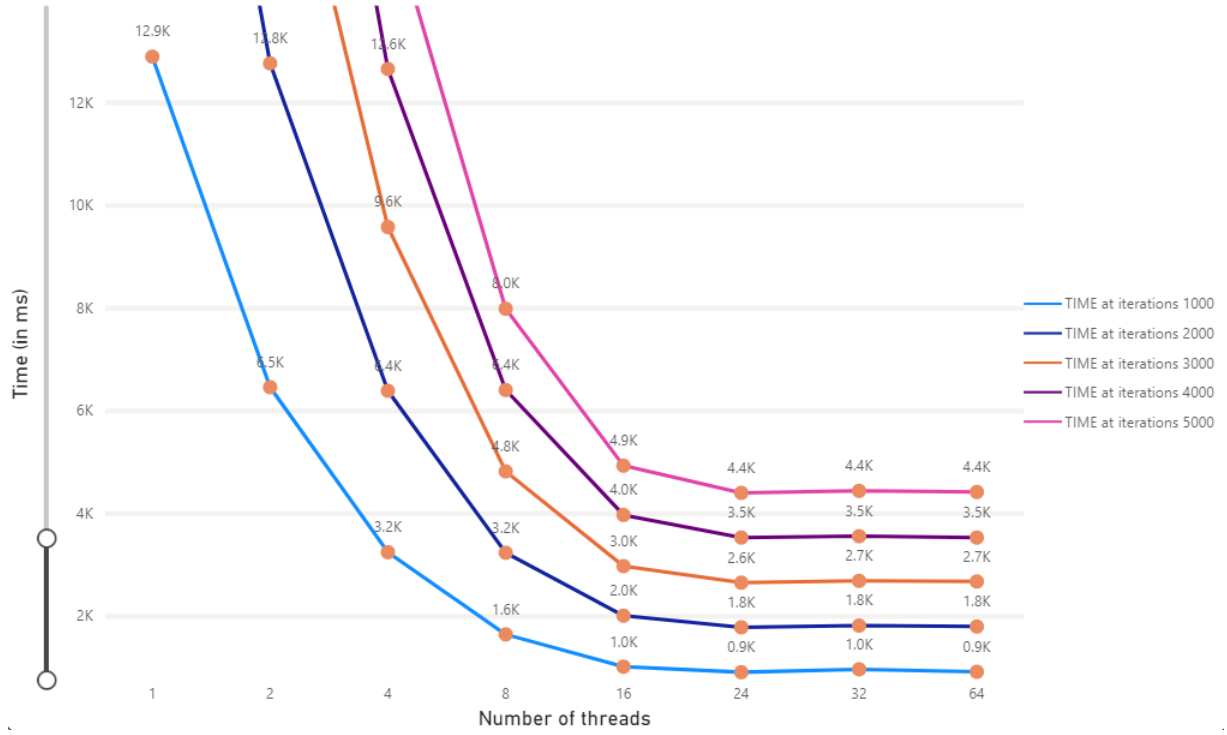
Guided Scheduling Results

#	RES 1000 – ITR 1000			RES 2000 – ITR 1000			RES 3000 – ITR 1000		
Thd	T/MS	S	E	T/MS	S	E	T/MS	S	E
64	908	13.22	0.21	3602	14.44	0.23	8071	14.37	0.22
32	911	13.17	0.41	3623	14.35	0.45	8152	14.23	0.44
24	903	13.29	0.55	3586	14.50	0.60	8046	14.42	0.60
16	1000	12.00	0.75	4011	12.96	0.81	9010	12.87	0.80
8	1638	7.33	0.92	6560	7.93	0.99	14750	7.86	0.98
4	3246	3.70	0.92	12967	4.01	1.00	29165	3.98	0.99
2	6472	1.85	0.93	25861	2.01	1.01	58209	1.99	1.00
1	12934	0.93	0.93	51686	1.01	1.01	116298	1.00	1.00

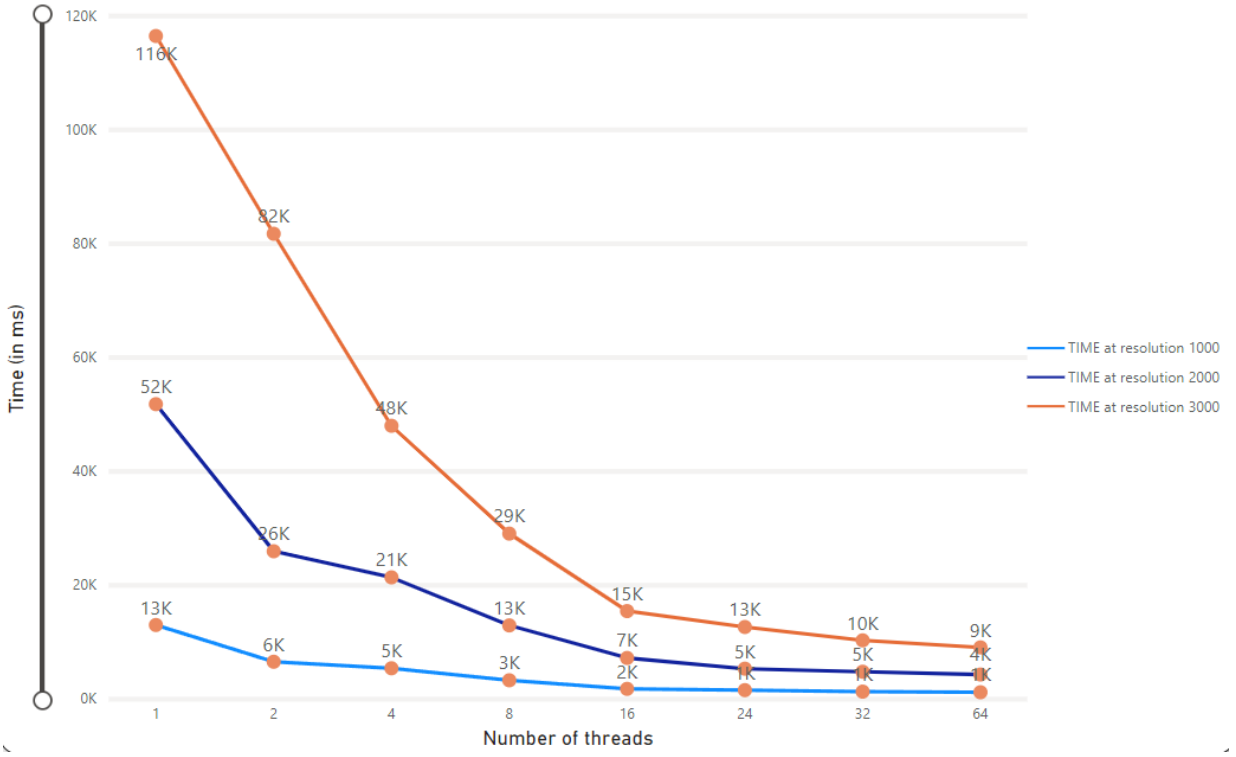
#	ITR 1000 – RES 1000			ITR 3000 – RES 1000			ITR 5000 – RES 1000		
Thd	T/MS	S	E	T/MS	S	E	T/MS	S	E
64	908	13.22	0.21	2668	14.25	0.22	4401	14.29	0.22
32	911	13.17	0.39	2667	14.19	0.44	4417	14.22	0.44
24	903	13.29	0.56	2664	14.38	0.60	4415	14.34	0.60
16	1000	12.00	0.75	2884	12.82	0.80	4803	12.80	0.80
8	1638	7.33	0.92	4840	7.90	0.99	7898	7.90	0.99
4	3246	3.70	0.93	9503	3.97	0.99	15771	3.99	1.00
2	6472	1.85	0.93	18981	1.99	1.00	31514	1.99	0.99
1	12934	0.93	0.93	37967	1.00	1.00	63029	1.00	1.00



Time by Threads with different iterations



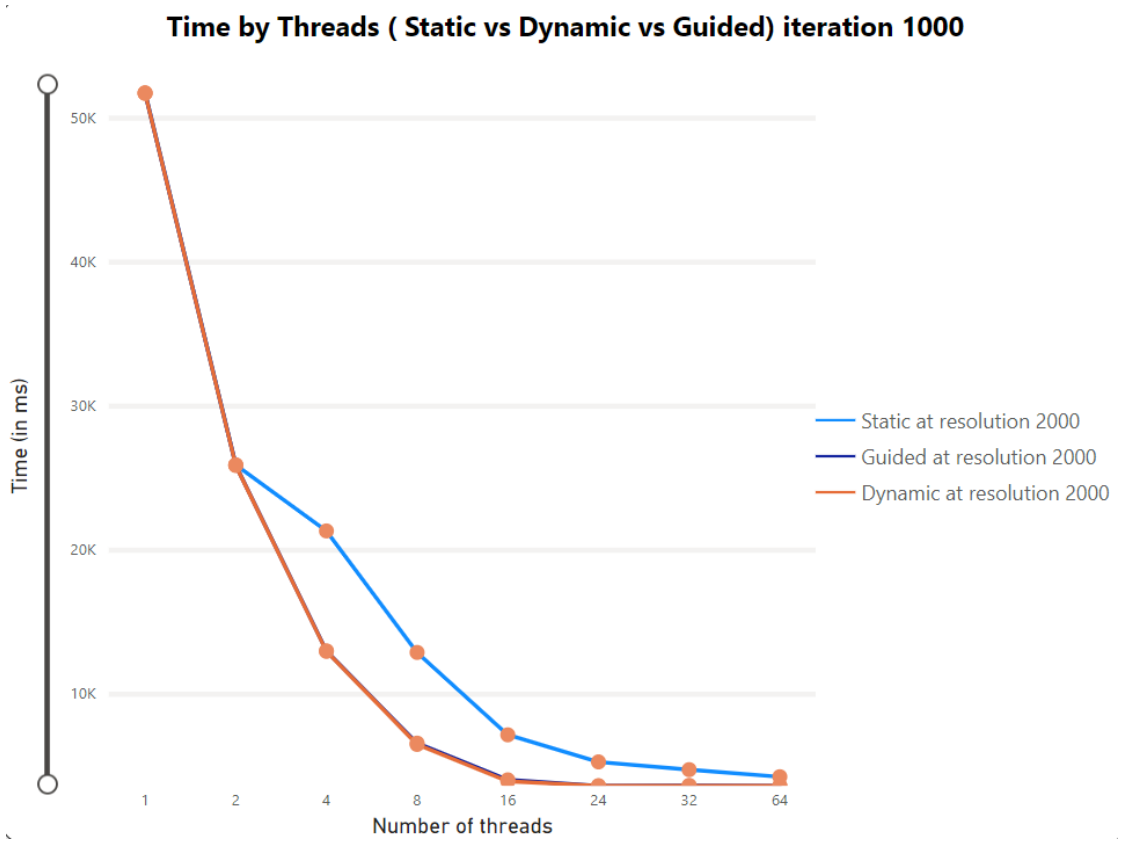
Time by Threads with different resolutions



We can see that the time decreases as we increase the threads number in a nearly linear way, till reaching the maximum count of threads in the host machine, which is /24/ threads.

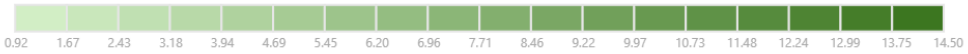
After reaching the maximum /24/ we can observe a plateau in the time elapsed.

Static vs Dynamic vs Guided



Speedup Analysis with OpenMp Dynamic, Static and Guided

	1	2	4	8	16	24	32	64
Res 1k - ITR 1k Dynaminc	0.92	1.85	3.7	7.39	12.21	13.3	13.21	13.2
Res 2k - ITR 1k Dynaminc	1	2.01	4.02	8.03	13.3	14.42	14.46	14.46
Res 3k - ITR 1k Dynaminc	0.99	1.99	3.98	7.95	13.18	14.38	14.39	14.37
Res 1k - ITR 2k Dynaminc	0.98	1.96	3.93	7.75	12.88	13.86	14.01	13.94
Res 1k - ITR 3k Dynaminc	1	2	4	7.85	13.18	14.26	14.25	14.24
Res 1k - ITR 4k Dynaminc	0.99	1.98	3.95	7.9	13.02	14.01	14.12	14.1
Res 1k - ITR 5k Dynaminc	1	2	3.99	7.98	13.12	14.27	14.26	14.31
Res 1k - ITR 1k Static	0.93	1.85	2.25	3.73	7.01	8.08	9.78	10.75
Res 2k - ITR 1k Static	1.01	2.01	2.44	4.04	7.29	9.89	11.02	12.3
Res 3k - ITR 1k Static	1	1.42	2.42	4	7.55	9.22	11.34	12.9
Res 1k - ITR 2k Static	0.98	1.96	2.37	3.91	5.96	8.44	10.74	10.7
Res 1k - ITR 3k Static	1	2	2.41	3.97	6.26	9.09	10.35	11.89
Res 1k - ITR 4k Static	0.99	1.97	2.38	3.93	6.53	8.88	10.7	11.87
Res 1k - ITR 5k Static	1	1.98	2.4	3.96	6.43	9.17	10.55	13.28
Res 1k - ITR 1k Guided	0.93	1.85	3.7	7.33	12	13.29	13.17	13.22
Res 2k - ITR 1k Guided	1.01	2.01	4.01	7.93	12.96	14.5	14.35	14.44
Res 3k - ITR 1k Guided	1	1.99	3.98	7.86	12.87	14.42	14.23	14.37
Res 1k - ITR 2k Guided	0.98	1.96	3.92	7.75	12.5	14.1	13.84	13.98
Res 1k - ITR 3k Guided	1	1.99	3.97	7.9	12.82	14.38	14.19	14.25
Res 1k - ITR 4k Guided	0.99	1.97	3.95	7.82	12.62	14.2	14.09	14.2
Res 1k - ITR 5k Guided	1	1.99	3.99	7.9	12.8	14.34	14.22	14.29



Efficiency Analysis with OpenMp Dynamic, Static and Guided

	1	2	4	8	16	24	32	64
Res 1k - ITR 1k Dynaminc	0.93	0.93	0.93	0.92	0.76	0.55	0.41	0.21
Res 2k - ITR 1k Dynaminc	1.01	1.01	1.01	1	0.83	0.6	0.45	0.23
Res 3k - ITR 1k Dynaminc	1	1	1	0.99	0.82	0.6	0.45	0.22
Res 1k - ITR 2k Dynaminc	0.98	0.98	0.98	0.97	0.8	0.58	0.44	0.22
Res 1k - ITR 3k Dynaminc	1	1	1	0.98	0.82	0.59	0.45	0.22
Res 1k - ITR 4k Dynaminc	0.99	0.99	0.99	0.99	0.81	0.58	0.44	0.22
Res 1k - ITR 5k Dynaminc	1	1	1	1	0.82	0.59	0.45	0.22
Res 1k - ITR 1k Static	0.93	0.93	0.56	0.47	0.44	0.34	0.31	0.17
Res 2k - ITR 1k Static	1.01	1	0.61	0.51	0.46	0.41	0.34	0.19
Res 3k - ITR 1k Static	1	0.71	0.61	0.5	0.47	0.38	0.35	0.2
Res 1k - ITR 2k Static	0.98	0.98	0.59	0.49	0.37	0.35	0.34	0.17
Res 1k - ITR 3k Static	1	1	0.6	0.5	0.39	0.38	0.32	0.19
Res 1k - ITR 4k Static	0.99	0.99	0.59	0.49	0.41	0.37	0.33	0.19
Res 1k - ITR 5k Static	1	0.99	0.6	0.49	0.4	0.38	0.33	0.21
Res 1k - ITR 1k Guided	0.93	0.93	0.92	0.92	0.75	0.55	0.41	0.21
Res 2k - ITR 1k Guided	1.01	1.01	1	0.99	0.81	0.6	0.45	0.23
Res 3k - ITR 1k Guided	1	1	0.99	0.98	0.8	0.6	0.44	0.22
Res 1k - ITR 2k Guided	0.98	0.98	0.98	0.97	0.78	0.59	0.43	0.22
Res 1k - ITR 3k Guided	1	1	0.99	0.99	0.8	0.6	0.44	0.22
Res 1k - ITR 4k Guided	0.99	0.99	0.99	0.98	0.79	0.59	0.44	0.22
Res 1k - ITR 5k Guided	1	0.99	1	0.99	0.8	0.6	0.44	0.22

After reading the visualizations we can analyze the following:

Static results are the worst in comparison to dynamic and guided scheduling. The iterations workload is not balanced, since each pixel requires a different number of iterations affecting the execution

time. So static scheduling is not the best way to distribute the workload between threads since distribution is decided at the compile time rather than the runtime.

Guided scheduling is slightly better than dynamic scheduling since dynamic scheduling introduces a larger overhead during the runtime than guided scheduling.

After employing /24/ threads in dynamic and guided scheduling, the execution time is not enhanced since the lab's machine has just /24/ threads and the task nature is a CPU-intensive task that will not benefit from virtual threads that we get after the first /24/ physical threads.

The speedup in our case is not linear, due to the scheduling and parallelization overhead impact, in addition to the resource limitation(number of physical threads). We can see how the change in speedup and threads efficiency is, in specific after employing more than /8/ threads, which reaches /0.55/ efficiency and nearly /13.3/ speedup while employing /24/ threads in the case of dynamic and guided scheduling, hence it was *linear* while employing less or equal than /8/ threads and become *sub-linear* after that, till reaching /24/ threads to get a *plateau* after that.

It is worth noting that OpenMP did not introduce any RMSE with all different usages.

CUDA

In Cuda, we developed a kernel to be executed by each GPU thread, where each thread handles the computation for a single pixel:

```
__global__ void calcPxl(int* image, int width, int height, double
step, int min_x, int min_y, int iterations){
    const int pos = threadIdx.x + blockIdx.x * blockDim.x;

        if (pos < width * height){

            image[pos] = 0;

            const int row = pos / width;
            const int col = pos % width;
            const complex<double> c(col * step + min_x, row * step +
min_y);

            // z = z^2 + c
            complex<double> z(0, 0);
            for (int i = 1; i <= iterations; i++)
            {
                z = pow(z, 2) + c;

                // If it is convergent
                if (abs(z) >= 2)
                {
                    image[pos] = i;
```

```
                    break;
                }
            }
        }
    return;
    }
```

In the Host code, we used a 1D Grid of blocks with a 1D Block of threads, it is worth mentioning that we need to allocate data in the GPU before passing the development image pointer to the executed kernel since the Host(CPU) and Device(GPU) do not share the same memory. For the same reason, we need to copy the development image back from the Device to the Host once all threads finish. Memory copy force synchronization, so there is no need to call any synchronization instruction.

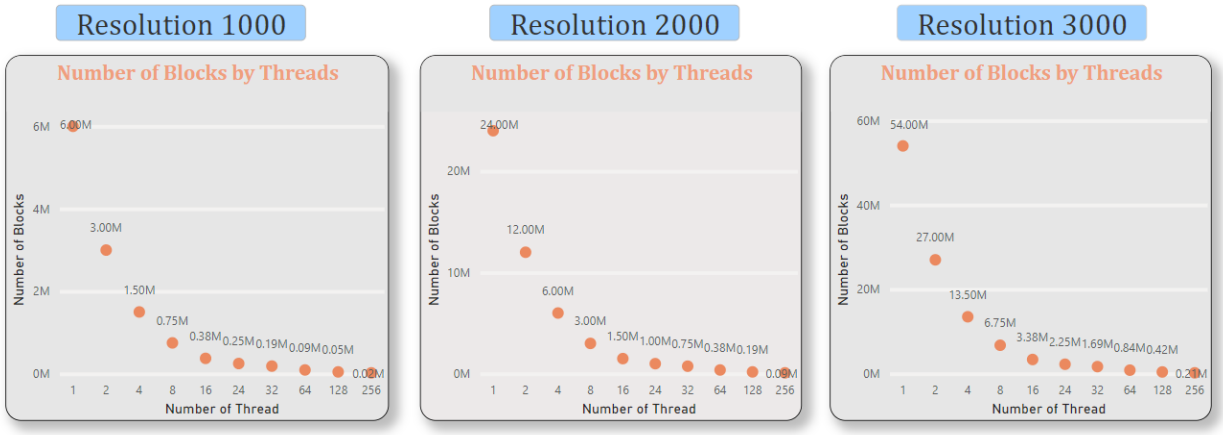
```
cudaMalloc( (void**)&dev_image, N * sizeof(int) );
    cudaMemcpy(dev_image, image, N * sizeof(int),
cudaMemcpyHostToDevice);
    dim3 threads(nThreads);
    dim3 blocks ( (N+threads.x-1)/threads.x );
    cout<<"Threads: "<<threads.x<<" Blocks: "<<blocks.x<<endl;

    calcPxl<<<blocks,threads>>>(dev_image, WIDTH, HEIGHT, STEP, MIN_X,
MIN_Y, ITERATIONS);


    cudaMemcpy(image, dev_image, N * sizeof(int),
cudaMemcpyDeviceToHost);
```

We tried different threads per block configurations with different collections of resolution iterations, to find the best threads per block configuration that maximize the speedup and minimize the time elapsed.

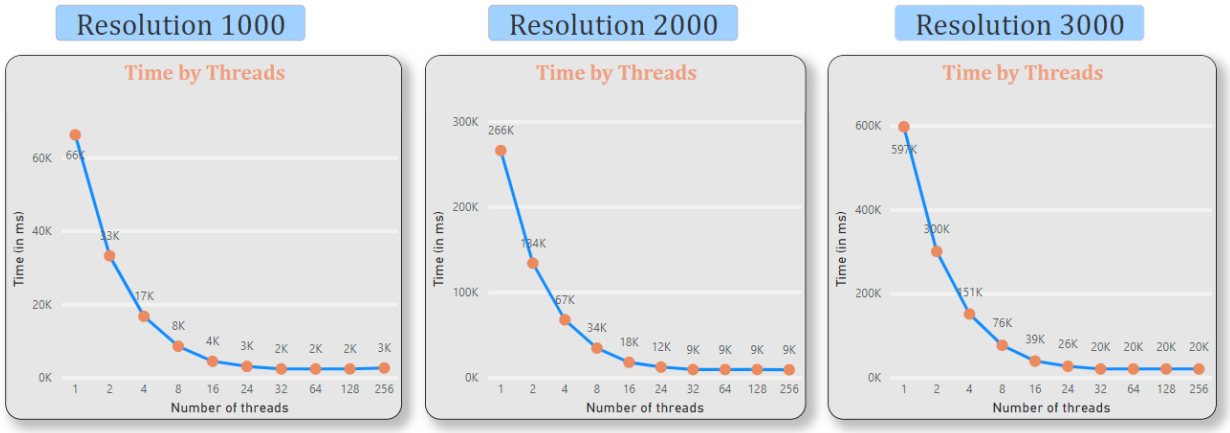
Threads per Block	RES-1000 & ITR-1000		RES-2000 & ITR-1000		RES-3000 & ITR-1000	
N	T(MS)	S	T(MS)	S	T(MS)	S
256	2547	4.71	8835	5.89	19838	5.85
128	2290	5.24	9102	5.71	19862	5.84
64	2289	5.24	9084	5.72	19877	5.84
32	2286	5.25	9091	5.72	19836	5.85
24	2980	4.03	11950	4.35	26171	4.43

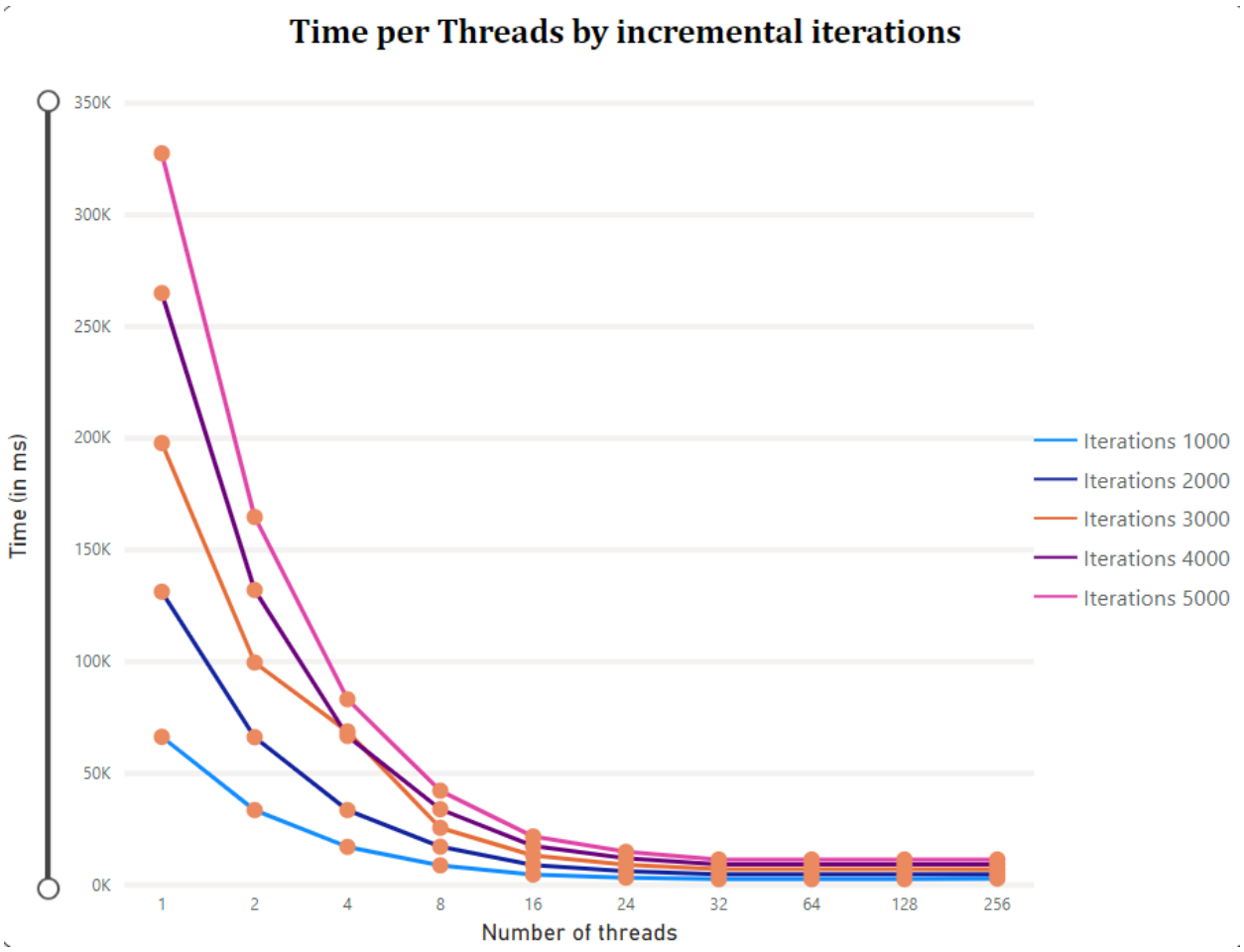
16	4366	2.75	17529	2.97	38671	3.00
8	8480	1.42	34163	1.52	76188	1.52
4	16657	0.72	67342	0.77	150990	0.77
2	33205	0.36	133595	0.39	299969	0.39
1	66257	0.18	265580	0.20	597160	0.19

Threads per Block	ITR-1000 & RES-1000		ITR-3000 & RES-1000		ITR-5000 & RES-1000	
N	T(MS)	S	T(MS)	S	T(MS)	S
256	2547	4.71	6697	5.67	11081	5.69
128	2290	5.24	6729	5.65	11096	5.68
64	2289	5.24	6725	5.65	11084	5.68
32	2286	5.25	6731	5.65	11084	5.68
24	2980	4.03	8789	4.32	14525	4.34
16	4366	2.75	12924	2.94	21389	2.95
8	8480	1.42	25296	1.50	41908	1.50
4	16657	0.72	68441	0.56	82838	0.76
2	33205	0.36	99300	0.38	164442	0.38
1	66257	0.18	197541	0.19	327291	0.19

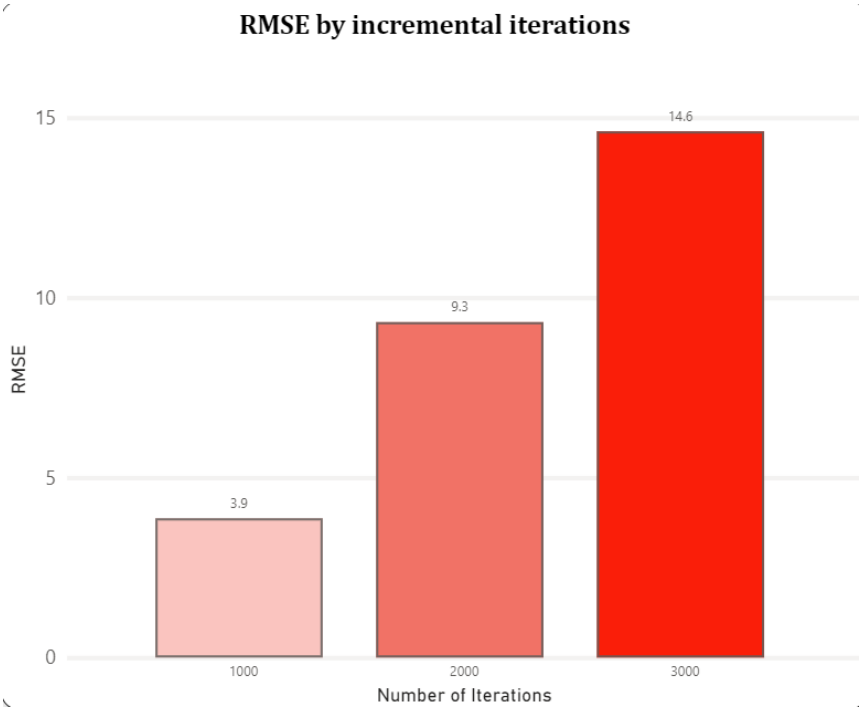


While changing the number of threads per block, the number of blocks changes considerably to fit the problem size.

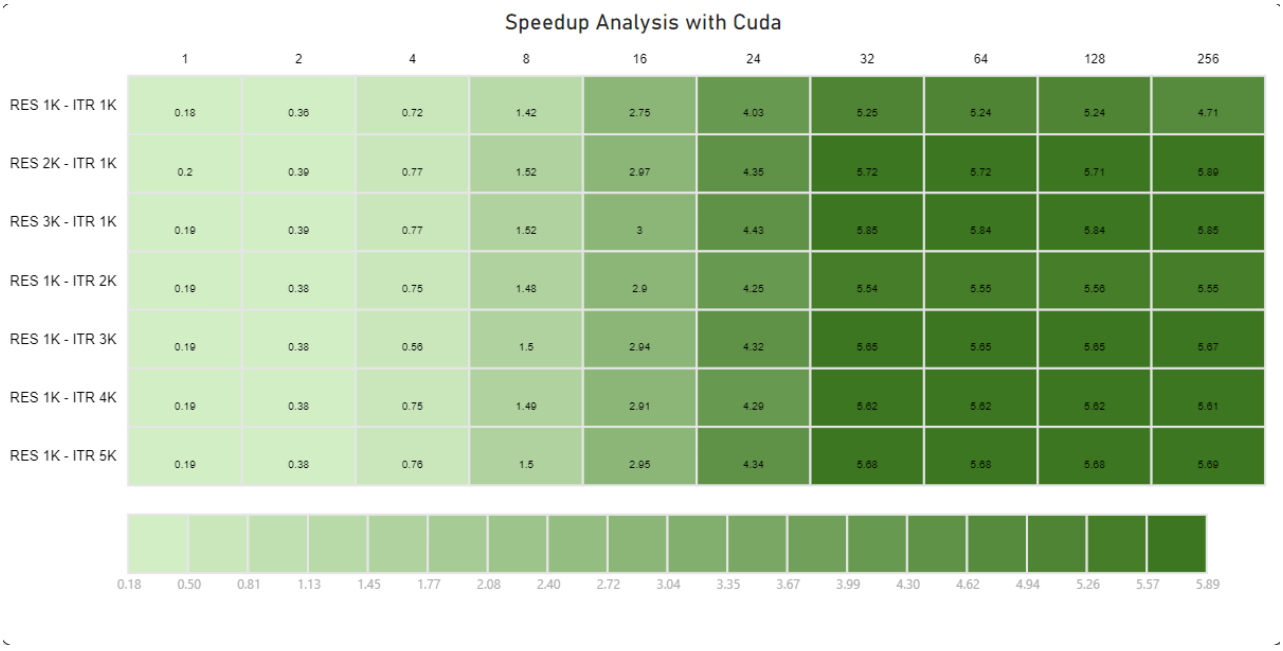




After utilizing /32/ threads per block, we get a plateau across the time elapsed.



Utilizing CUDA to compute the Mandelbrot set introduces RMSE in the generated images, which makes sense because the GPU is not as accurate as the CPU. Also increasing the iterations increases the error in a nonlinear way, while increasing the resolution does not, since RMSE is normalized. Increasing the iteration increases the accumulated error because calculations are built on top of each other.



CUDA threads are lightweight threads that deal with independent light tasks. Apparently after reaching the /32/ threads per block speedup does not enhance and could be justified by the fact that the wrap size is /32/ inside the GPU.

We can also notice the difference between the maximum speedup achieved by OpenMP /14.29/ and the maximum speedup achieved by CUDA /5.89/. Last observation, the speedup occurred by different configurations RES-ITR is aligned with a slight difference.

MPI

MPI code will get executed on different processes across different machines, where data is exchanged between them, MPI uses Message Passing to share data, which is different from OpenMP which uses shared memory.

Each process is responsible for a specific portion of the problem size decided by the number of processes and identified by its rank.

After all processes finish we need to *gather* the data from all processes in the root node.

```
    int myid, numprocs;
    int * image;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);

    auto start = chrono::steady_clock::now();
    int dutySize =  (HEIGHT * WIDTH) / numprocs;
    int startPos = myid * dutySize;
    int endPos = startPos + dutySize;

    //if the process rank is zero, start counting time and define the
array where we want to gather all processes results
    if(myid == 0){
         cout<< "Image Resolution: "<< RESOLUTION<<endl;
         cout<< "#Iterations: "<< ITERATIONS <<endl;
        image = new int[HEIGHT * WIDTH];
        std::cout<<"Number of Processes: "<<numprocs<<std::endl;
        std::cout<<"Duty Size for each Process is:
"<<dutySize<<std::endl;
    }


    int * partialImage = new int[dutySize];

    //omp_set_num_threads(256);
```

```
    //#pragma omp parallel for schedule(dynamic)
    for (int pos = startPos; pos < endPos; pos++)
    {
        partialImage[pos - startPos] = 0;

        const int row = pos / WIDTH;
        const int col = pos % WIDTH;
        const complex<double> c(col * STEP + MIN_X, row * STEP +
MIN_Y);

        // z = z^2 + c
        complex<double> z(0, 0);
        for (int i = 1; i <= ITERATIONS; i++)
        {
            z = pow(z, 2) + c;

            // If it is convergent
            if (abs(z) >= 2)
            {
                partialImage[pos - startPos] = i;
                break;
            }
        }
    }

    MPI_Gather(partialImage, dutySize, MPI_INT, image, dutySize,
MPI_INT, 0, MPI_COMM_WORLD);
```

Finally, we need to terminate the MPI execution environment by calling:

```
    MPI_Finalize();
```

We tried to use different experiments to evaluate the performance of the MPI parallelization:

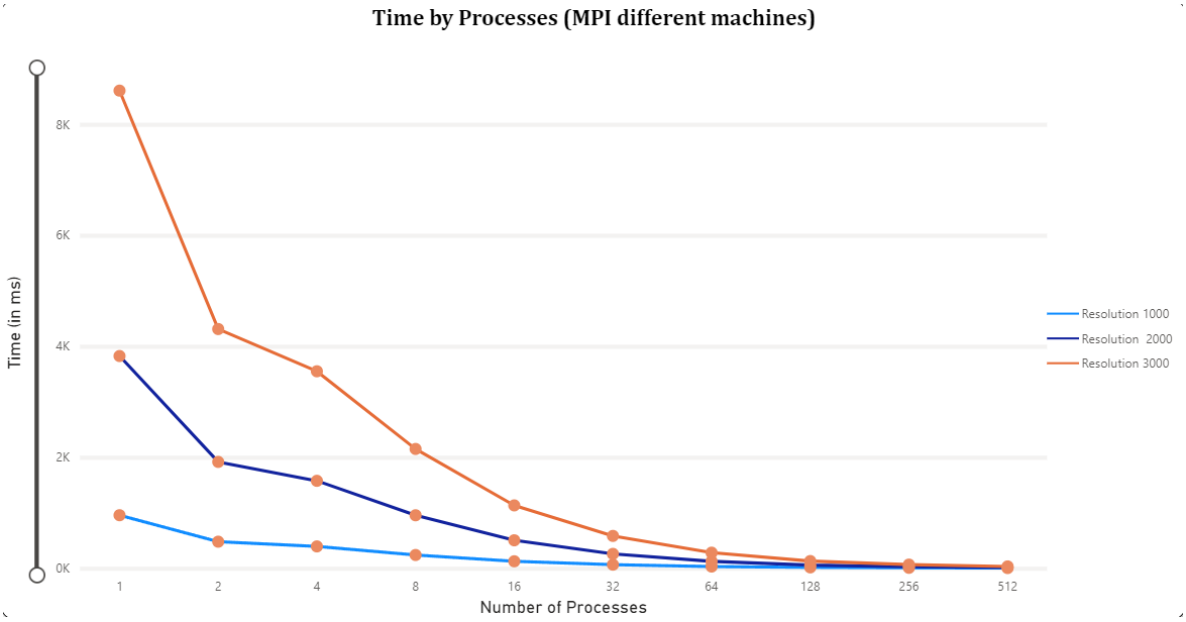
- Trying to distribute processes evenly across all the available machines in the cluster.
- Trying to use a single machine with multiple processes running inside.
- Trying to mix *MPI* with *OpenMP*, where the /256/ threads inside each machine were exploited.

Processes are distributed on all machines.

Processes Number	RES-1000 & ITR-1000		RES-2000 & ITR-1000		RES-3000 & ITR-1000	
	T(S)	S	T(S)	S	T(S)	S
512	3	4	15	3.47	32	3.62
256	7	1.71	28	1.86	65	1.78
128	15	0.8	56	0.93	129	0.9

64	30	0.4	124	0.42	280	0.41
32	64	0.19	258	0.2	582	0.2
16	125	0.1	503	0.1	1132	0.1
8	238	0.05	954	0.05	2149	0.05
4	393	0.03	1575	0.03	3549	0.03
2	478	0.03	1916	0.03	4310	0.03
1	955	0.01	3824	0.01	8607	0.01

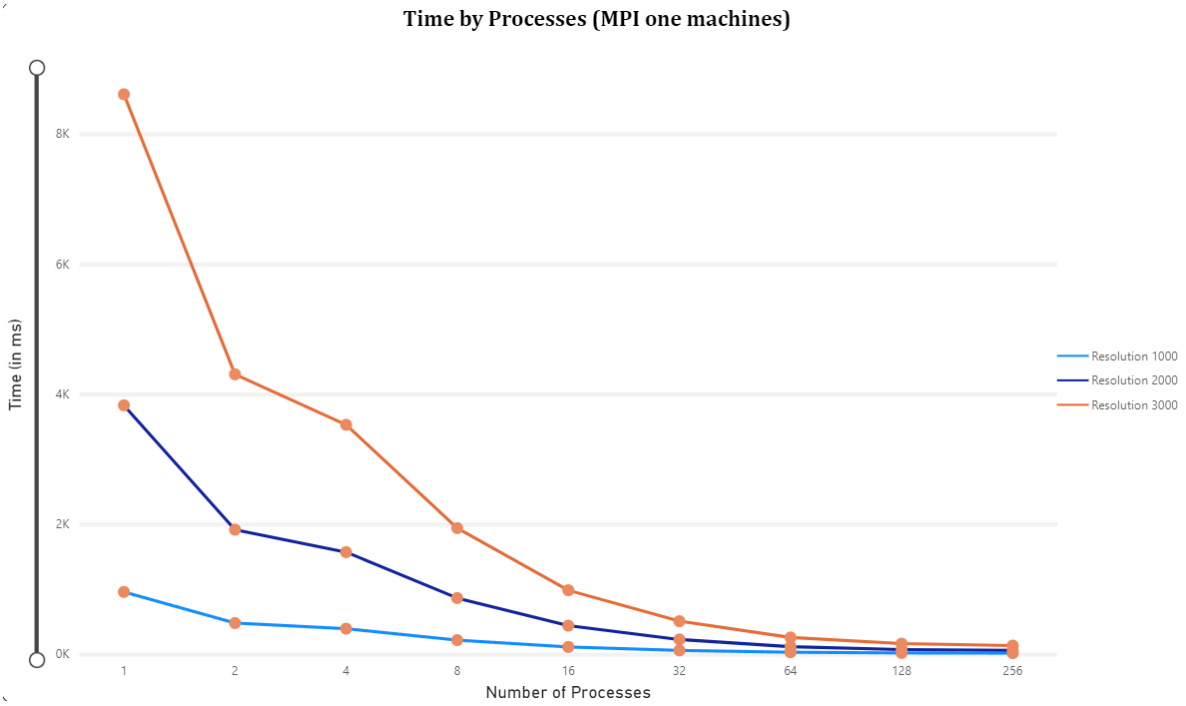
We can notice the time enhancement while increasing the number of processes reaching the maximum number of processes experimented without getting a plateau.



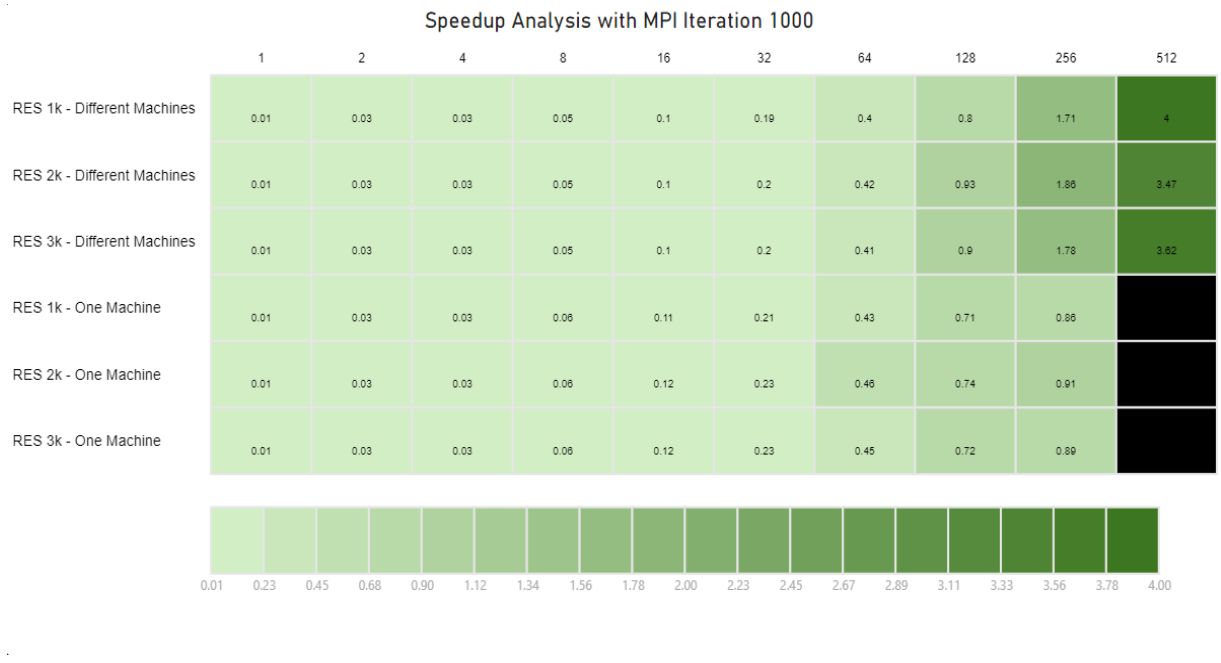
Processes distributed on one machine:

Processes Number	RES-1000 & ITR-1000		RES-2000 & ITR-1000		RES-3000 & ITR-1000	
	T(S)	S	T(S)	S	T(S)	S
256	14	0.86	57	0.91	130	0.89
128	17	0.71	70	0.74	160	0.72
64	28	0.43	113	0.46	255	0.45
32	56	0.21	224	0.23	505	0.23
16	109	0.11	436	0.12	981	0.12
8	214	0.06	861	0.06	1936	0.06
4	390	0.03	1567	0.03	3526	0.03
2	477	0.03	1912	0.03	4302	0.03
1	955	0.01	3825	0.01	8609	0.01

We can notice the time enhancement while increasing the number of processes reaching /64/ processes, while getting a plateau after that, which could be justified by the resource capacity.



All machines vs. one Machine:



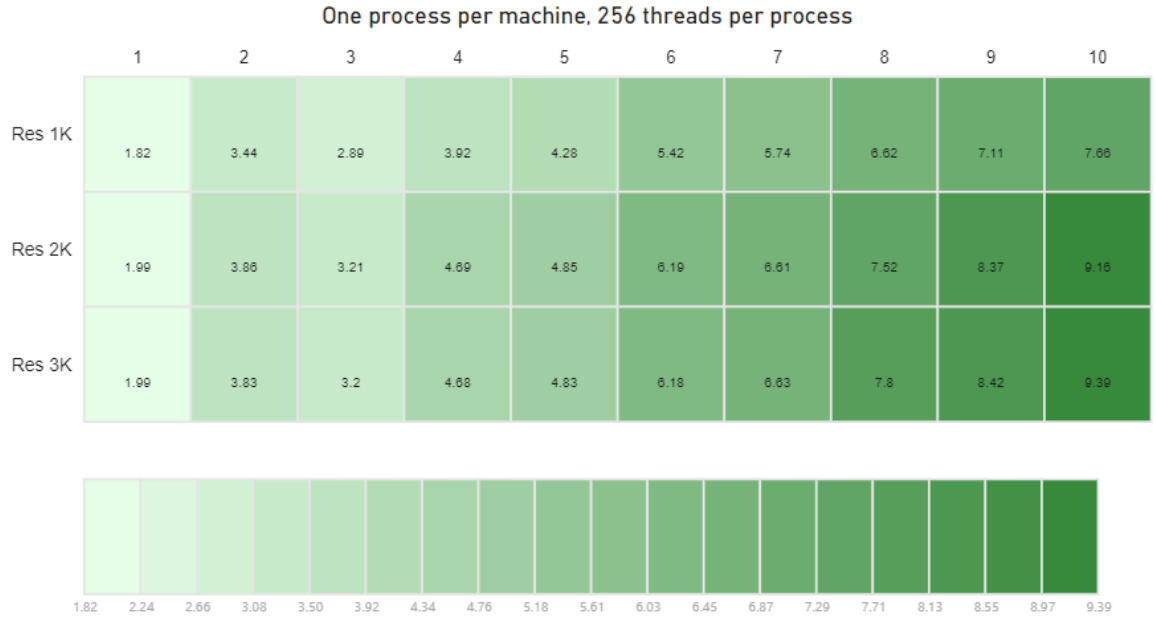
After comparing the speedup achieved by employing processes across different machines and processes inside a single machine, we find that while employing processes till /64/ the performance of processes in a single machine is slightly better and could be justified by the low overhead compared with communication across different nodes. After employing more processes, more than /64/, it becomes the opposite since a single machine has a limited resource to serve the requested process so overhead across different nodes becomes worthy and more effective.

MPI-OpenMP

Employing all threads inside a single machine in one process, in this experiment we assigned one process to each machine.

Number of Machines	RES-1000 & ITR-1000		RES-2000 & ITR-1000		RES-3000 & ITR-1000	
	T(S)	S	T(S)	S	T(S)	S
10	1.567	7.66	5.677	9.16	12.348	9.39
9	1.688	7.11	6.214	8.37	13.774	8.42
8	1.812	6.62	6.917	7.52	14.875	7.8
7	2.089	5.74	7.872	6.61	17.507	6.63
6	2.212	5.42	8.396	6.19	18.757	6.18
5	2.802	4.28	10.726	4.85	24.006	4.83
4	3.059	3.92	11.079	4.69	24.779	4.68
3	4.148	2.89	16.179	3.21	36.223	3.2
2	3.488	3.44	13.466	3.86	30.258	3.83
1	6.590	1.82	26.070	1.99	58.330	1.99

Speedup



We can notice that employing *OpenMP* with *MPI* enhanced the time elapsed, Although it is good, the speedup is not linear. Combining the two techniques allows us to make a trade between exploiting the maximum of the local resource, which decreases the overhead that occurs while distributing work among different nodes, and exploiting all the available nodes.

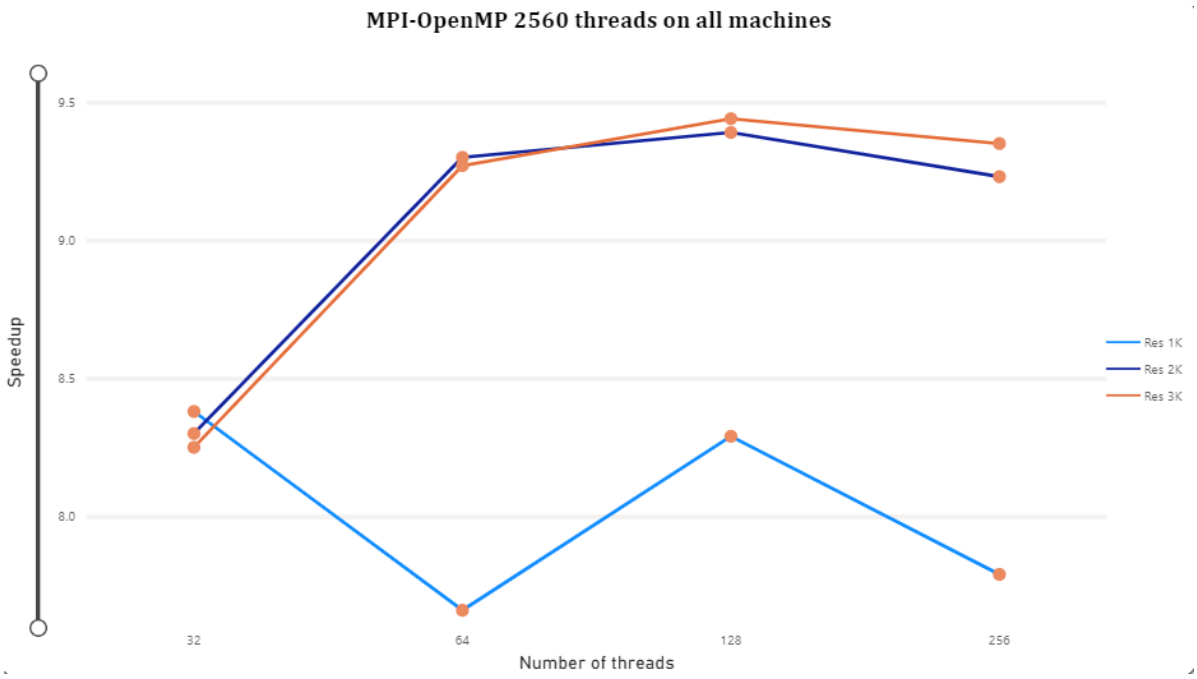
When we compare the speedup analysis of this approach with the previous two MPI approaches we can see the speedup here is almost triple!

*Distributing all threads in the cluster [256(threads per node) * 10 (number of active nodes)] evenly among different processes distributed evenly among all machines.*

#P means the number of processes employed across all machines, and #THR is the number of threads that each process gets.

#P	#THR	RES-1000 & ITR-1000		RES-2000 & ITR-1000		RES-3000 & ITR-1000	
		T(S)	S	T(S)	S	T(S)	S
80	**32**	1.431	8.38	6.265	8.3	14.045	8.25
40	**64**	1.566	7.66	5.590	9.3	12.503	9.27
20	**128**	1.447	8.29	5.532	9.39	12.283	9.44
10	**256**	1.539	7.79	5.628	9.23	12.398	9.35

Speedup



In this experiment, we are trying to find the best number of threads per process that we should employ. After analyzing the graph we can find that /128/ threads per process was the best configuration in the case of all applied resolutions. While employing /128/ threads in each process, 20 processes are employed across all machines which means /2/ processes per machine, given that we have 10 nodes available.

Conclusion

Each parallelization technique has it is own use case that fits the most. According to the timing and speed-up numbers, we find OpenMP as the leading technique in terms of speedup reaching nearly /14/ as a maximum speed up, following it, CUDA reaching nearly /5/ as a maximum speed up and lastly, MPI reaching nearly /4/ as maximum speed up. Regarding accuracy, CUDA introduced errors while computing the Mandelbrot set, while OpenMP and MPI did not.

It is not a fair comparison to compare different techniques with different hardware specifications since the Lab's CPU is powerful but the nodes on the cluster and the employed GPU are not.

OpenMP is good with powerful machines, especially in the case of vertical scaling, which is not always possible or limited, while MPI allows to exploit horizontal scaling. Mixing OpenMP and MPI allows to mix vertical with horizontal scaling, giving the possibility for a wider effective scalability. For that reason we found an enhanced speed up while mixing them reaching nearly /9/ as a maximum speed up.