University of Lille

Master 2 Scientific Computing

Supercomputing Project Report

Parallel Mandelbrot Set-based generation of fractals

Report by
(42031822) Taniya Kapoor

Course Instructor
Prof. Nouredine MELAB
Dr. Jan Gmys
Dr. Alexandre Mouton

January 4, 2021

Abstract

Mandelbrot Set is a mathematical application that has gained wide recognition outside of mathematics because of its appeal and complex structure. Our goal is to examine the serial algorithm provided by the course instructors to devise a parallel algorithm, and implement it using the strategies of Message Passing Interface (MPI), OpenMP and CUDA. Hybrid implementations are also explored. The report is concluded with performance measurements and a comprehensive analysis.

Contents

List of Figures

1 Introduction

1.1 Problem Description

The Mandelbrot set, named after Mathematician Benoit Mandelbrot, is a fractal. A fractal is a shape whose boundary exhibits a complicated structure at all length scales and is recursively constructed. It is sometimes referred to as "self-similar," which means that it is exactly or just about similar to a part of itself (i.e. the object resembles one or more of its parts). Fractals often resemble complex patterns found in nature. These complex patterns are coloured resulting in beautiful computer generated images. The question now becomes how do fractals generate these complex patterns? The answer is not easy. For one, a fractal is a set composed of complex numbers. A basic understanding of complex numbers is therefore required to understand fractal generation. Additionally a formula is needed to compute numbers in the fractal set. The Mandelbrot Set, for example, is generated by repeatedly calculating the two formulas listed below. Given the standard X-Y coordinates, choose a point on the plane with coordinates R and S (let's say) and calculate the two formulas as follows:

$$X_n + 1 = X_n^2 - Y_n^2 + R \tag{1.1}$$

$$Y_n + 1 = 2 * X_n * Y_n + S \tag{1.2}$$

1.2 Complex Numbers

The Mandelbrot set is a mathematical set, a collection of numbers. However, these are not our everyday numbers; they are complex numbers. Complex numbers are made up of two parts; a real and imaginary. The real part is a number such as -5, and the imaginary part is a real number (e.g. 9) multiplied by a special symbol (appropriately called i). An example of a complex number would then be 5 + 9i
By definition, i is the square root of -1. In fact, i is what separates real from imaginary numbers. As a matter of fact, i was invented to alleviate the shortcoming of real numbers. Equipped with this invention, we can successfully square an imaginary number and get a negative number. For example, if we square 3i, we will obtain -9. Going back to the above formulas and relating them to our complex number discussion, it is easy to see that we can transform them into one simple formula that looks something like this
C is a constant number; therefore, we can view R and S from the first two formulas as real and imaginary respectively.

1.3 Graphing Mandelbrot set

Real numbers can be represented on a special line known as the number line with negatives numbers on the left and positives on the right of the line. Any real number can be graphed on this line. However, because imaginary numbers have two parts, they can not be graphed on this line. Instead, we add a vertical dimension to the number line and graph them on a complex number plane, which is similar to the X-Y coordinate system. Since the Mandelbrot set is a set of complex numbers, we can graph it on a complex number plane. However, we must answer one question before we proceed. How do we know which numbers are part of the Mandelbrot set and which are not. We need a test to accomplish this.

1.4 Membership Test

Our test to determine the membership of the set (membership test) is based on the quadratic recurrence equation
During the testing process, constant C remains the same throughout. This is the number that we are testing. It is the point on the complex plane that we will be plotting after the testing is complete. Z starts at zero, but it repeatedly changes as we iterate through the above equation. During each iteration, we create a new Z that is equal to the old Z squared plus C. The membership is then determined by the magnitude of Z. If the magnitude of Z remains inside 2 then C is part of the Mandelbrot set, otherwise not. The number of iterations depends on the choice of user.

1.5 Development environment

Initially the code is developed on a MacOS machine. Once completed, we used the supercomputer environments available at the Grid 5000 cluster to test our code. The program is coded in C. To parallelize a problem, a parallel language or a library, beside the high level language we traditionally use, is needed. We used OpenMP and Message Passing Interface (MPI), a set of libraries that can be called from a high level language such as Fortran, C, and C++. For facilitate the working on GPU, we have implemented the code on CUDA as well. Instructions to compile and execute the code on the above environments can be found on the readme file.

1.6 Hardware

The G5K cluster: It has 8 sites, we used the Lille site for our computations. The Lille site has 4 clusters: Chetemi, Chichlet, Chifflet, Chifflot having 15, 8, 8 and 8 nodes

respectively. Each node is equiped with 2 CPUs having 10, 16 14 and 12 cores respectively. Out of the four, two clusters chifflet and chifflot have GPU support. Cuda computations have to be done only on those platforms.

1.7 Software

First, we present a sequential approach. Then, a parallel one with OpenMP and a distributed one with MPI. Next, we perform computations on a GPU to show a version implemented with CUDA.

The application is composed of following features:

- It is written in C
- Given maximum number of sequences iterations, it computes that particular Mandelbrot set
- It must be run in an HPC environment
- GNUPLOT is used for visualisation

2 Serial Algorithm and Implementation

Because the Mandelbrot Set has proven its worthiness as a topic to be studied within the computing community, various algorithms for generating its representation exist today. This goes without saying that the membership test and the magnitude of Z described above play a major role in all of the algorithms. First the serial code provided by the course instructors is implemented and key points are noted.

2.1 Algorithm

As we have described above, continuous iteration through the equation determines the quality of the Mandelbrot set image (fractal) we produce. As a consequence, the more the iterations, the better the image. In algorithm, a repeated calculation is performed on C (the point on the complex plane that we will be plotting once the membership test is completed) and based on its behaviour, a colour is chosen for that pixel. The x and y location of each C are used as a starting values in the iteration. During each iteration, we create a new Z that is equal to the old Z squared plus C and use the result as the starting value for the next iteration. The values are checked during each iteration to see if they have reached a critical condition. If this condition is true, further calculation is stopped, a pixel is drawn and examination of the next C will commence. Some starting values reach the stopping condition fast whereas others may take a large number of iterations to do so. Furthermore, values which pass our membership test (members of the Mandelbrot Set) will never reach the critical condition. Programmers must choose how many iterations or depth they wish to examine, but they also should consider the consequences of their choice. As mentioned above, better images are produced with a larger number of iterations, but this means a longer calculation time for the picture.

2.2 Sequential version

A serial code is run on consumer-level computer with an 2 Ghz Intel Core i7 CPU (the source is compiled with gcc with no particular optimisation flags) than on the cluster Grid5000. Time comparison plot of the same can be found below.

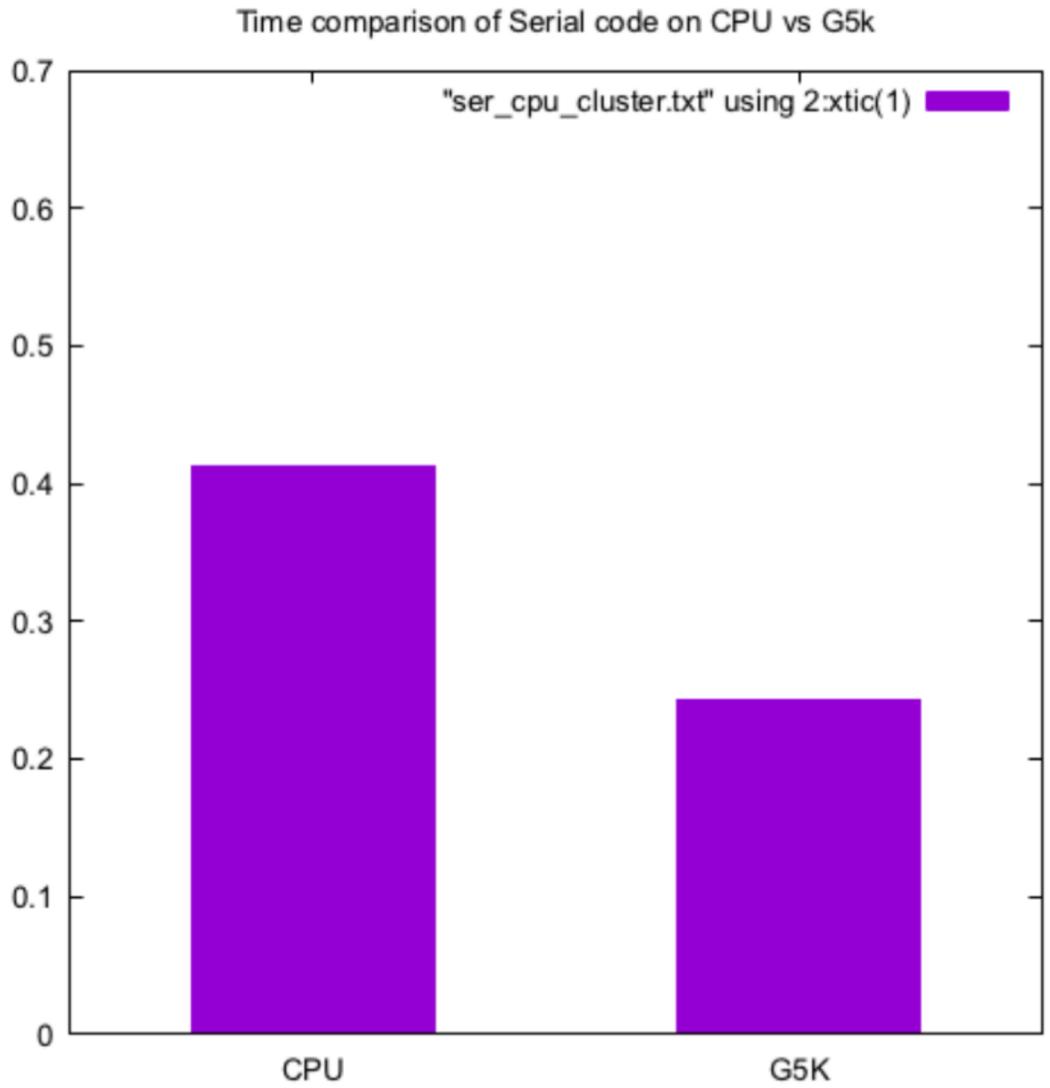

Figure 2.1: Time comparison plot of Serial code on CPU vs G5k (chetemi-6)

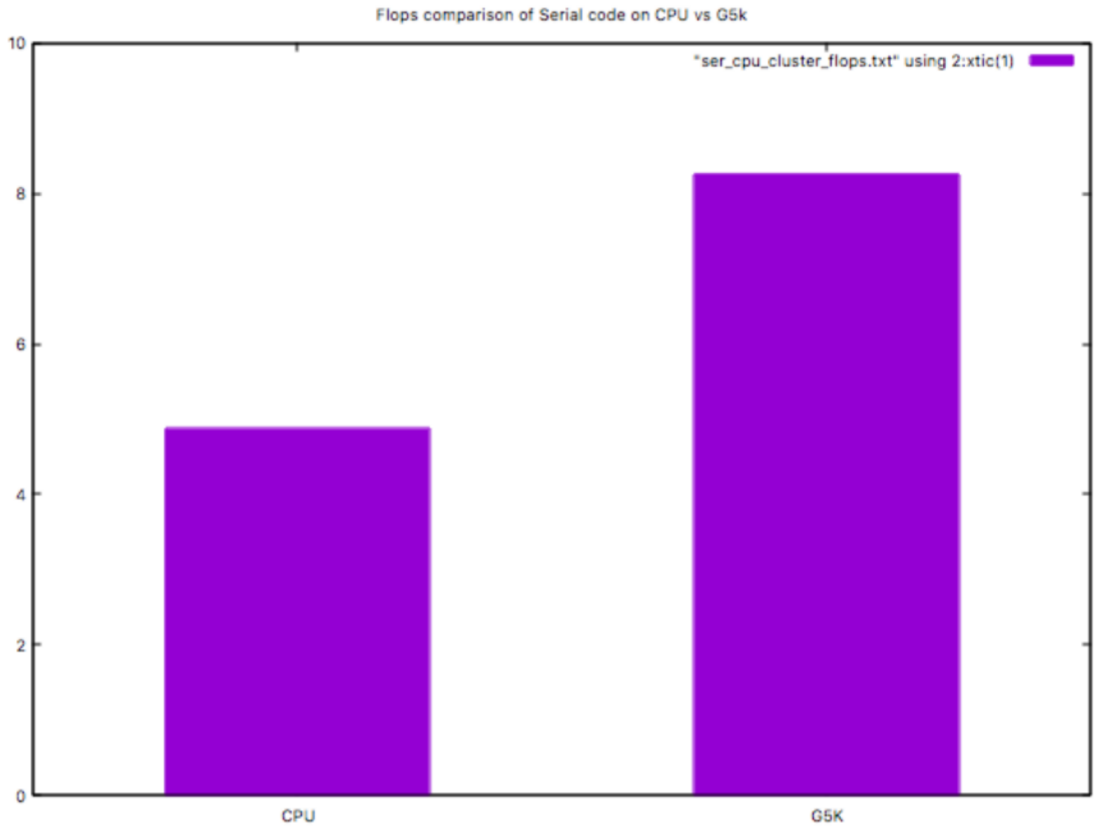

Figure 2.2: Flops comparison plot of Serial code on CPU vs G5k (chetemi-6)

3 MPI implementation

In this chapter we first of all discuss the MPI approach taken to parallelise the code. MPI routines are used and have been implemented assuming that all the processors have distributed memory.

3.1 MPI Gather

MPI Gather routine: The very first routine used is MPI Gather. The idea behind using MPI Gather is first of all, the whole big block of rectangular pixels is divided equally among all the processors, such that each processor picks a block divided on the basis of height(including processor 0), then all the processors call the compute function and calculate the Mandelbrot set for (width X local height). The local results are stored in each processors local memory. Finally MPI Gather function is called for processor 0, which gathers all the local images to form a global image. The plot below shows the speedup of our parallel implementation using the Amdahl's law, which is the ratio of sequential time to the parallel time. Also, plots for load balancing between all the processors and time required to build each block are presented.

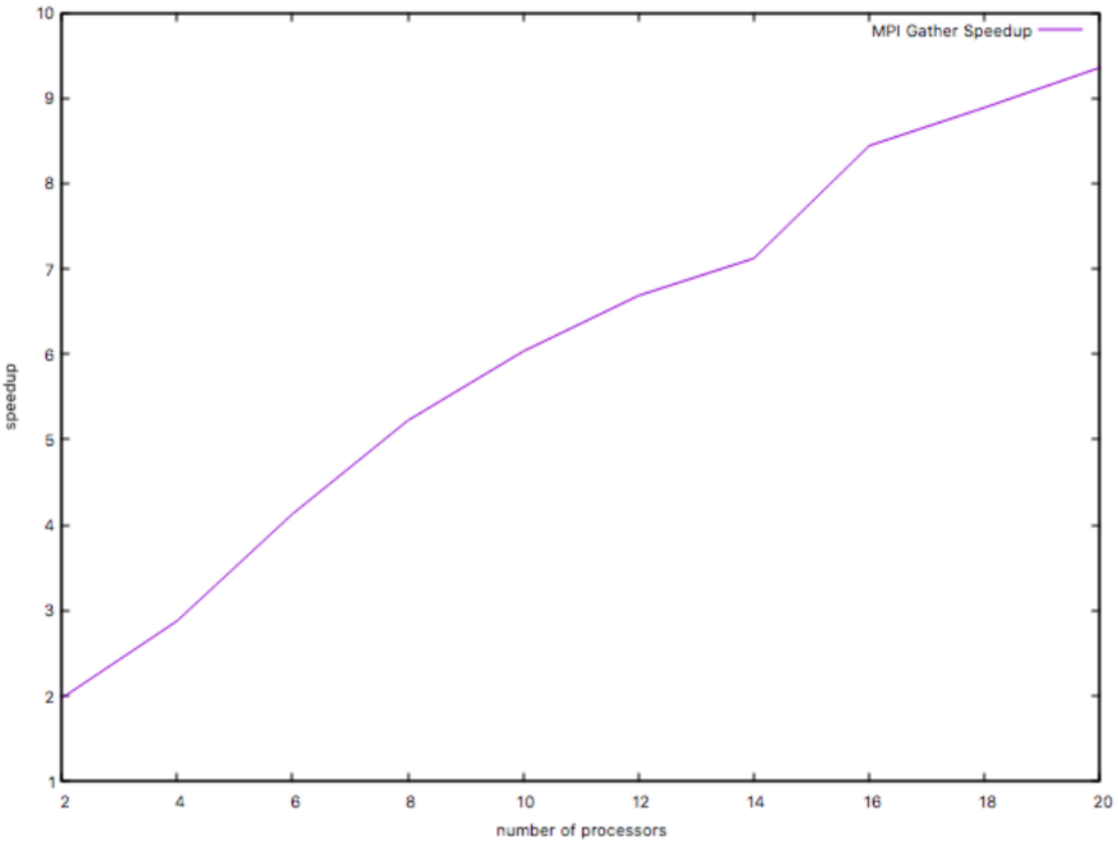

Figure 3.1: MPI Gather number of proc. vs speedup plot

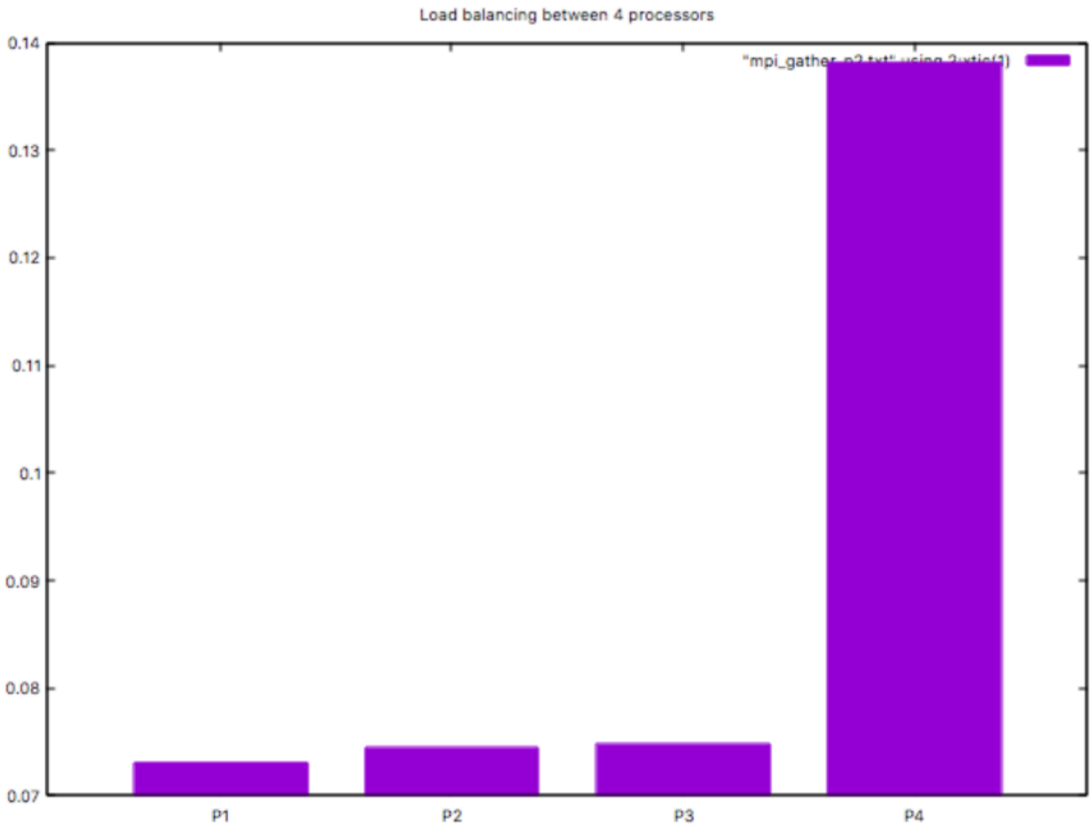

Figure 3.2: MPI Gather Load balancing on 4 procs

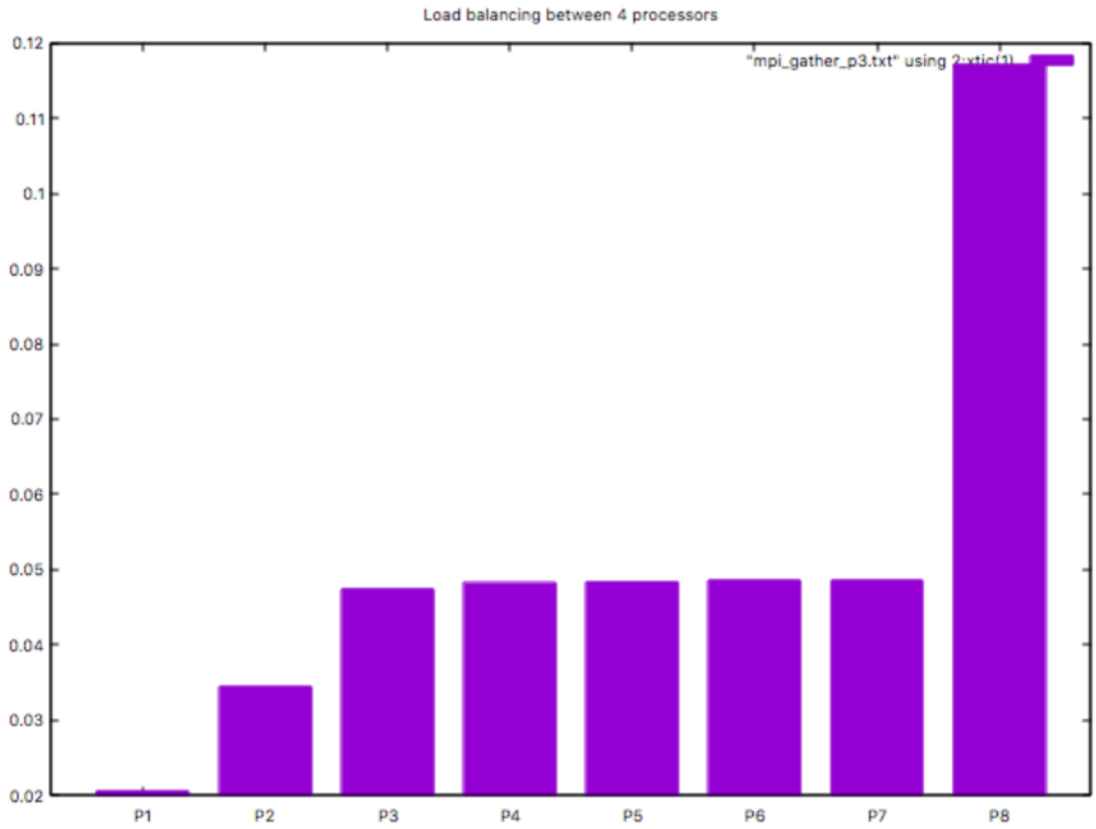

Figure 3.3: MPI Gather Load balancing on 8 procs

3.2 MPI Send Receive

Our next approach is using the MPI routine Send/Recv. we change our method of distributing the work in terms of blocks. Now, we distribute the work between different processors line by line(height wise). The approach is that processor 0 keeps the 1st line, processor 1 is given the 2nd line, processor k is given the k+1 line. The process is done through MPI send by processor 0 and MPI Recv by processor!=0. Now each processor computes the line image and sends it back to the master processor. This is done by MPI Send by processor !=0 and MPI Recv by processor 0. After these are collected in processor 0 the process of sending line (k+1) starts, and the process is repeated till all lines are exhausted batch by batch. speed up plot is presented below. along with the load balancing plots between different processors for the test case of 4 and 8 processors. We observe that the load balancing is correct and is much better than the MPI Gather routine, where processor 0's work was not balanced with the slave processors.

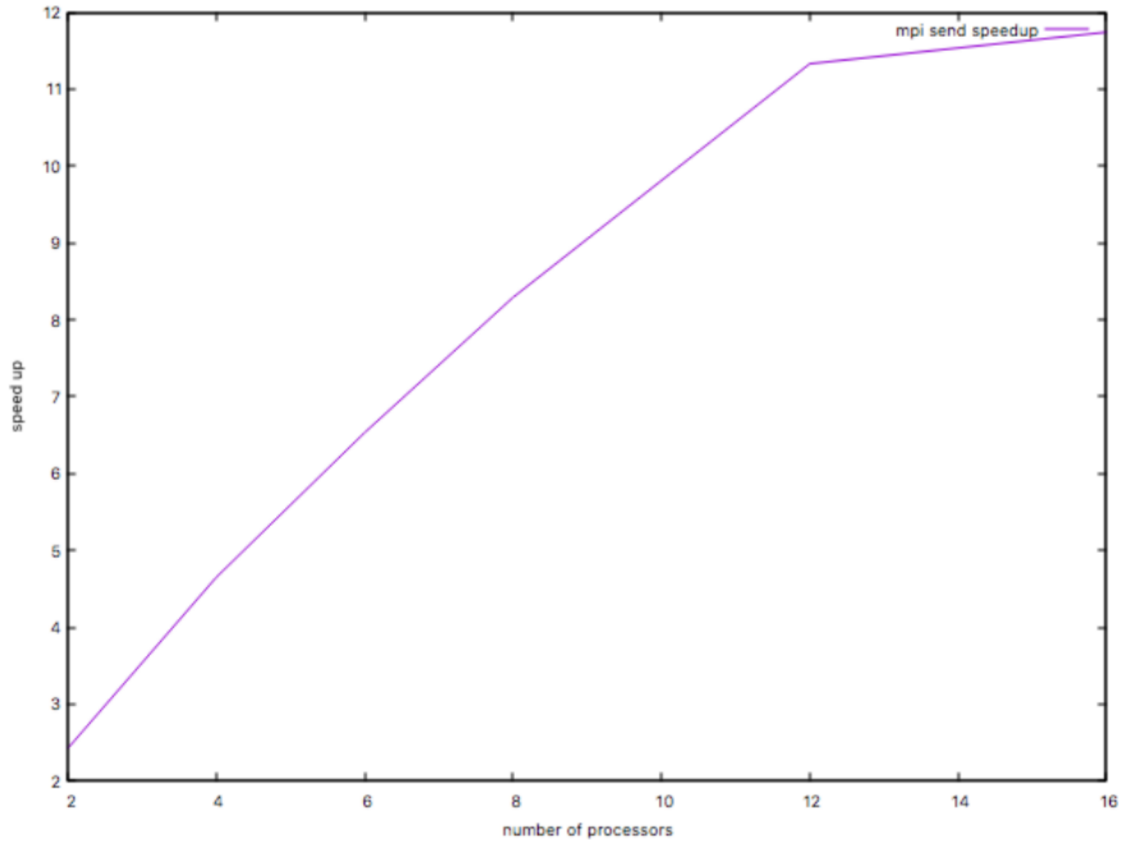

Figure 3.4: MPI send number of proc. vs speedup plot

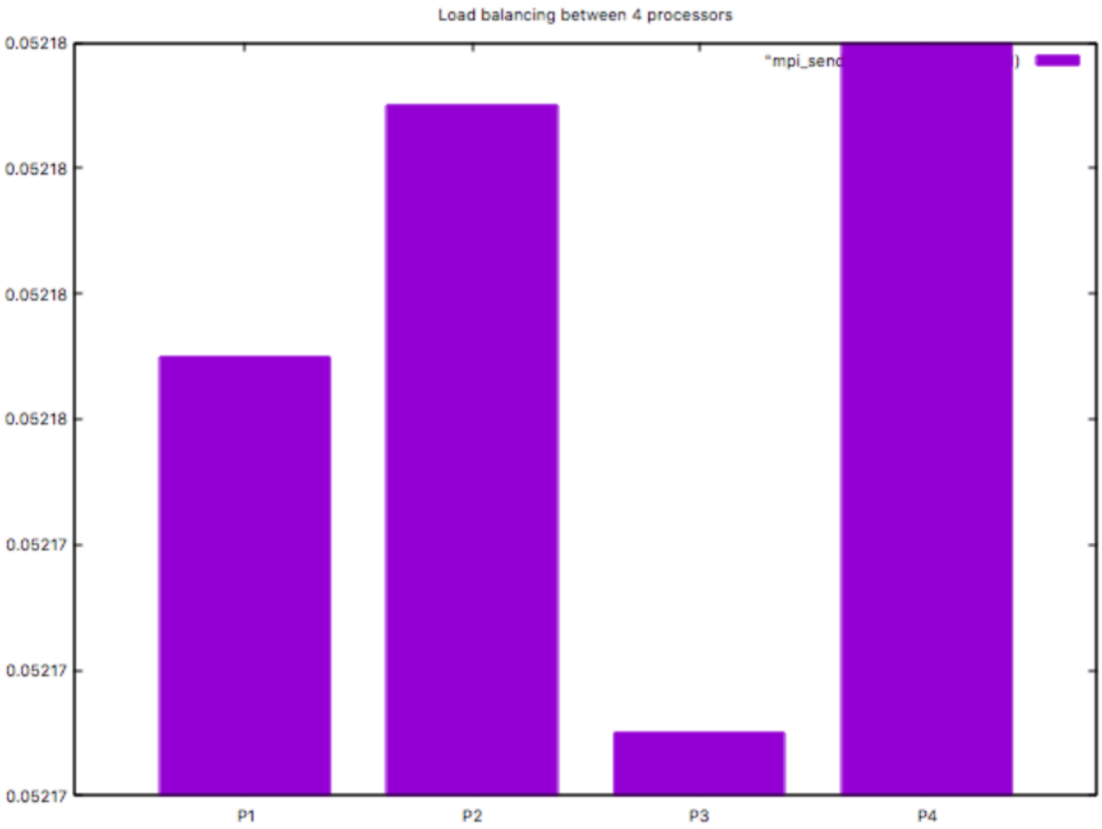

Figure 3.5: MPI send Load balancing on 4 procs

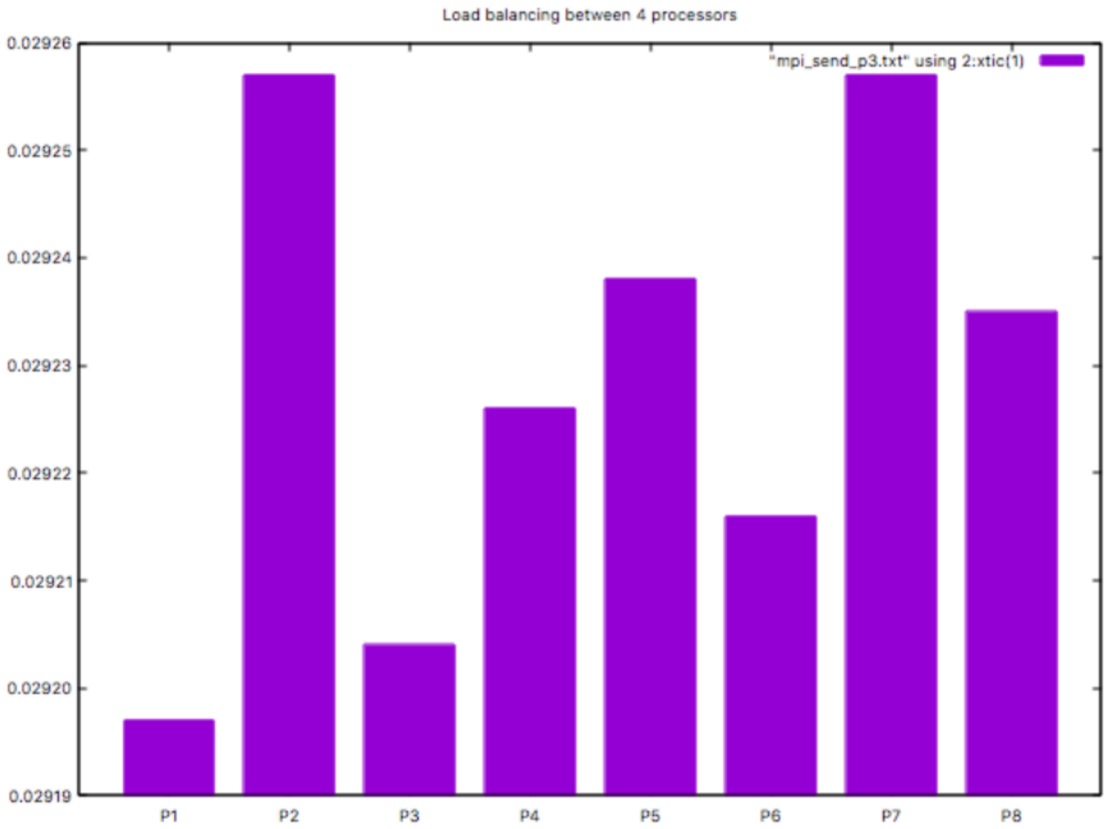

Figure 3.6: MPI send Load balancing on 8 procs

3.3 MPI Isend/Ireceive

MPI Isend/Irecv : In this routine non-blocking communications are explored, in which the difference with the Send/Recv routine lies in the sender side for allowing the computation to overlap with the communication and for the receiver side, this routine allows overlapping a part of the communication overhead, that is copying the message directly into the address space of the receiving side of processors. Speed up plots are presented below. We observe that the best results are obtained in this case and this routine proves to be extremely efficient from both aspects, computational time as well as the load balancing.

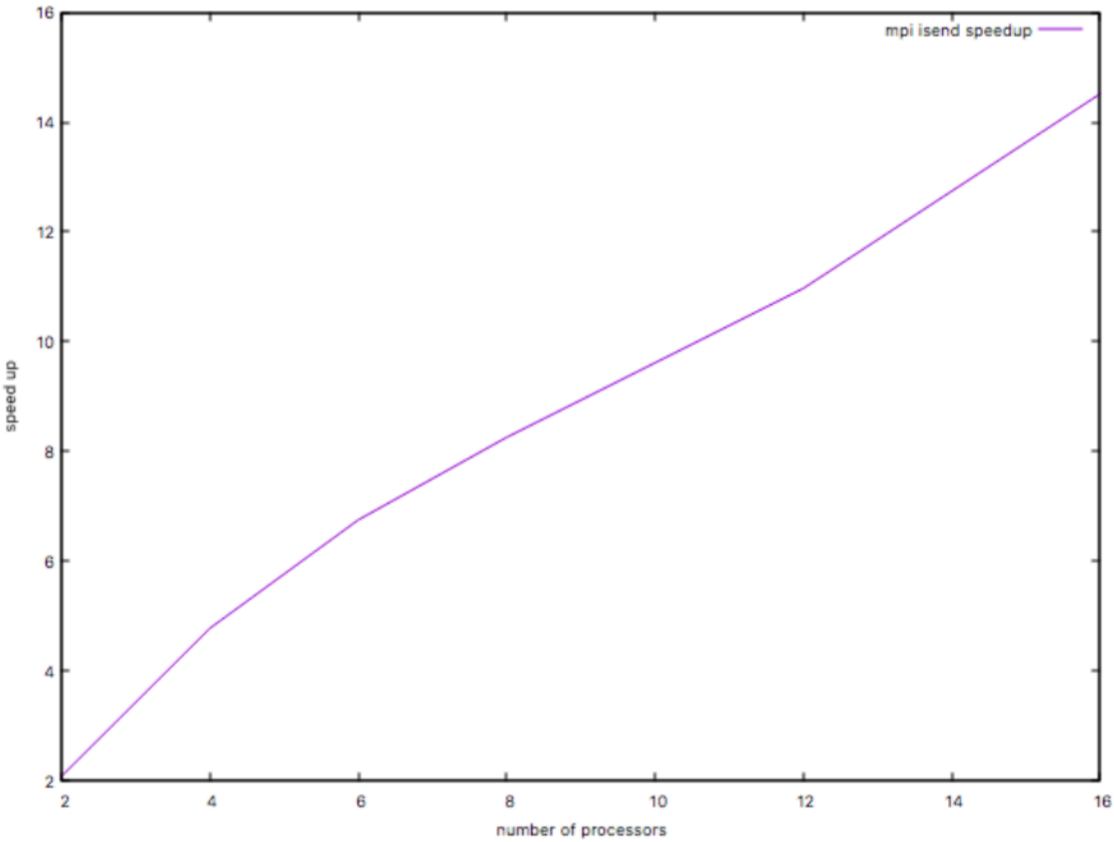

Figure 3.7: MPI Isend number of proc. vs speedup plot

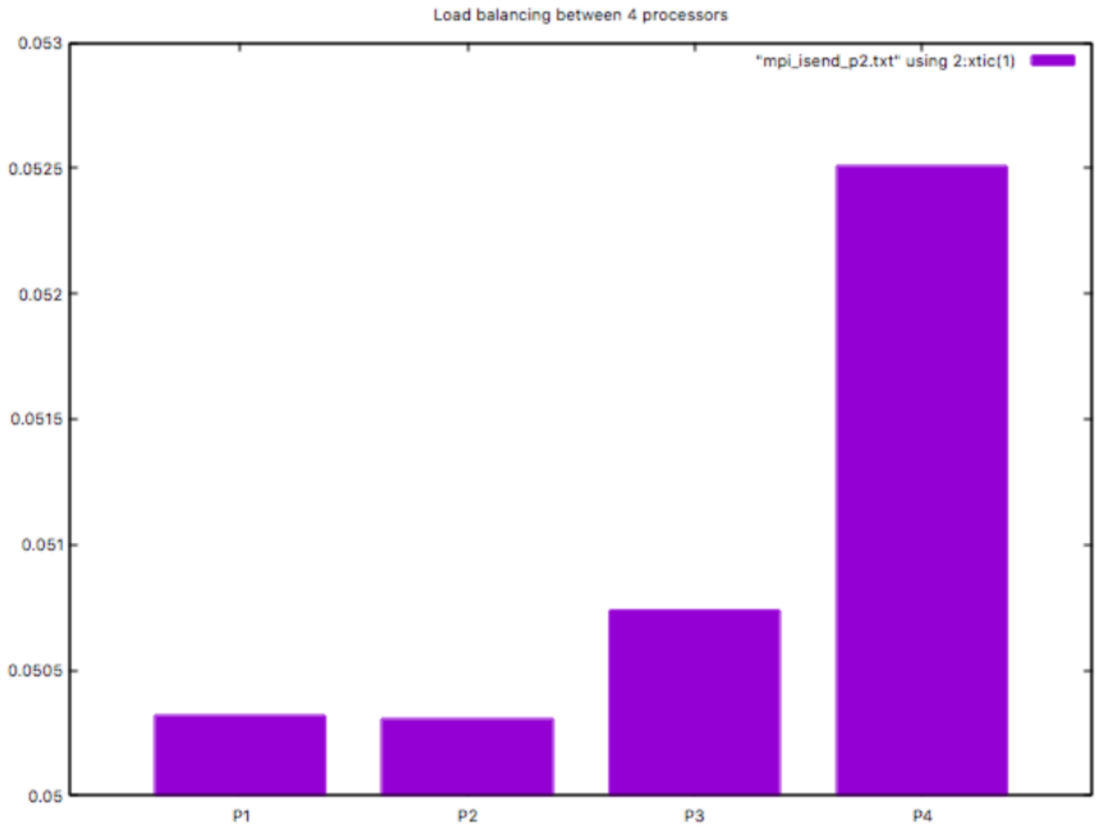

Figure 3.8: MPI Isend Load balancing on 4 procs

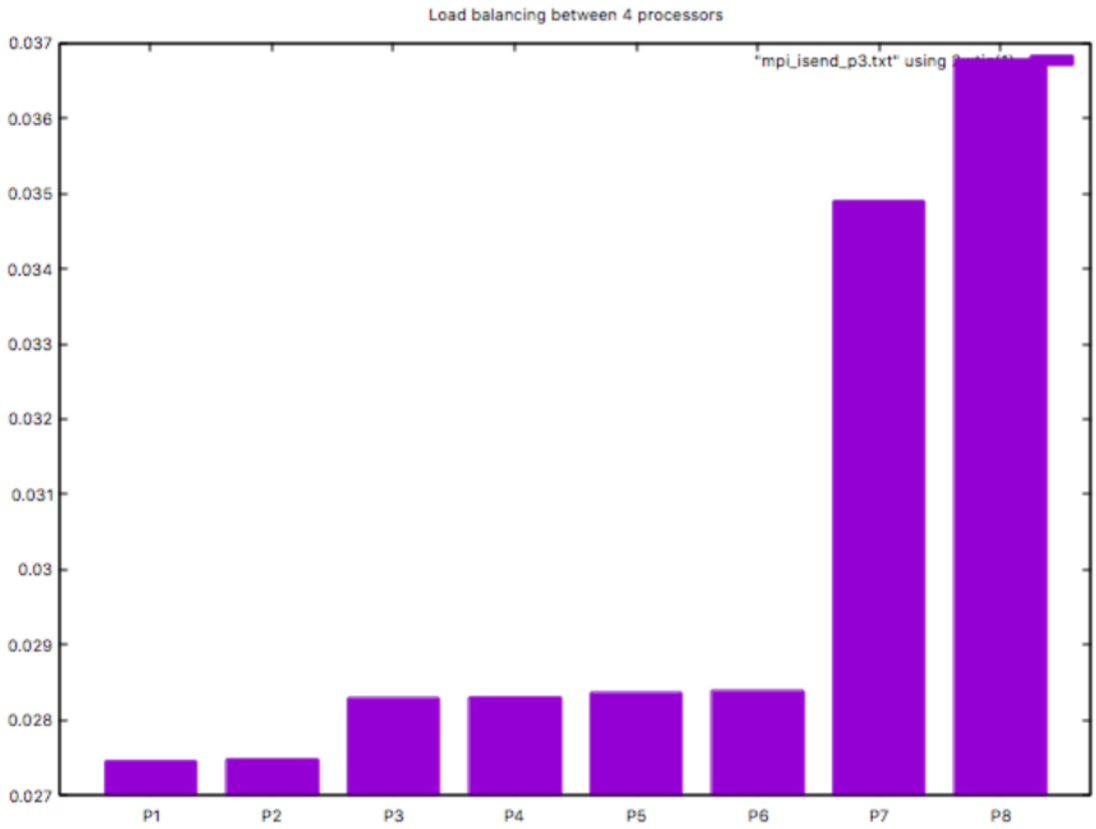

Figure 3.9: MPI Isend Load balancing on 8 procs

3.4 MPI Pack/Unpack

MPI Pack/ Unpack : This is the final MPI routine used for our parallelisation task, In this routine first we have computed all the local images for processor !=0 and the packed into a large size buffer. The buffer is sent by using the MPI send routine(It is worthwhile to mention that processor 0 also computes its own chunk and the uses it later to store at im from imloc.) The processor 0 first receives the buffer using the routine MPI Recv and then unpacks it using MPI Unpack. Processor 0 stores the buffer at correct indices and print the image. The speed up plots are presented below. This concludes the MPI strategies used to parallelise the Mandelbrot set generation.

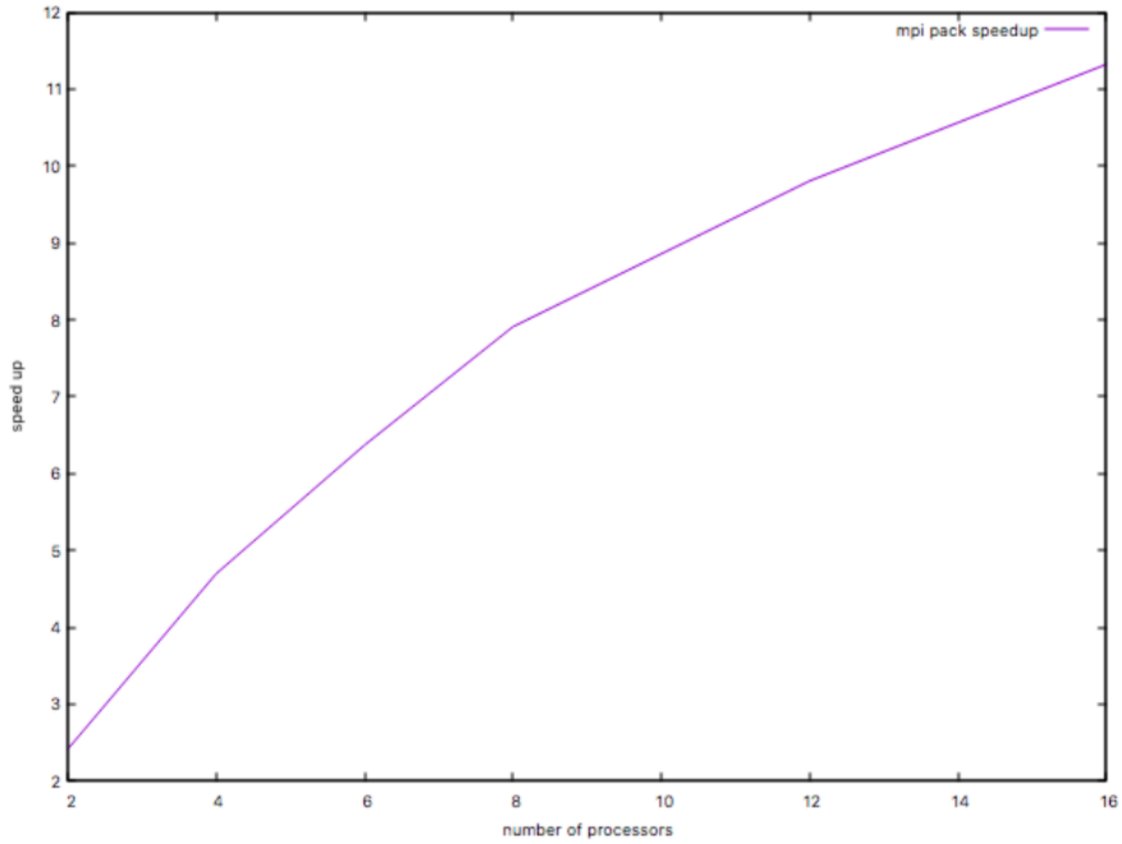

Figure 3.10: MPI pack number of proc. vs speedup plot

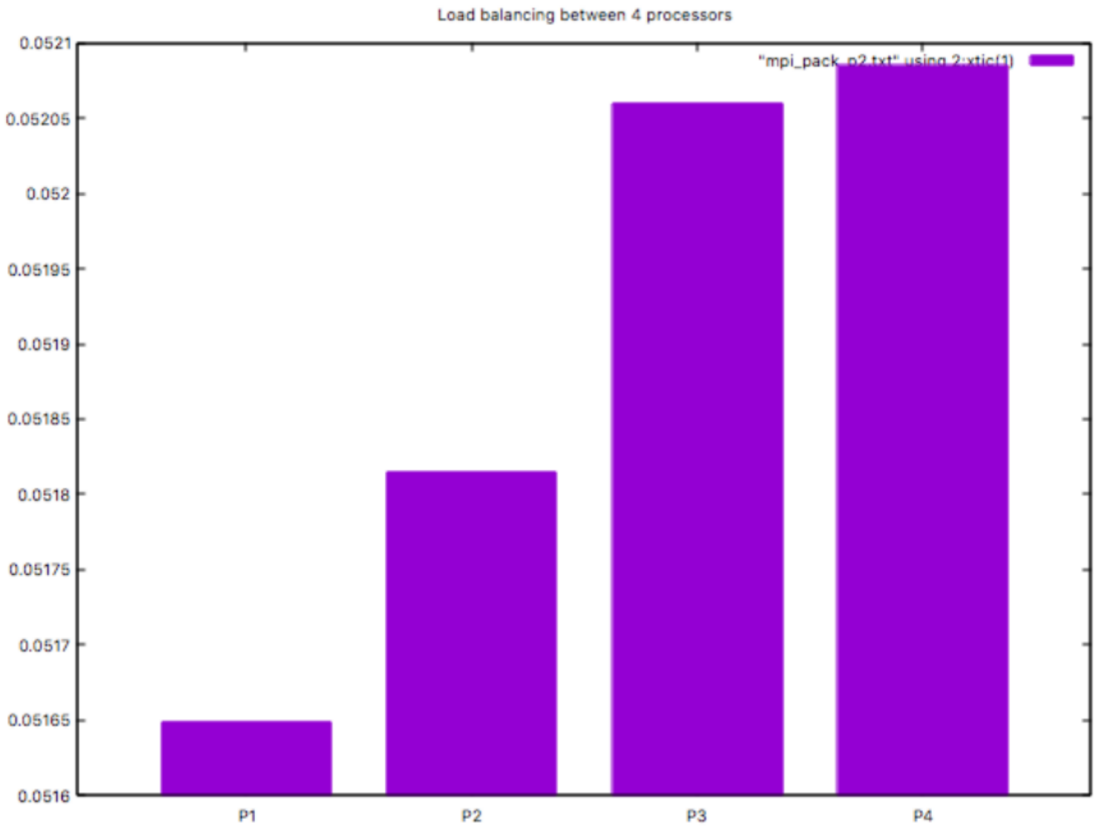

Figure 3.11: MPI pack Load balancing on 4 procs

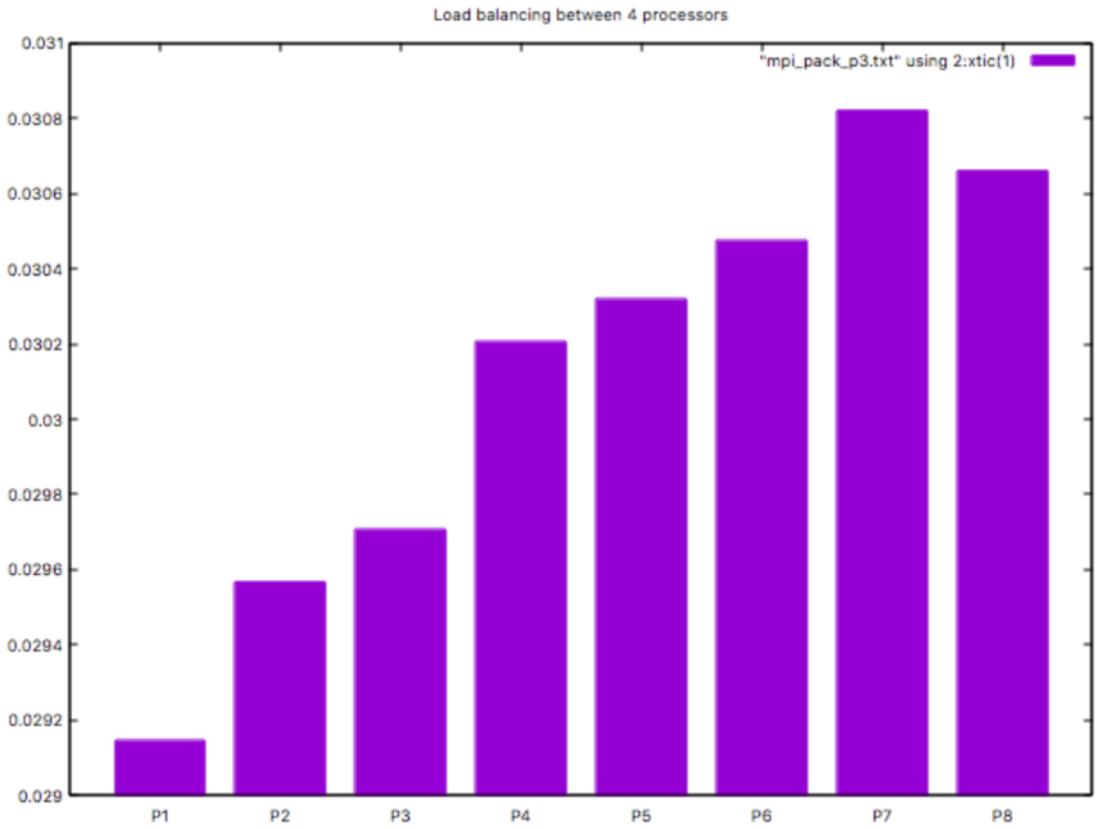

Figure 3.12: MPI pack Load balancing on 8 procs

4 OpenMP implementation

In this chapter we discuss the OPenMP parallelisation technique. In this method we have not modified the main function. We have only modified the compute function. The integer "pos" which was earlier used to append the array of linearised pixels is not useful now. We have to derive another way. So, to call the position of array, we have used the formula which is used to map matrix entries into the array sequentially row by row. Next, important thing in which variables are private to each thread and which are shared: l, c which are used in for loop should be private so that we are sure each thread calls different pixels, i need to be private, so that iteration counter is correct. a, b, x, y and tmp needs to be private so that the algorithm for each complex number c = a+ib is carried out separately. If any of these variable is shared algorithm will either escape early of the algorithm or will do more iterations than max. specified. Both static and dynamic scheduling speed ups are plotted below by increasing N. Also, a comparison with MPI has been plotted.

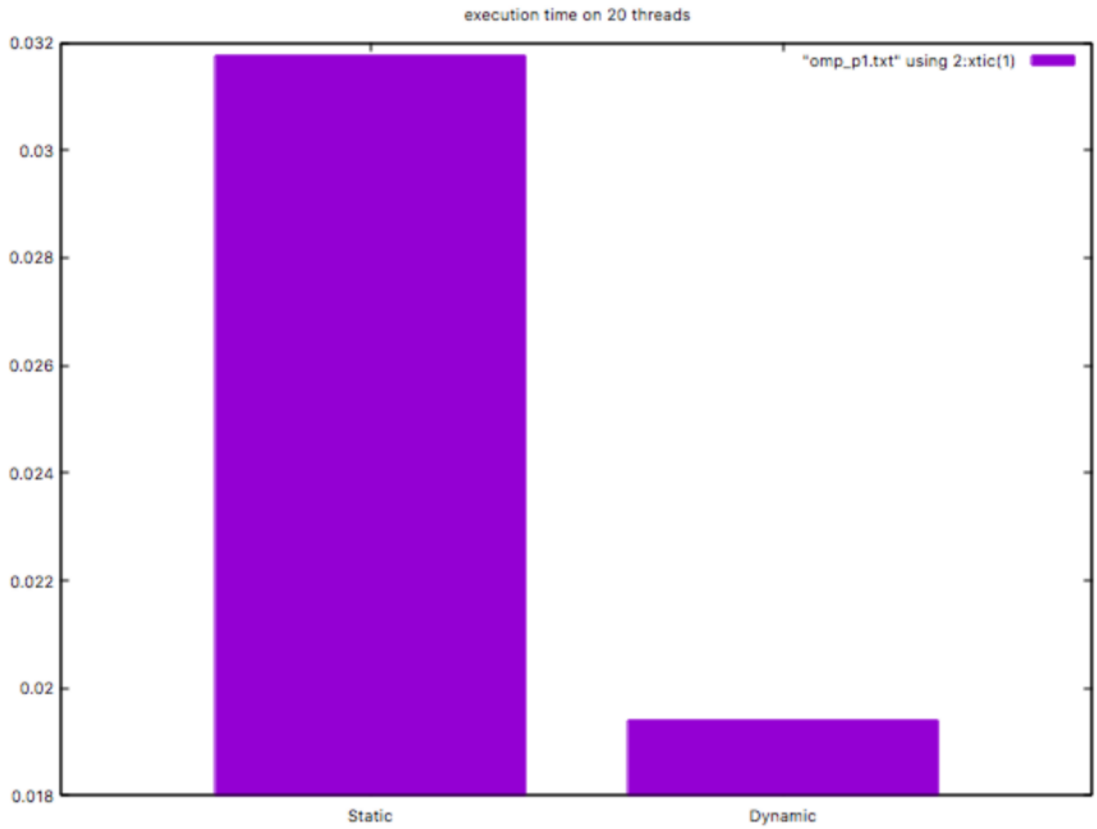

Figure 4.1: OMP execution time static vs dynamic scheduling

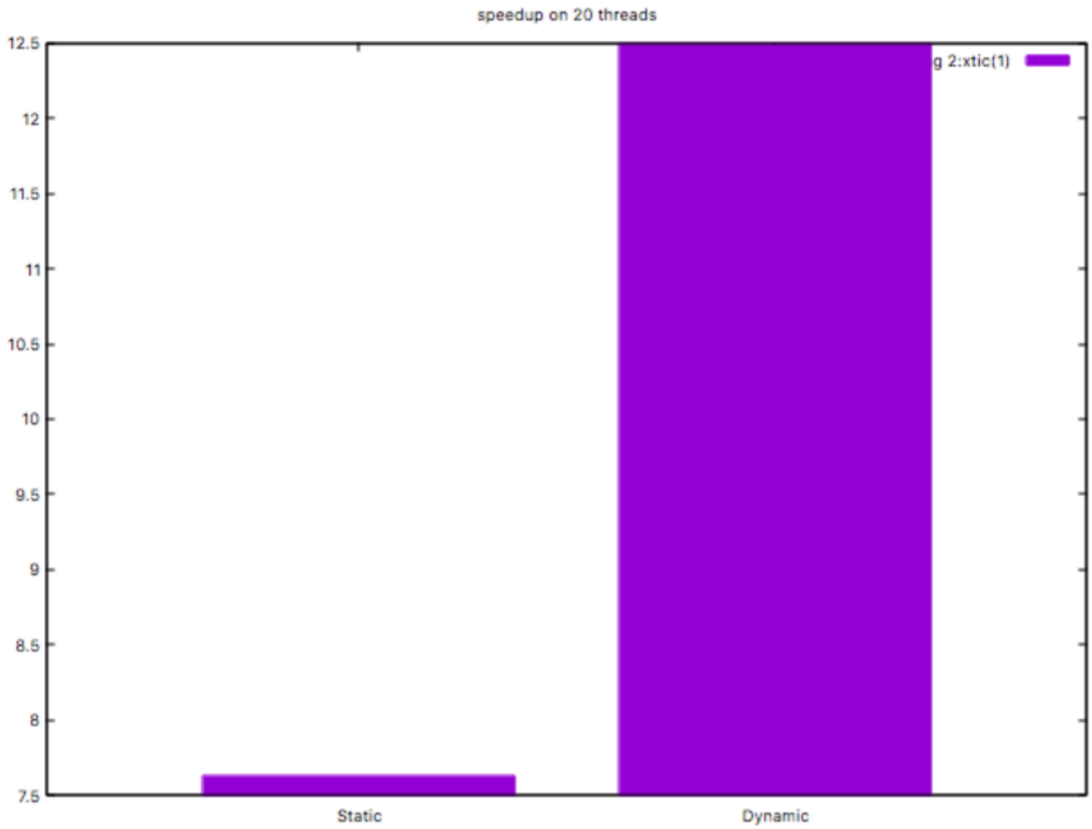

Figure 4.2: OMP speedup static vs dynamic scheduling

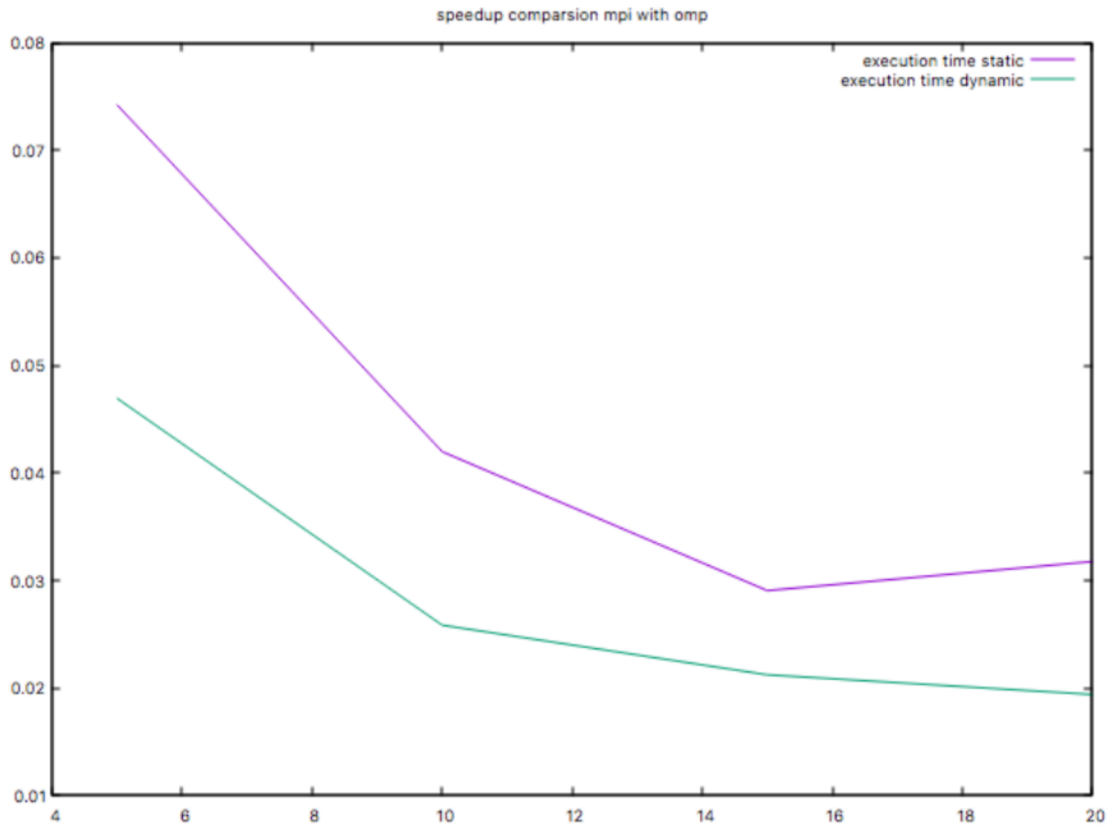

Figure 4.3: OMP max iter 100 execution time static vs dynamic scheduling

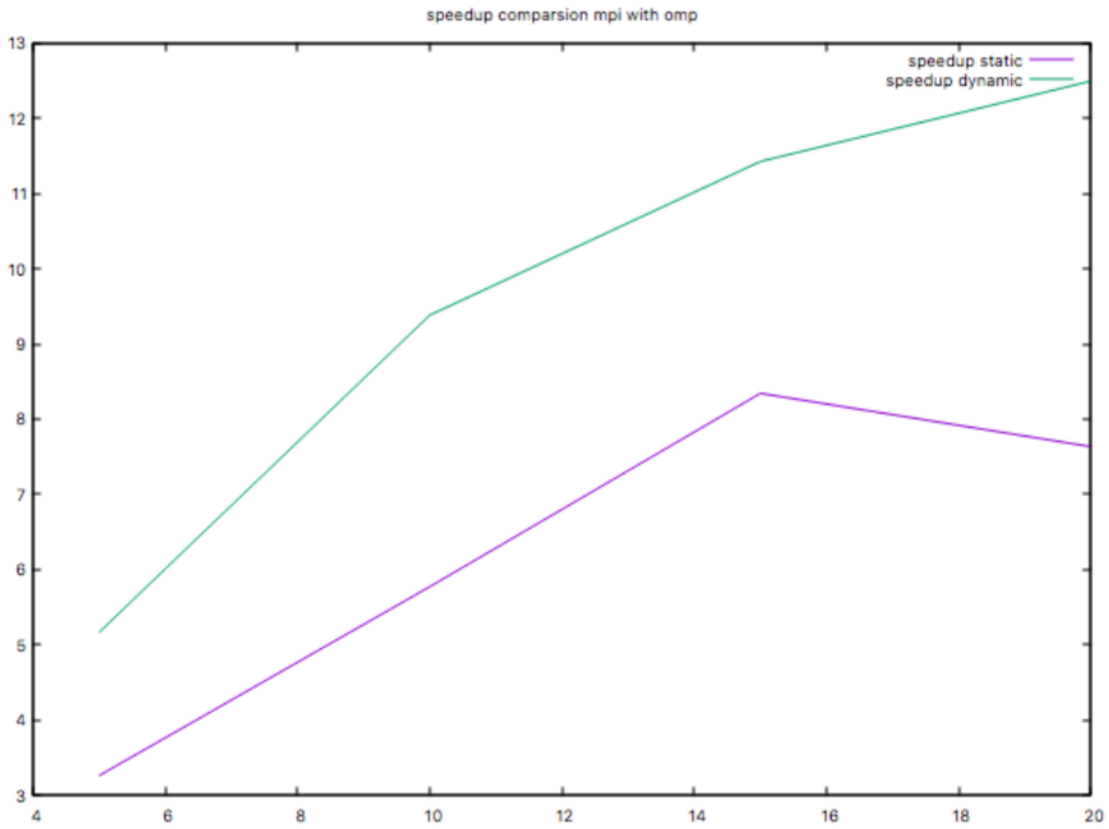

Figure 4.4: OMP speedup max iter 100 speedup static vs dynamic scheduling

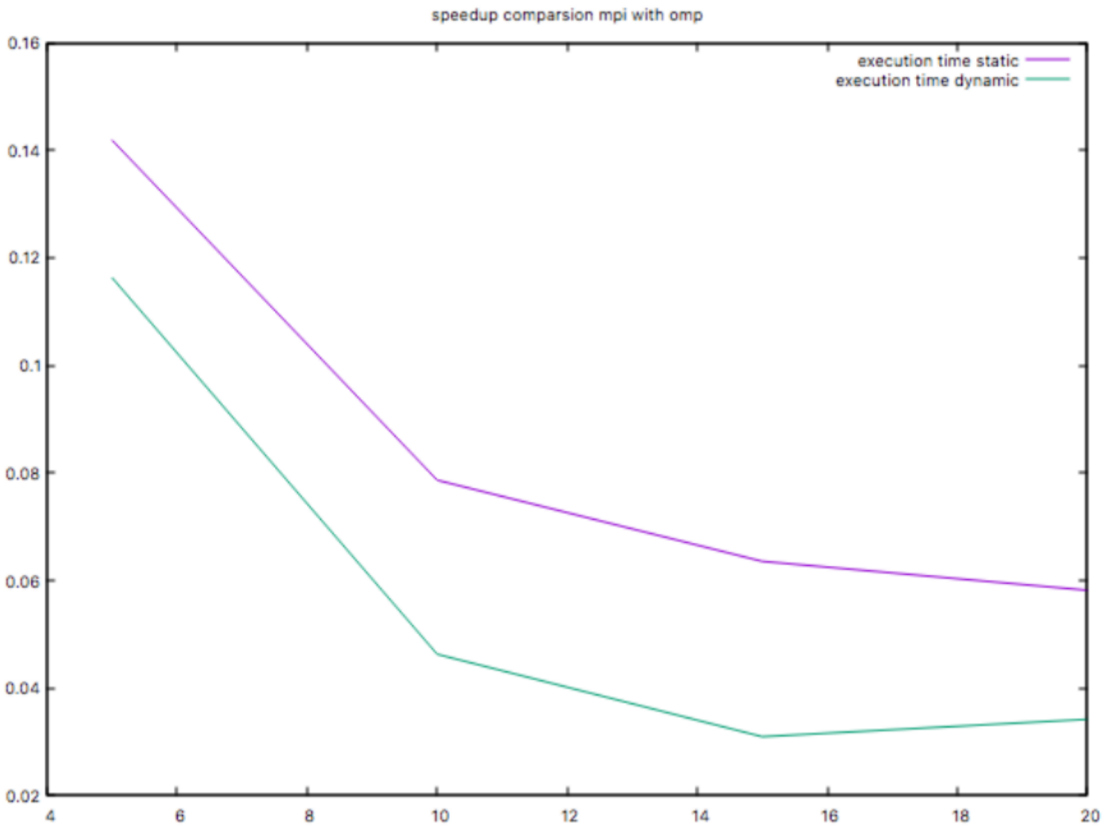

Figure 4.5: OMP max iter 200 execution time static vs dynamic scheduling

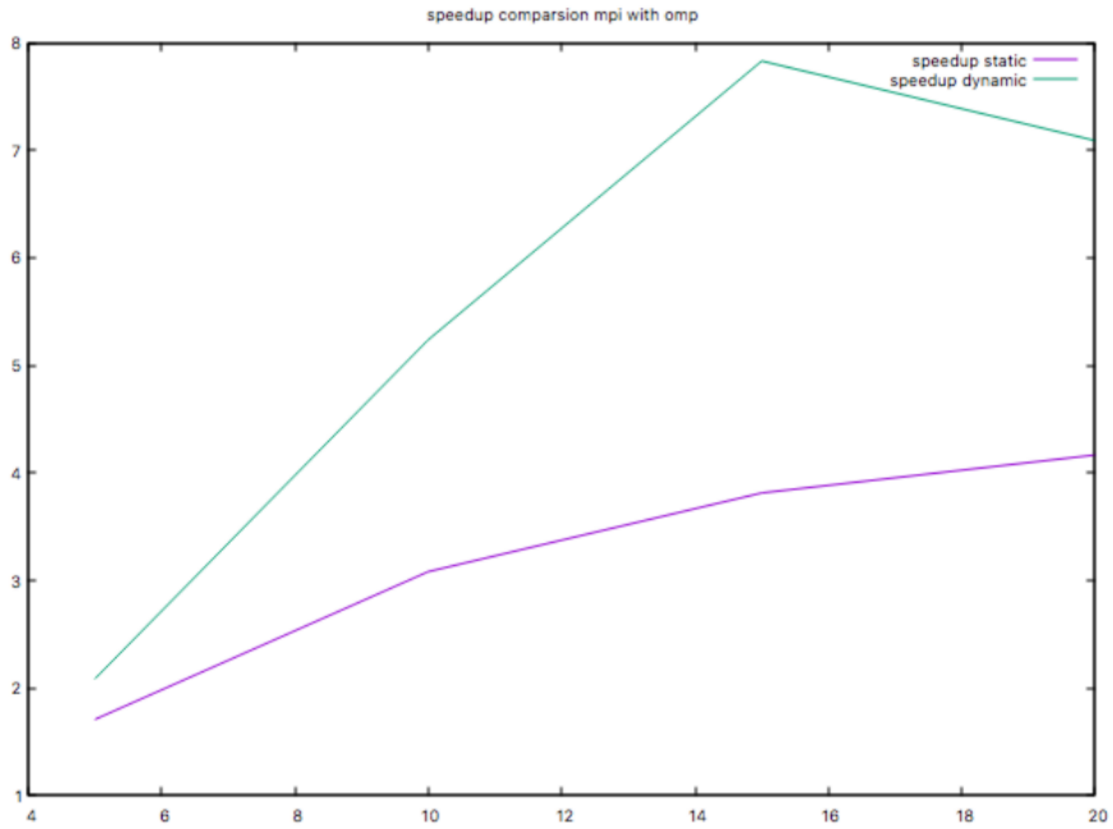

Figure 4.6: OMP speedup max iter 200 speedup static vs dynamic scheduling

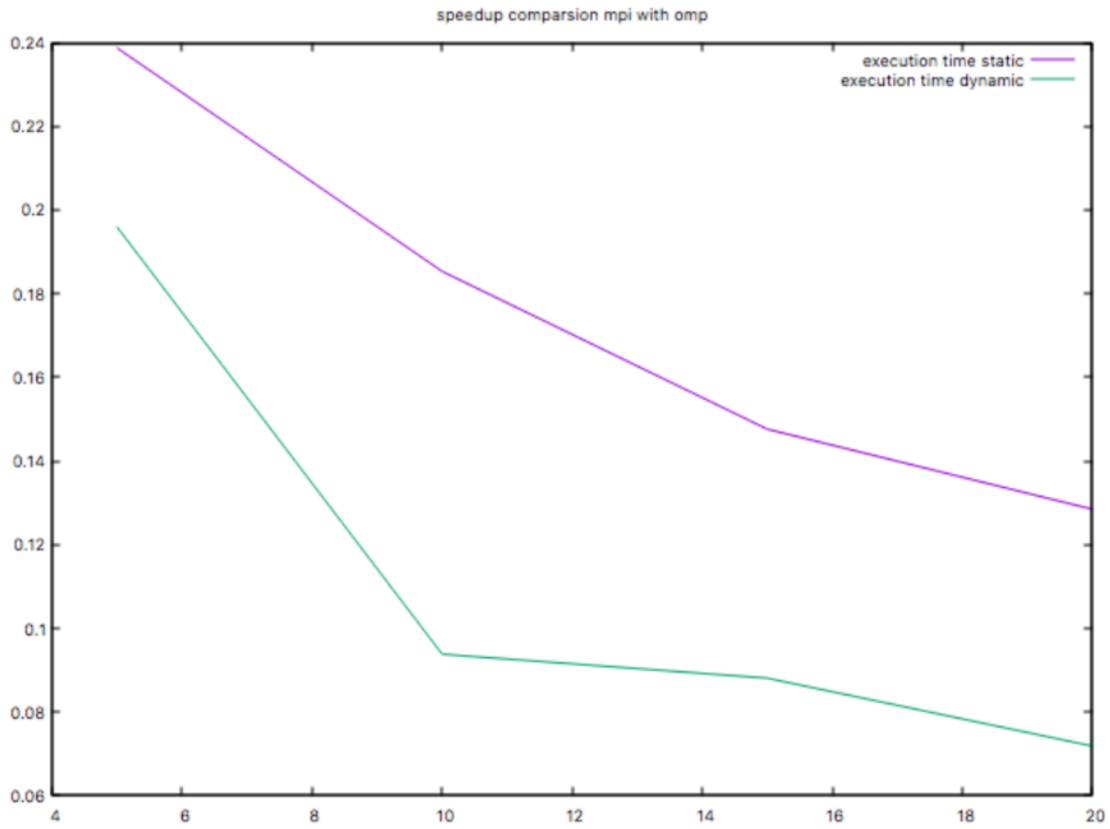

Figure 4.7: OMP max iter 500 execution time static vs dynamic scheduling

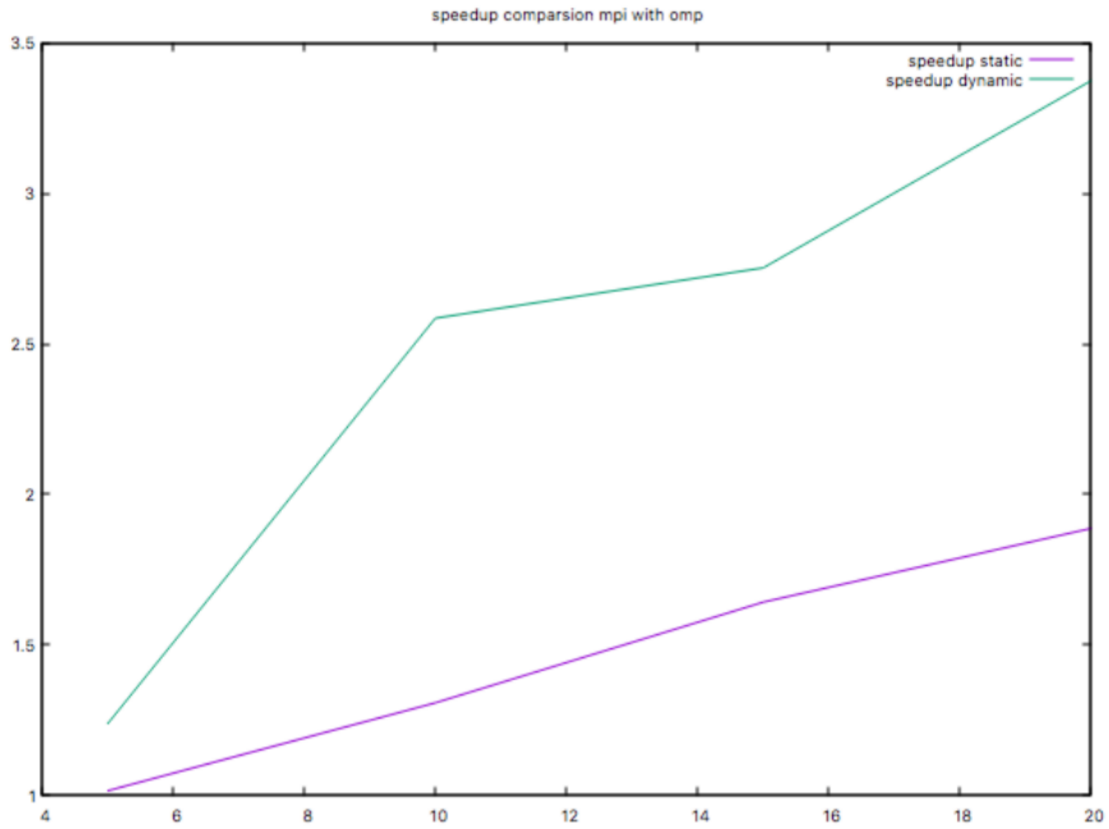

Figure 4.8: OMP OMP speedup max iter 500 speedup static vs dynamic scheduling

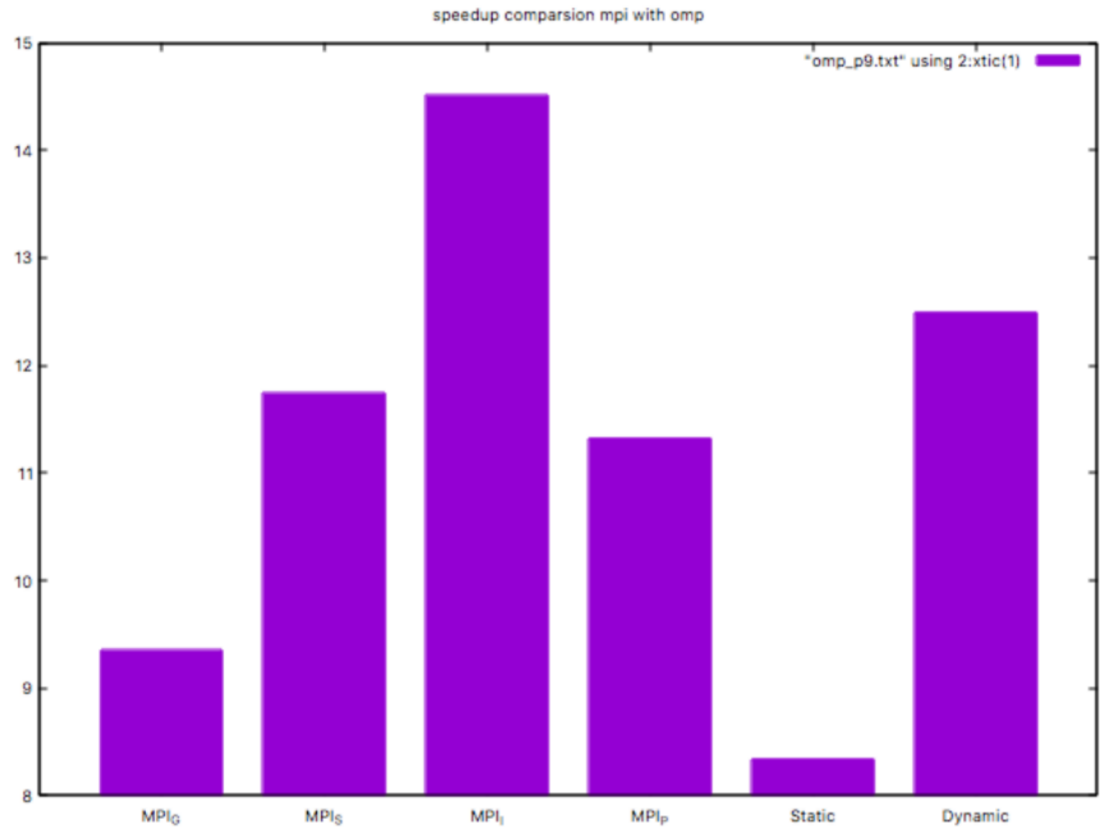

Figure 4.9: MPI vs OMP comparison

5 CUDA implementation

The acceleration based approach is carried out by Cuda. We have launched 16 blocks of 16 threads each to compute our results. The only change is in the compute function, which again replaces the "pos" variable with matrix to array mapping formula. First of all sizes are allocated using malloc function. After copying data from CPU to GPU Inside the compute function we launch a kernel to compute on GPU, after computation in GPU is completed data is copied back to CPU and stored in correct image location. We were not able to compute the time which gpu and cpu took and no time was produced even after lot of trying. therefore no analysis is possible. However the code works fine and the Mandelbrot image is produced correctly which can be inferred from the code.

6 Hybrid implementation

IN this chapter we discuss hybrid implementations. In hybrid approach we have implemented two approaches the MPI + OMP and MPI + Cuda approach.

6.1 MPI + OpenMP

In MPI-OMP approach we have used MPI Gather routine and used block partitioning technique: Inside each of those blocks we have launched threads which have same private and shared variables as the openMp alone case. We have presented a plot to compare the MPI-openMp approach based on static and dynamic scheduling for openMp.

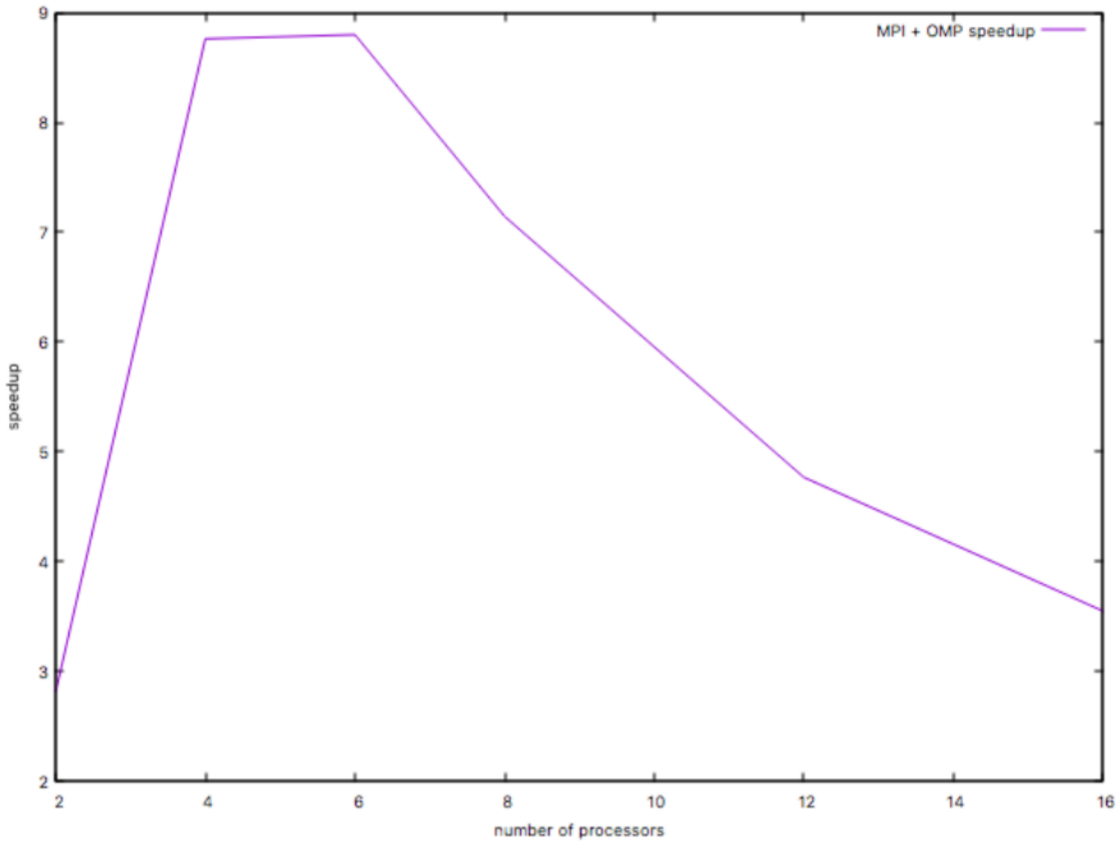

Figure 6.1: MPI + OMP speedup

7 Bibliography

- Lecture Slides of Prof. Melab
- Tutorial sheets of Dr. Gmys
- Class notes of Dr. Mouton