School *of* Engineering

APMA2822B
Parallel Programming - FALL 2024

Course Project

Maryam Nouh

Hatem Mohamed

Dec. 2024

1. Introduction

This project focuses on solving Poisson's equation in three dimensions using high-performance computational techniques. Poisson's equation is a fundamental partial differential equation widely used in fields such as physics, engineering, and computational sciences to model phenomena like heat conduction, electrostatics, and fluid flow. The computational approach employs C++ programming, leveraging parallel programming paradigms with MPI (Message Passing Interface) and OpenMP (Open Multi-Processing) to enhance performance and scalability. The solution framework is designed to handle large-scale, three-dimensional grids by efficiently distributing the computational workload across multiple processors using MPI for distributed memory systems. OpenMP is integrated to exploit shared memory parallelism within each node, ensuring optimal utilization of resources. This hybrid parallel programming model ensures both fine-grained and coarse-grained parallelism, enabling the solution to scale effectively on modern high-performance computing platforms.

2. Computational framework

The problem is to be solved using the finite difference Method, where the domain is to be represented as a finite number of points and each number is dependent on the neighboring ones.

2.1 Poisson's equation

Poisson's equation is a second-order partial differential equation widely encountered in mathematical physics and engineering. It is used to describe a variety of physical phenomena, including electrostatics, heat conduction, fluid flow, and gravitational fields. The general form of Poisson's equation is given by:

$$\nabla^2 \phi = f(x, y, z)$$

Poisson's equation is often accompanied by boundary conditions, which specify the behavior of the solution on the edges of the domain. These boundary conditions can be Dirichlet (specifying the value of the function), Neumann (specifying the value of the normal derivative of the function), or mixed. In three-dimensional problems, solving Poisson's equation poses a computational challenge due to the large number of unknowns and the need for accurate representation of the solution over the entire domain. Numerical methods, such as the finite difference method, are commonly employed to discretize the domain and approximate the solution. Applications of Poisson's equation are vast, including modeling the electric potential in a charged region, temperature distribution in a steady-state heat conduction problem, and pressure fields in incompressible fluid flows. The ability to solve this equation accurately and efficiently is crucial for advancing technology and understanding complex physical systems.

2.2 Finite difference method

The finite difference method (FDM) is a numerical approach for solving differential equations by approximating derivatives with difference equations. It is widely used in engineering and science to model and solve problems that arise in heat transfer, fluid dynamics, electromagnetism, and structural mechanics. FDM transforms differential equations into algebraic equations that can be

solved computationally. The domain of the problem is discretized into a grid or mesh of points, and the values of the solution are computed at these discrete points. Derivatives in the differential equation are replaced with finite difference approximations, typically using formulas derived from Taylor series expansions. The used formula for calculation is as follows:

$$u(i,j,k) = \frac{1}{6}\left[u(i+1,j,k) + u(i-1,j,k) + u(i,j+1,k) + u(i,j-1,k) + u(i,j,k+1) + u(i,j,k-1) - h^2 f(i,j,k)\right]$$

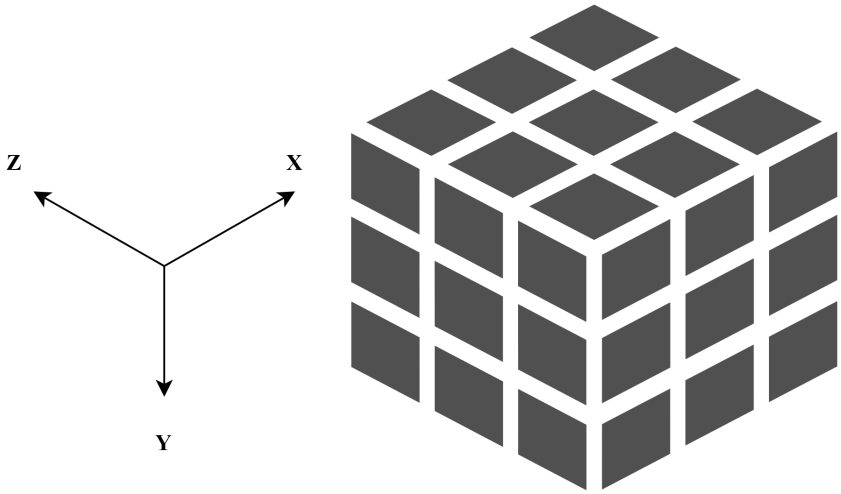
2.3 Problem definition

It is required to solve Poisson's equation in a three dimensional domain of Length equals 1 unit:

$$\nabla^2 \phi = f(x,y,z)$$

$$\text{Where } f(x,y,z) = -(n^2 + m^2 + k^2)\pi^2 sin(n\pi x)cos(m\pi y)sin(k\pi z)$$

Where n, m, and k are the wavenumbers, representing the spatial frequencies in the x, y, and z directions, respectively.

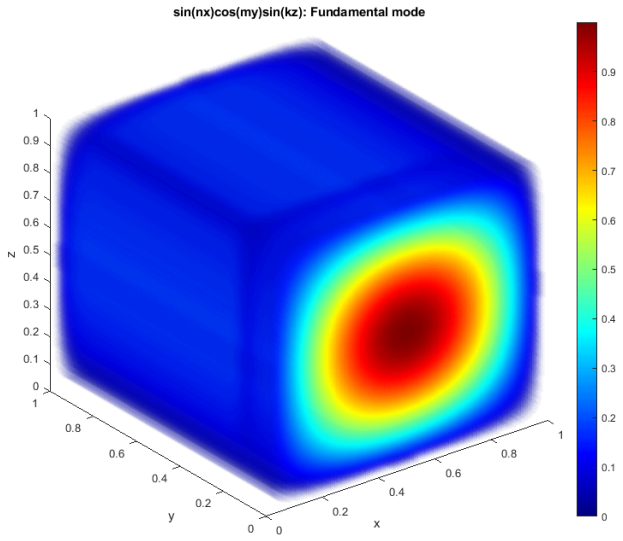


2.4 Analytical solution

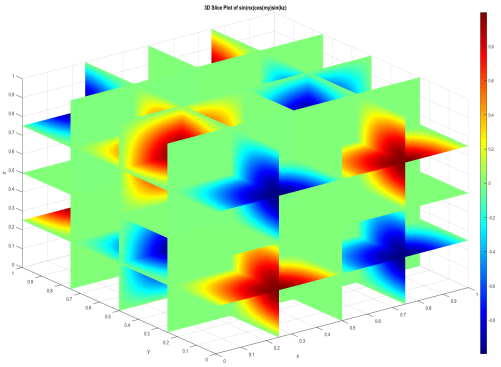
The theoretical solution for the problem is given to be:

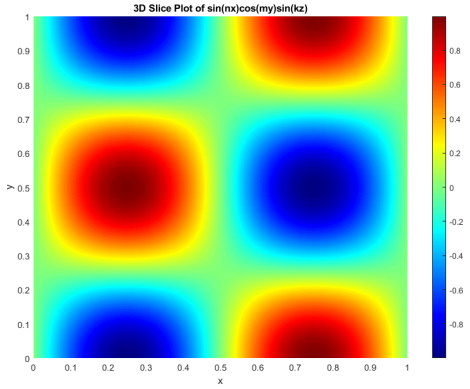
$$sin(n\pi x)cos(m\pi y)sin(k\pi z)$$

Where n, m, and k are the wavenumbers, representing the spatial frequencies in the x, y, and z directions, respectively. The following figure depicts the common mode (n=1, m=0, k=1) exact solution in the 3D domain



Due to dirichlet's condition, and the boundary function, the vertical boundaries are non zero ones (plane y=0, y=1) as can be seen from the following image. The following result is for (n,m,k =2) and this one is used for the analysis as the boundaries are periodic over 2π





3. Implementation

For both implementations, a structured 3D grid is used, with arrays initialized for the potential field and the right-hand side (RHS) of the equation. The RHS is derived from a sinusoidal source term, embodying the equation forcing function. Boundary conditions are imposed on the potential field, ensuring its values are consistent along specified boundaries, and Dirichlet boundary conditions are imposed on the grid edges. Poisson's equation is discretized over a 3D domain represented as a structured grid. The grid size N, the physical domain length L, and the spatial step size are defined, with the solution and the RHS of the equation stored in one dimensional arrays of contiguous memory.

3.1 OpenMP

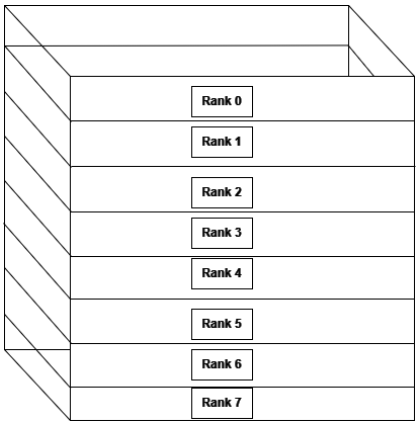
In this implementation, OpenMP pragmas are used to parallelize the solution computation, exploiting multi-core processors to accelerate the process. Boundary conditions are explicitly defined for grid points at the edges. For the interior points, the solution arrays holding the old phi and the new phi (phi_o and phi_n) are initialized to zero.

The grid values are then updated iteratively using the red black algorithm until a specified global error threshold is reached, noting that with our implementation, the threshold is defined on the residual and not the (normalized) error. This approach is ideal for this problem because it alternates updates between two sets of grid points (red and black), ensuring updated values from one set are available for the other. The algorithm proceeds to update each type of the two points in separate for loops. For red point updates, points satisfying `(i+j+k)%2==0` are updated using their six neighbors and the RHS. Similarly, black points satisfying `(i+j+k)%2==1` are updated.

These updates, split into separate loops, are each parallelized with `#pragma omp parallel for reduction(+:global_error) collapse`. OpenMP distributes iterations among threads, allowing multiple red or black points to be updated simultaneously. Moreover, loops are structured to avoid updating boundary points, as their values are fixed by the problem's boundary conditions.

After updating red and black points, the global error is computed as the sum of the absolute differences between the old and new values of the solution. This step is critical for determining convergence. Once an iteration is completed, pointers to the old and new solution arrays are swapped, eliminating the need for redundant copying of data. The iterative process continues until the global residue falls below a predefined threshold (assumed to be `0.0000001`), indicating that the solution has converged.

3.2 MPI



The MPI implementation divides the 3D domain of size NxNxN into subdomains assigned to individual processes. Each process identifies its neighboring processes in two directions: north, and south. This decomposition ensures that each subdomain’s boundaries align with those of its neighbors, facilitating efficient parallel computation.

The three dimensional grid (N points in each dimension) is decomposed among processes, dividing the workload evenly. Each process is responsible for updating a subset of the grid of size N/8 x N x N, referred to as its "local domain." Boundary values are communicated between neighboring processes that are assumed to be organized as shown in the figure below. This structure divides the processes in the y axis into eight groups. Each process is required to communicate with two neighbors to retrieve boundary values and update its local phi.

Assumptions:

- x-axis: positive in the east direction, negative in the west direction
- y-axis: positive in the south direction, negative in the north direction
- z-axis: positive in the inward direction, negative in the outward direction

Boundary communication is essential in solving PDEs across multiple processes, which is applied here with MPI's `MPI_ISend` and `MPI_IRecv` functions ensuring non-blocking and efficient data transfer. Each process exchanges data with its neighbors to update its boundary values. For example, values from the southern boundary of one process (ie. process 0) are sent to the northern boundary of its southern neighbor (ie. process 1), and vice versa. This symmetric exchange ensures consistency across subdomains.

Efficient communication is pivotal in this approach. For each boundary, specific planes of data are exchanged between processes. Temporary arrays are allocated for these planes, minimizing redundant memory operations. MPI barriers are strategically used to synchronize processes, ensuring data consistency before subsequent computations.

The red-black algorithm is then applied to accelerate the convergence when calculating the value of phi at each point, where each rank is required to calculate the values of the points in its subdomain only, with the help of the boundary values of its neighbors. This implementation ensures computational efficiency by performing updates only on interior points of the local subdomain. Points on the boundaries are updated using received data from neighboring processes.

After each iteration, the local error, calculated as the difference between the old and new potential values, is summed. The global error is then computed using `MPI_Allreduce` to aggregate errors across processes. Iterations continue until the global error falls below a predefined threshold (here assumed to be `0.0000001`), signifying convergence.

4. Reference Model (MATLAB)

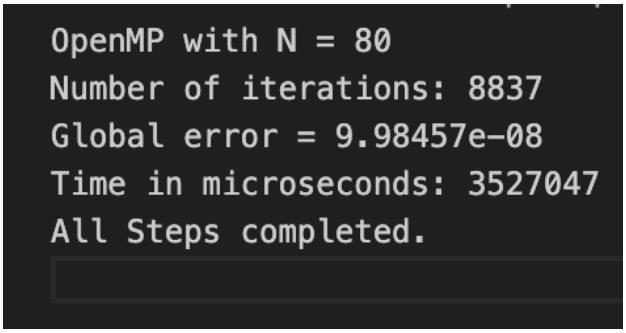
For N = 5 the results are shown below (left: matlab), (right: finite difference results) (nmk=1):

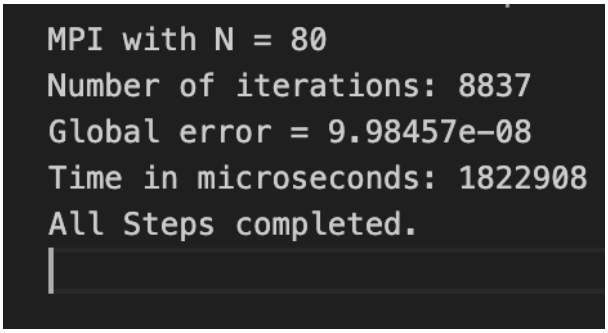
```
XY Planes in Order (Z slices):
Z = 0.00:
     0     0     0     0     0
     0     0     0     0     0
     0     0     0     0     0
     0     0     0     0     0
     0     0     0     0     0


Z = 0.25:
         0    0.5000    0.7071    0.5000    0.0000
         0    0.3536    0.5000    0.3536    0.0000
         0    0.0000    0.0000    0.0000    0.0000
         0   -0.3536   -0.5000   -0.3536   -0.0000
         0   -0.5000   -0.7071   -0.5000   -0.0000


Z = 0.50:
         0    0.7071    1.0000    0.7071    0.0000
         0    0.5000    0.7071    0.5000    0.0000
         0    0.0000    0.0000    0.0000    0.0000
         0   -0.5000   -0.7071   -0.5000   -0.0000
         0   -0.7071   -1.0000   -0.7071   -0.0000


Z = 0.75:
         0    0.5000    0.7071    0.5000    0.0000
         0    0.3536    0.5000    0.3536    0.0000
         0    0.0000    0.0000    0.0000    0.0000
         0   -0.3536   -0.5000   -0.3536   -0.0000
         0   -0.5000   -0.7071   -0.5000   -0.0000


Z = 1.00:
   1.0e-15 *

         0    0.0866    0.1225    0.0866    0.0000
         0    0.0612    0.0866    0.0612    0.0000
         0    0.0000    0.0000    0.0000    0.0000
         0   -0.0612   -0.0866   -0.0612   -0.0000
         0   -0.0866   -0.1225   -0.0866   -0.0000
```

```
Z= 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
-0 -0 -0 -0 -0


Z= 0.25
0 0.5 0.707107 0.5 8.65956e-17
0 0.363942 0.514692 0.363942 0
0 1.6225e-17 1.74746e-17 1.61796e-17 0
0 -0.363942 -0.514692 -0.363942 0
-0 -0.5 -0.707107 -0.5 -8.65956e-17


Z= 0.5
0 0.707107 1 0.707107 1.22465e-16
0 0.514692 0.727884 0.514692 0
0 1.74775e-17 5.18606e-17 3.61535e-17 0
0 -0.514692 -0.727884 -0.514692 0
-0 -0.707107 -1 -0.707107 -1.22465e-16


Z= 0.75
0 0.5 0.707107 0.5 8.65956e-17
0 0.363942 0.514692 0.363942 0
0 2.54769e-17 3.59783e-17 3.46833e-17 0
0 -0.363942 -0.514692 -0.363942 0
-0 -0.5 -0.707107 -0.5 -8.65956e-17


Z= 1
0 8.65956e-17 1.22465e-16 8.65956e-17 1.49976e-32
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
-0 -8.65956e-17 -1.22465e-16 -8.65956e-17 -1.49976e-32
```

There is a small noticeable error between them due to the fact that the Finite difference method solves the problem numerically and error is produced, also the fact that the number of elements is considered small makes this approach less robust.

5. Results and Conclusions

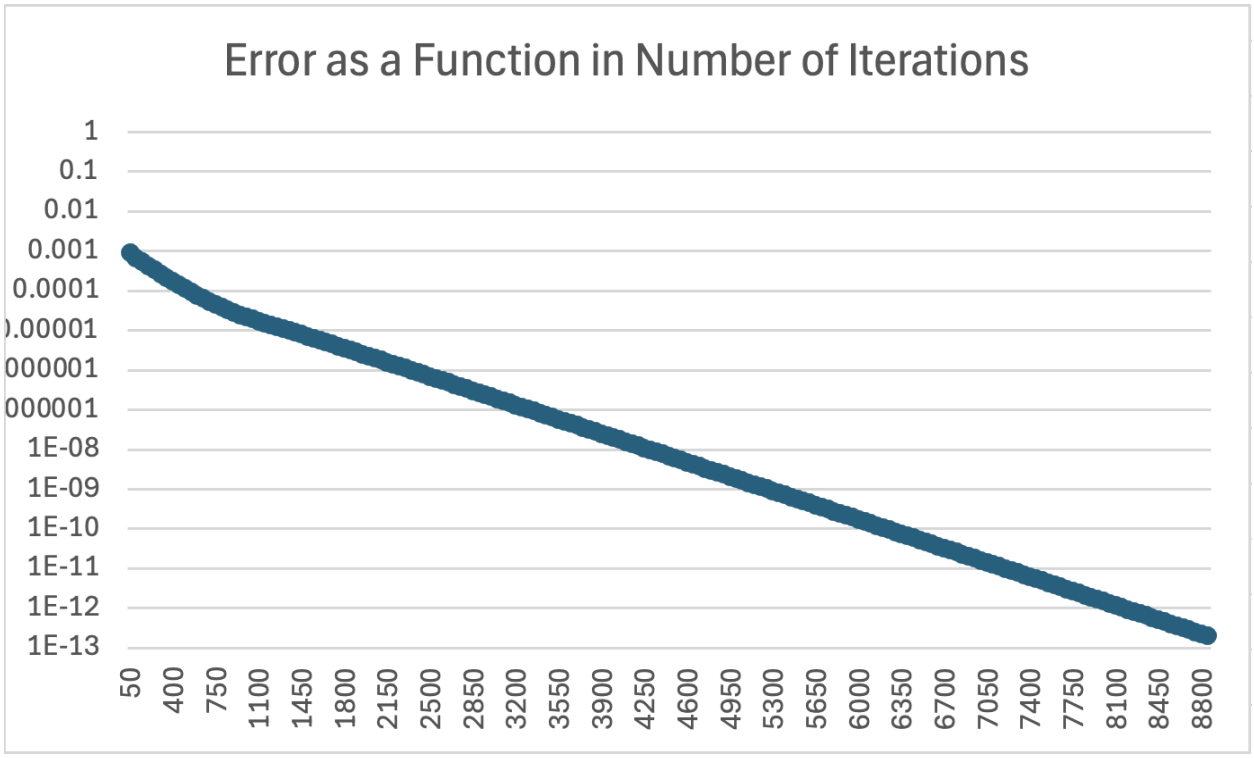


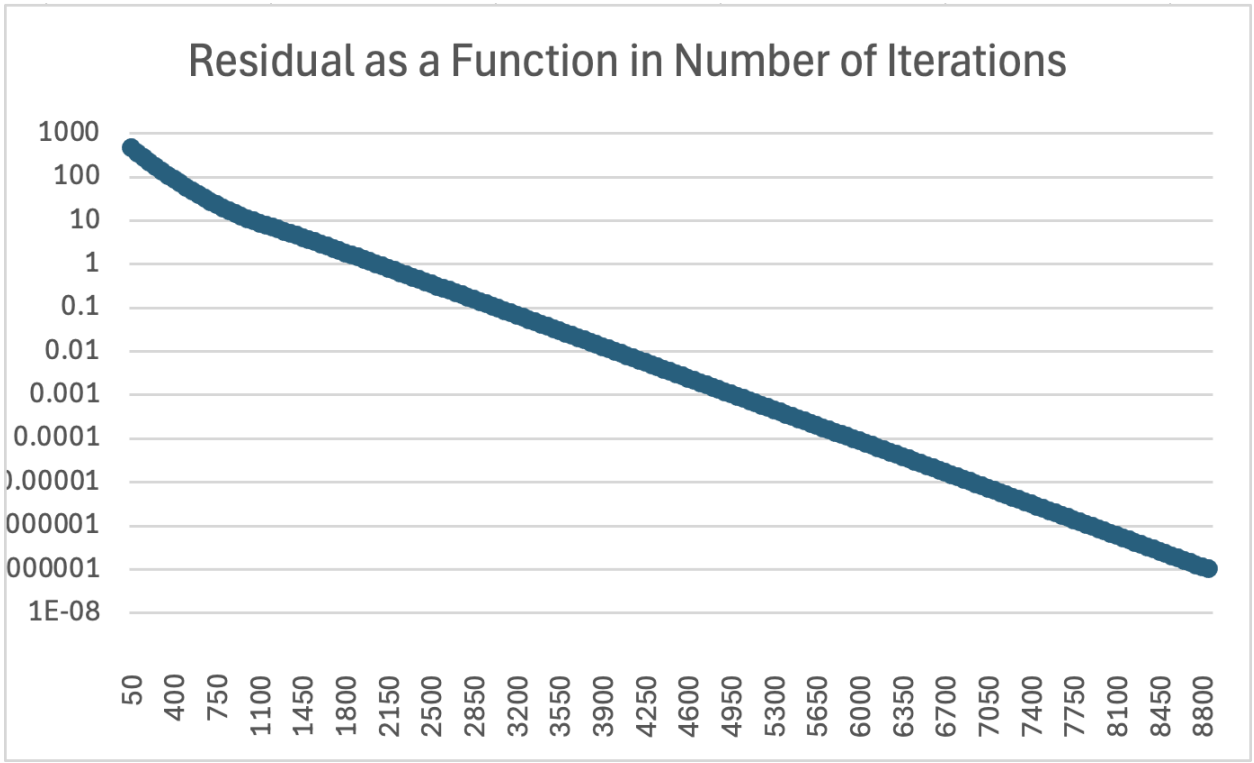


	Task I (Baseline)	Task II (OpenMP)	Task III (MPI)
Time in μs	9779728	3527047	1822908
Number of iterations	8837	8837	8837
Time per iteration (μs)	1106.67	3/99.1226	206.28

The comparison between MPI and OpenMP for solving the 3D Poisson equation highlights MPI's superior performance for a grid size of N = 80. This advantage is primarily due to MPI's distributed memory model, which avoids the memory contention issues inherent in OpenMP's shared memory approach. MPI partitions the domain across processes, ensuring efficient memory usage and balanced workloads. Although MPI incurs communication overhead, the benefits of reduced contention and localized memory access outweigh this cost.

While OpenMP is easier to implement and works well for smaller problem sizes on single machines, MPI excels in scalability and efficiency, especially for larger problems or distributed systems. Thus, for computations like the 3D Poisson equation, MPI is the more effective choice when optimal performance and scalability are required.


Error as a Function in Number of Iterations
1
0.1
0.01
0.001
0.0001
.00001
000001
000001
1E-08
1E-09
1E-10
1E-11
1E-12
1E-13
50
400
750
1100
1450
1800
2150
2500
2850
3200
3550
3900
4250
4600
4950
5300
5650
6000
6350
6700
7050
7400
7750
8100
8450
8800


Residual as a Function in Number of Iterations
1000
100
10
1
0.1
0.01
0.001
0.0001
.00001
000001
000001
1E-08
50
400
750
1100
1450
1800
2150
2500
2850
3200
3550
3900
4250
4600
4950
5300
5650
6000
6350
6700
7050
7400
7750
8100
8450
8800

6. Roofline Model (N=80)

For the roofline model analysis, according to the used finite difference function for calculation, the number of floating point operations per element are 10 operations (type: double), and 56 operations in the indices of type int. While 11 elements of the type double are being read/stored.

$$communication\ per\ iteration\ = 11 \times (N-1)^3 \times 8 = 43.387MB/it$$

$$computation\ per\ iteration\ = 10 \times (N-1)^3 \times 2(double) + 56 \times (N-1)^3 = 37.47MF/it$$

6.1 OpenMP

In the OpenMP part, the time per iteration is 399.122 microseconds.

$$Computation_{/s} = 37.47MF/399.1226\mu s = 93.88GF/s$$

$$BW = 43.387MB/399.1226\mu s = 108.68GB/s$$

$$flop/byte = 0.86373F/B$$

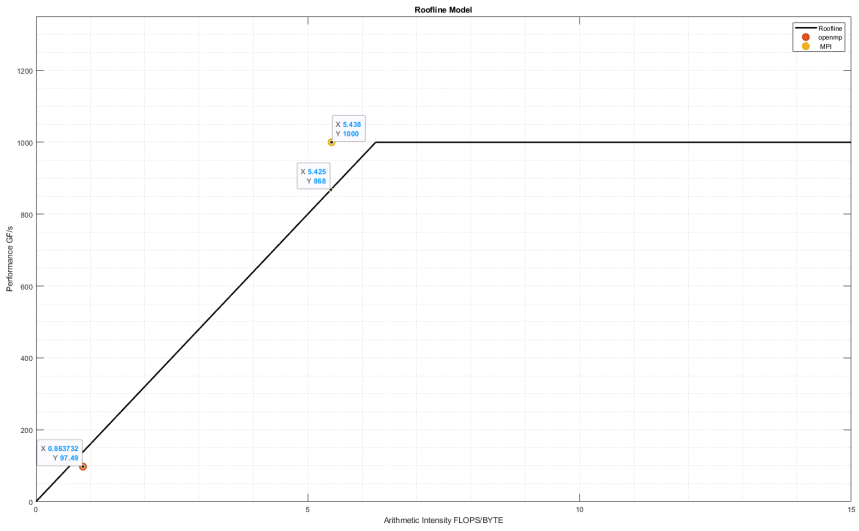
6.2 MPI

In the MPI part, the time per iteration is 2063.28 microseconds.

$$Computation_{/s} = (10 \times (N-1)^3 \times 2(double) + 50 \times (N-1)^3 \times 8(processes))/206.28 = 1\,TF/s$$

$$BW = 11\,MB/206.28\mu s = 217.66GB/s$$

$$flop/byte = 5.318\,F/B$$



Resources are underutilized in OpenMP, however it could be considered as communication bounded. MPI implementation is communication bound. Intuitively, the point changed because the number of index calculations increased, leading to a computation to communication ratio

increase, and an overall tendency toward the crossover point. The point is out of the boundaries as the used specs are the default specs given in assignment I that may not be accurate here.

7. References

[1] Chapra, S. C., & Canale, R. P. (2015). *Numerical Methods for Engineers* (7th ed.). McGraw-Hill Education.