INDIAN INSTITUTE OF TECHNOLOGY GANDHINAGAR
ES 611: ALGORITHMS ON ADVANCED COMPUTER ARCHITECTURE

Report on Computational Project 5: Using MPI and CUDA to parallelize selected numerical algorithms

By Group 4:
Nishant Rao (11110059)
Parth Gudhka (11110062)
Ravi Kumar (11110081)
Shashank Heda (11110096)

ABSTRACT:
To use MPI and CUDA in parallelizing selected numerical algorithms for iterative numerical solution of the Poisson Equation and to estimate parallel performance metrics associated with the parallelization.

EXERCISES:

1. **WARM UP EXERCISES WITH SERIAL and MPI PARALLEL ITERATIVE SOLVERS**

(a) Study the code and its documentation within it, the convergence criteria set to terminate the solution process and run the code for arbitrary diagonally dominant **A** matrix and arbitrary **b** vector to get familiar with the code. Modify the code using MPI to implement the **parallel red-black Gauss-Seidel (GS)** method.

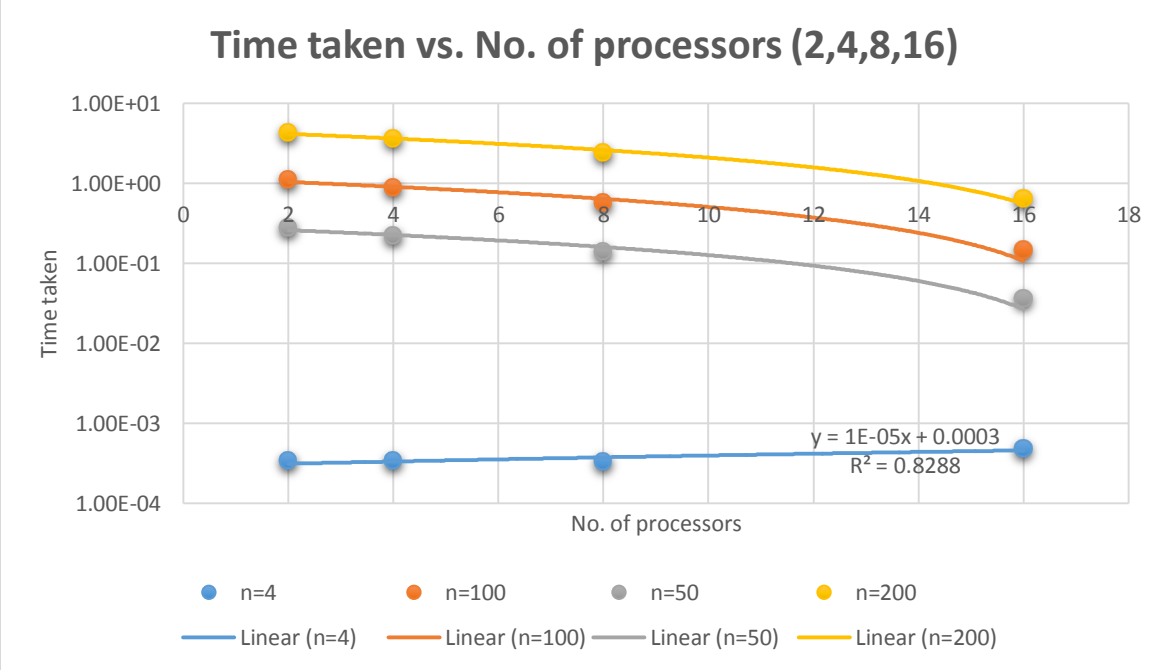


Figure 1: Time taken vs. No. of processors where n is order of matrix

Figure 1 shows the time taken vs. the number of processors for the different order of matrices, ranging from 4 to 200. N=4 is taken as a reference to see how increasing the number of processors leads to an increase in the time when the size of the problem is very small. Otherwise, the graph shows a very

interesting trend. The time keeps on decreasing as the number of processors is increased. This is as expected.

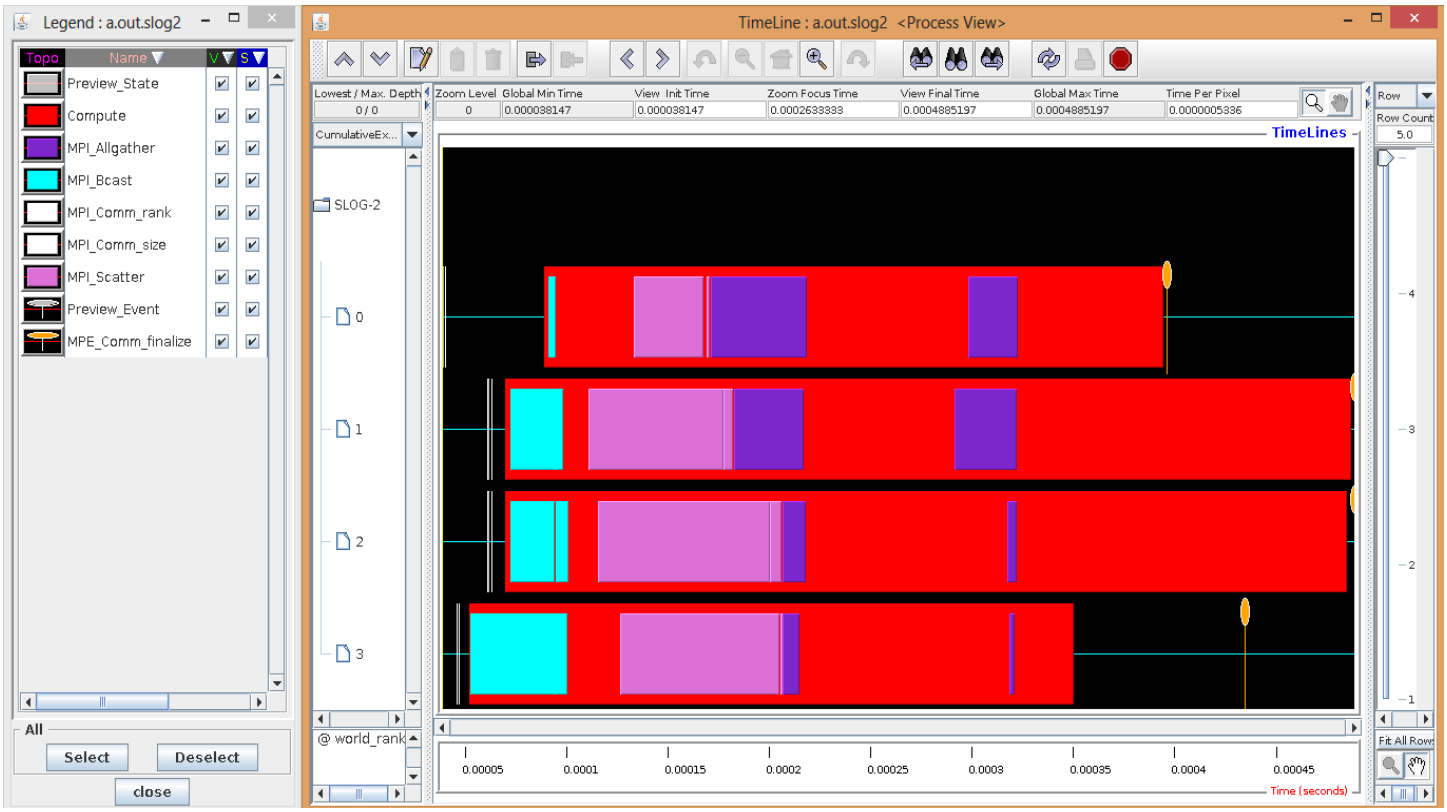


Figure 2: Jumpshot profile for the parallel red-black Gauss-Seidel method (n=4)

In figure 2, we can see the jumpshot profile for the parallel red-black Gauss Seidel method. It is quite interesting to note that the processors 1 and 2 take more time than the other processors. Also, a majority of the time is spent in the computation and the MPI_Allgather function of MPI.

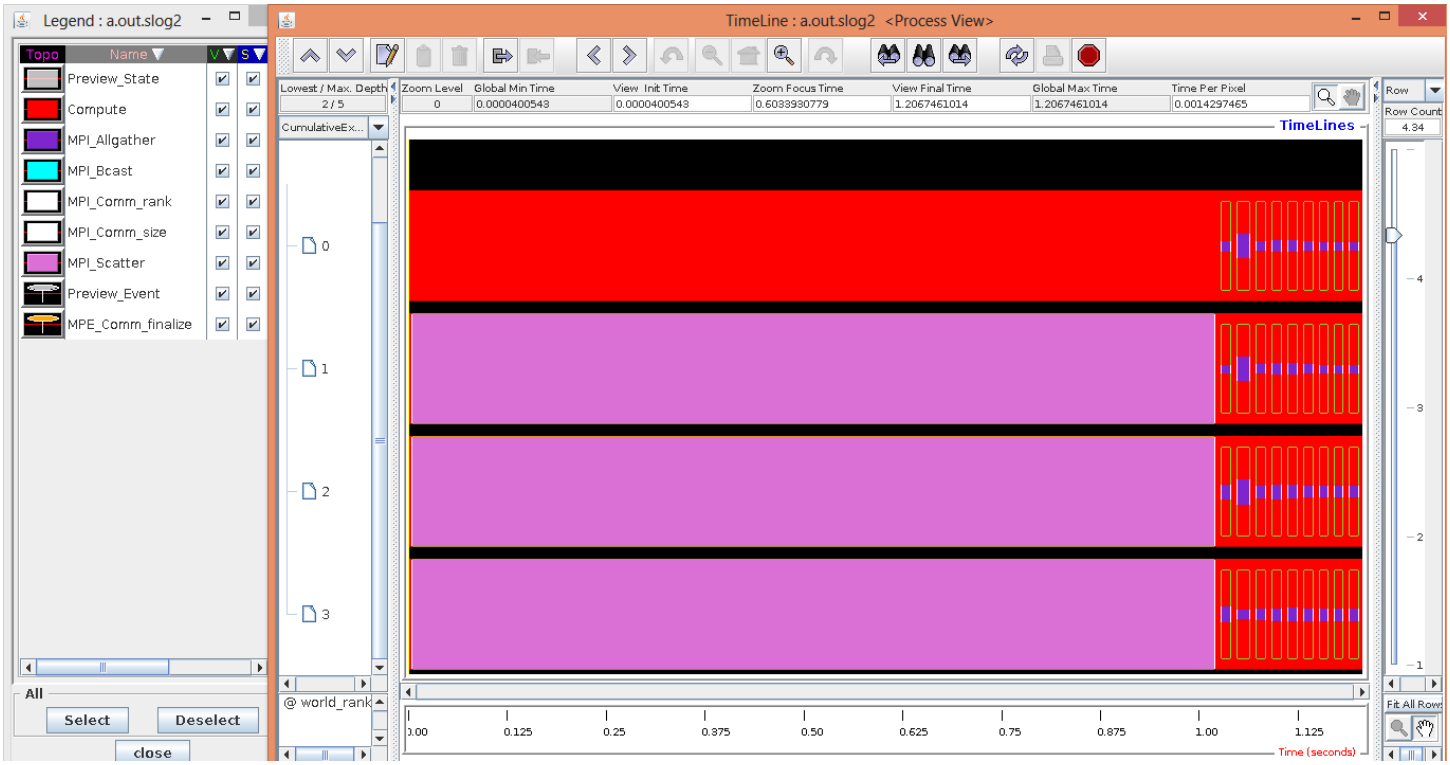


Figure 3: Jumpshot for 4 processors and n=100

Figure 3 shows the jumpshot profile for 4 processors and order of matrix equal to 100. Maximum of the time is spent in the MPI_Allgather function, which is also true for the previous jumpshot profile, but in this case it consumes more than half of the total time taken, and thus is a curious case.

CONCLUSIONS

Thus, we have implemented the Parallel red-black GS method in MPI, and observed that the time taken for execution is exactly as per expectation, i.e. as the no. of the processors increases, the time taken decreases uniformly, as can be seen in Figure 1. Thus, we can say that MPI is a very efficient implementer of the Gauss-Seidel algorithm.

MATLAB PROGRAMS FROM COMPUTATIONAL PROJECT 6

Initially, we tried implementing MATLAB softwares for small scale things like parallelising a small section of a code, parallelising 'for' loop etc.

A plot below shows the timings observed for a serial code and parallel code based on parallelising a 'for' loop simly by using 'parfor' in PMATLAB, much similar to what is normally done in OpenMP.

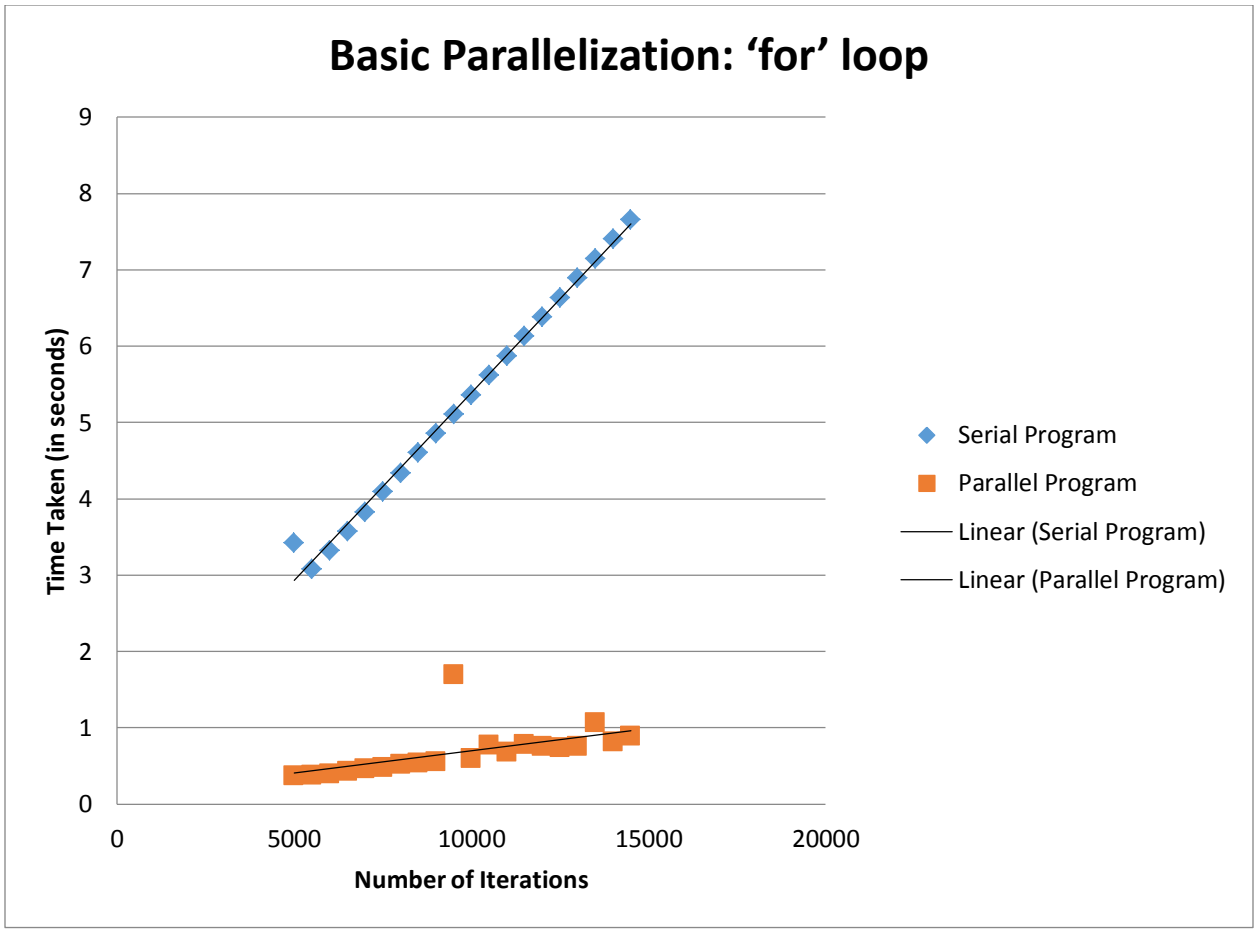


Fig a Change in time with change in number of iterations (for loop)

- The program control in such programs can be achieved using 'matlabpool' through which we can fix the number of cores/ workers/labs to be used for parallelising a program using CPU processors
- The syntax for parallelising a for loop is 'parfor <operation to be performed>'

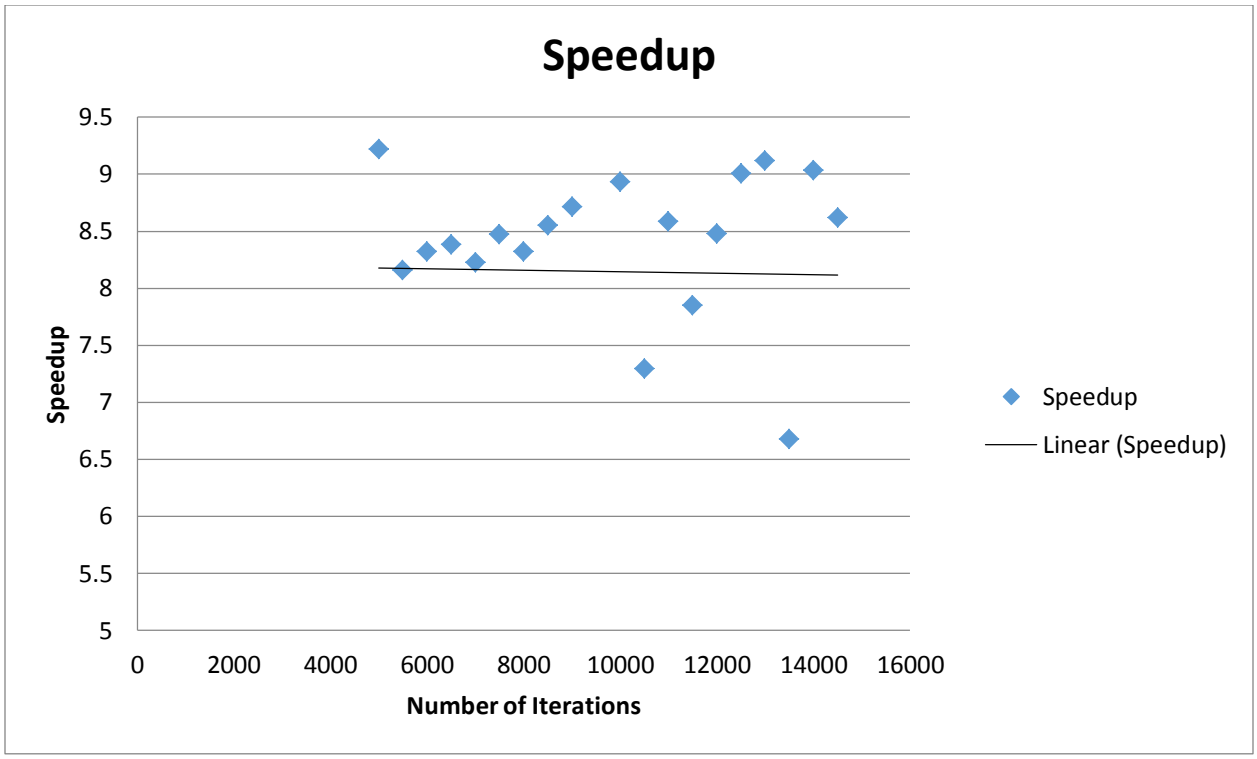
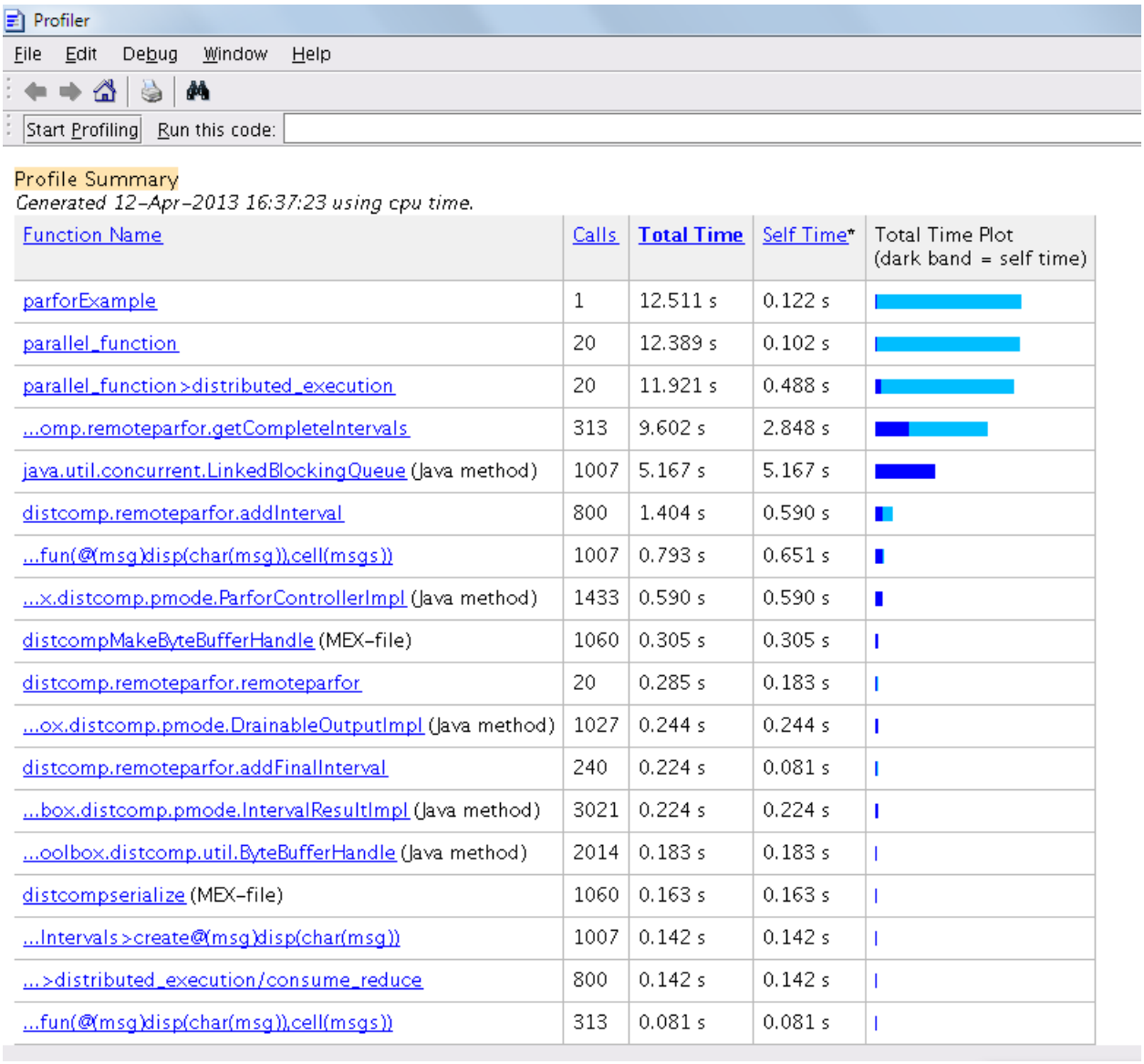


Fig b Change in Speedup with change in number of iterations ('for' loop)

- Figure b shows that speedup on CPU cores remains almost constant as we increase the number of iterations.
- Figure c shows a PMATLAB Profiler for the same:



Function Name	Calls	Total Time	Self Time*	Total Time Plot (dark band = self time)
parforExample	1	12.511 s	0.122 s	
parallel_function	20	12.389 s	0.102 s	
parallel_function>distributed_execution	20	11.921 s	0.488 s	
...omp.remoteparfor.getCompleteIntervals	313	9.602 s	2.848 s	
java.util.concurrent.LinkedBlockingQueue (Java method)	1007	5.167 s	5.167 s	
distcomp.remoteparfor.addInterval	800	1.404 s	0.590 s	
...fun(@(msg)disp(char(msg)),cell(msgs))	1007	0.793 s	0.651 s	
...x.distcomp.pmode.ParforControllerImpl (Java method)	1433	0.590 s	0.590 s	
distcompMakeByteBufferHandle (MEX-file)	1060	0.305 s	0.305 s	
distcomp.remoteparfor.remoteparfor	20	0.285 s	0.183 s	
...ox.distcomp.pmode.DrainableOutputImpl (Java method)	1027	0.244 s	0.244 s	
distcomp.remoteparfor.addFinalInterval	240	0.224 s	0.081 s	
...box.distcomp.pmode.IntervalResultImpl (Java method)	3021	0.224 s	0.224 s	
...oolbox.distcomp.util.ByteBufferHandle (Java method)	2014	0.183 s	0.183 s	
distcompserialize (MEX-file)	1060	0.163 s	0.163 s	
...Intervals>create@(msg)disp(char(msg))	1007	0.142 s	0.142 s	
...>distributed_execution/consume_reduce	800	0.142 s	0.142 s	
...fun(@(msg)disp(char(msg)),cell(msgs))	313	0.081 s	0.081 s	

Fig c Parallel MATLAB Profiler

1. Matlab_without_gpu.m

   The first section of this code defines the size/resolution of the Mandelbrot figure to be generated in terms of pixels (for example 1000x1000 grid). This is then converted to a set of complex numbers. Then a complex number matrix z is created which generated at each point of the grid corresponding to its coordinates. Another unit matrix/grid (of size 1000x1000) *'count'* is then created for storing color values at all points.

$$z = z_0 = \begin{bmatrix} 1+i1 & \cdots & 1+i1000 \\ \vdots & \ddots & \vdots \\ 1000+i1 & \cdots & 1000+i1000 \end{bmatrix}$$

$$count = \begin{bmatrix} 1 & \cdots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \cdots & 1 \end{bmatrix}$$

The second section calculates the color value at each location of the matrix *count*. The way it does is that it performs the action $z = z.* z + z_0$ for *maxIteration* number of times. The matrix z is updated (which means that all the elements of z are updated) after each iteration. The absolute value of each element of z is then compared with 2. If the value is less than 2 then the corresponding element in *count* is incremented by one.
Finally the logarithm of *count* is taken (just to get significant difference in colors) and is then plotted as an image.

2. arrayfun.m
   This function performs the same operation as that of serial, but it uses GPUs instead. All the declarations are done in the setup part of the function. z and $count$ matrices are the same but the type of operation is different.

   Element-wise Operation:
   The algorithm is operating equally on every element of the input, we can place the code in a helper function and call it using arrayfun. For GPU array inputs, the function used with arrayfun gets compiled into native GPU code. In this case the loop was placed in pctdemo_processMandelbrotElement.m:

```
function count= pctdemo_processMandelbrotElement (x0,y0,
maxIterations)
z0 = complex(x0,y0);
z = z0;
count = 1;
while (count <= maxIterations) && (abs(z) <= 2)
    count = count + 1;
    z = z*z + z0;
end
count = log(count);
```

   An early abort has been introduced because this function processes only a single element. For most views of the Mandelbrot Set a significant number of elements stop very early and this can save a lot of processing. The for loop has also been replaced by a while loop because they are usually more efficient. This function makes no mention of the GPU and uses no GPU-specific features - it is standard MATLAB code.

   Using arrayfun means that instead of many thousands of calls to separate GPU-optimized operations (at least 6 per iteration), we make one call to a parallelized GPU operation that performs the whole calculation. This significantly reduces overhead.

3. Matlab_with_ CUDA.m

Improved performance is achieved by converting the basic algorithm to a C-Mex function. Parallel Computing Toolbox can be used to call pre-written CUDA kernels using MATLAB data with the help of C/C++. This can be done with the parallel.gpu.CUDAKernelparallel.gpu.CUDAKernel feature.

The CUDA/C++ code is a little more involved than the MATLAB versions, due to the lack of complex numbers in C++. However, the algorithm is unchanged:

```
__device__
unsigned int doIterations( double const realPart0,
                           double const imagPart0,
                           unsigned int const maxIters ) {
   // Initialize: z = z0
   double realPart = realPart0;
   double imagPart = imagPart0;
   unsigned int count = 0;
   // Loop until escape
   while ( ( count <= maxIters )
          && ((realPart*realPart + imagPart*imagPart) <= 4.0) ) {
      ++count;
      // Update: z = z*z + z0;
      double const oldRealPart = realPart;
      realPart   =   realPart*realPart   -   imagPart*imagPart   +
realPart0;
      imagPart = 2.0*oldRealPart*imagPart + imagPart0;
   }
   return count;
}
```

One GPU thread is required for location in the Mandelbrot Set, with the threads grouped into blocks. The kernel indicates how big a thread-block is, and the code given in the attached file is used to calculate the number of thread-blocks required. This then becomes the GridSize.
The kernel is then called with the appropriate GridSize, it performs the calculations and then the result is fetched back from the GPU kernel.

ANALYSIS OF EXECUTION TIME

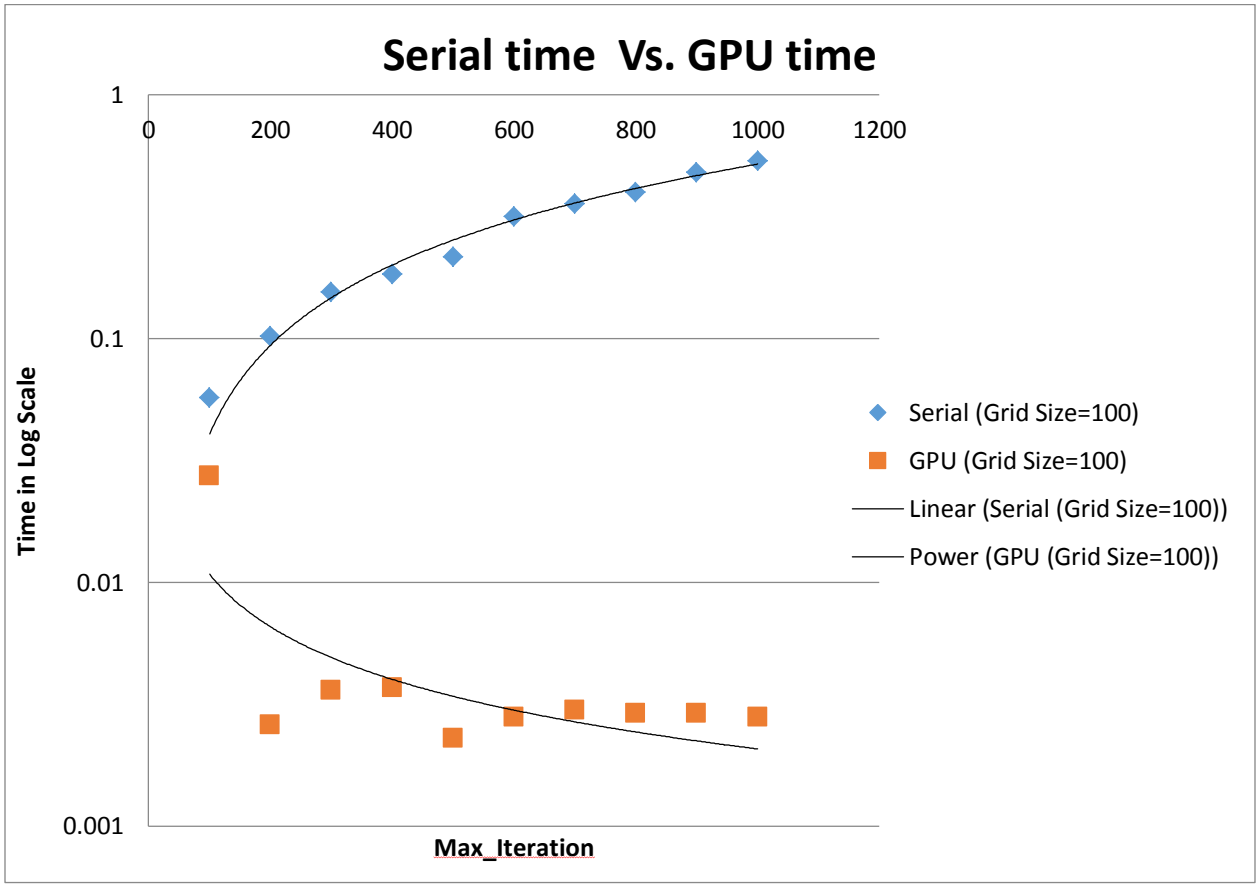


Figure 2: Time taken vs. Max_iteration for a serial code and on Parallel MATLAB

- In Figure 1, it can be seen that the parallel code is about a 100 times faster than the serial code. This is because the parallelization done with the GPU is so efficient in the generation of the Mandelbrot set, that we get such amazing speedups with the GPU.

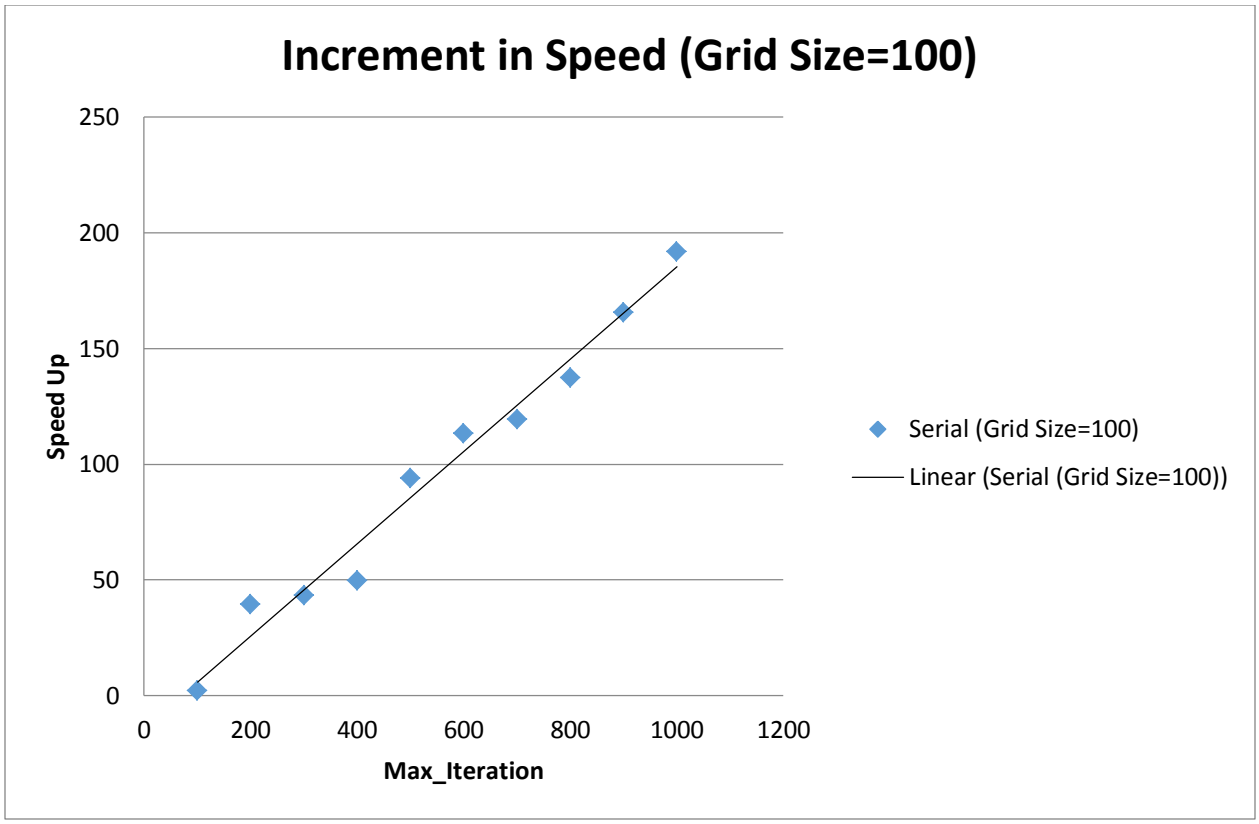


Figure 3: Speedup vs. Max_Iteration for a grid size of 100.

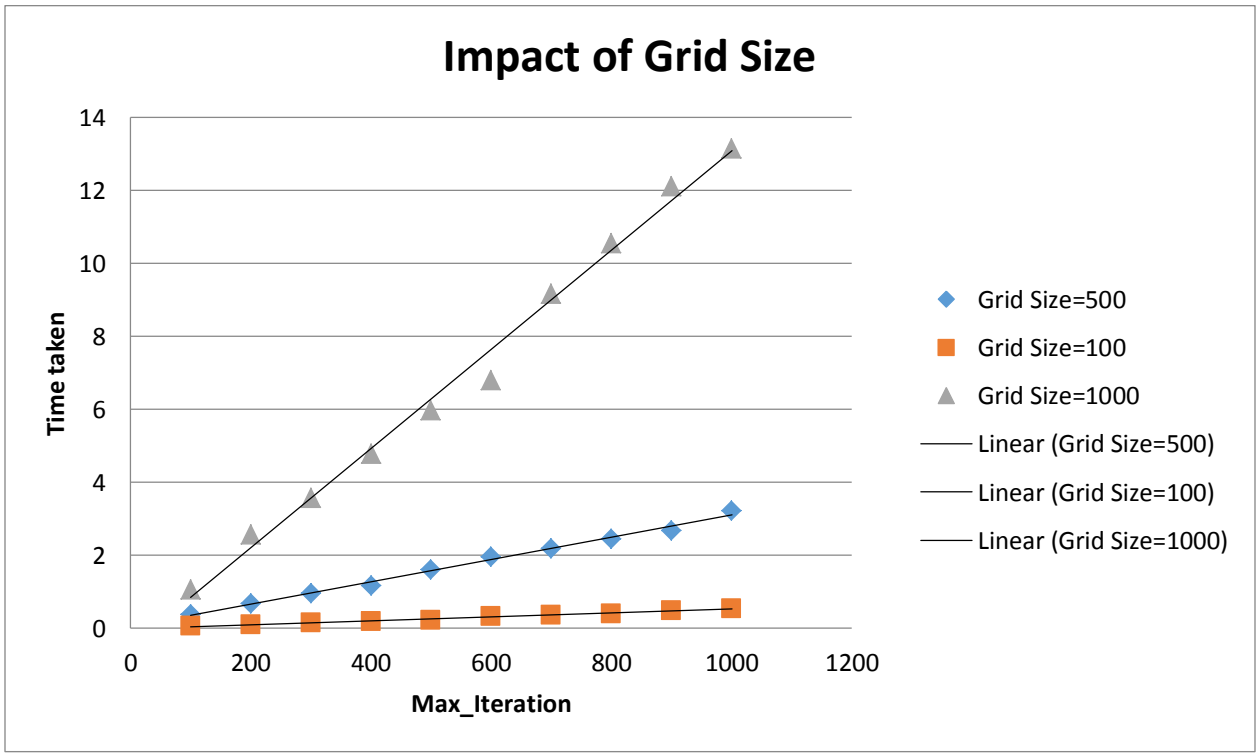


Figure 4: Time taken vs. maximum iterations for different grid sizes

- Figure 3 shows the variation of execution time with change in the number of maximum iterations. We observe a very uniform increase in the time, as expected. Also, as the grid size is increased, the time taken increases. The variation in the data from the linear behavior is very small, and shows the results are obtained as per expectations.

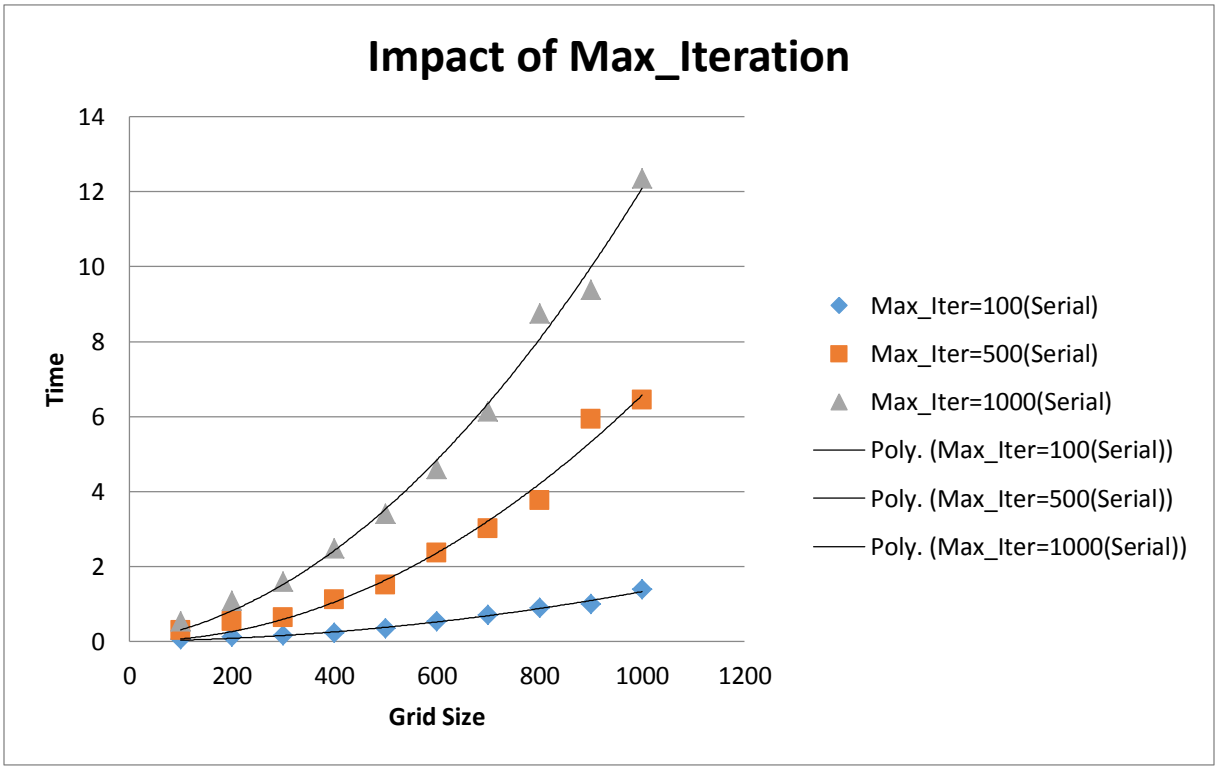


Figure 5: Time taken vs. Grid size for different maximum iterations (serial)

- Figure 4 is just a modified graph of figure 3, but the focus has been shifted to the effect of increasing the grid size on the time taken on the CPU. It can be seen that as the grid size is increased, the there is a 2^{nd} order increase in the time taken. This is perfectly as per expectation, because on increasing the grid size from 100 to 200, the number of elements increase from 10000 to 40000, i.e. on a two-fold increase in the grid size, the computation cost increases four times. So, these results are as per expectations. We also see that as the maximum iterations are increased, there is an increase in the time, but its effect is not as large as the grid sizes effect i.e. increasing the grid size increases the time taken linearly, as opposed to a second order increase in the time taken when the grid size is increased.

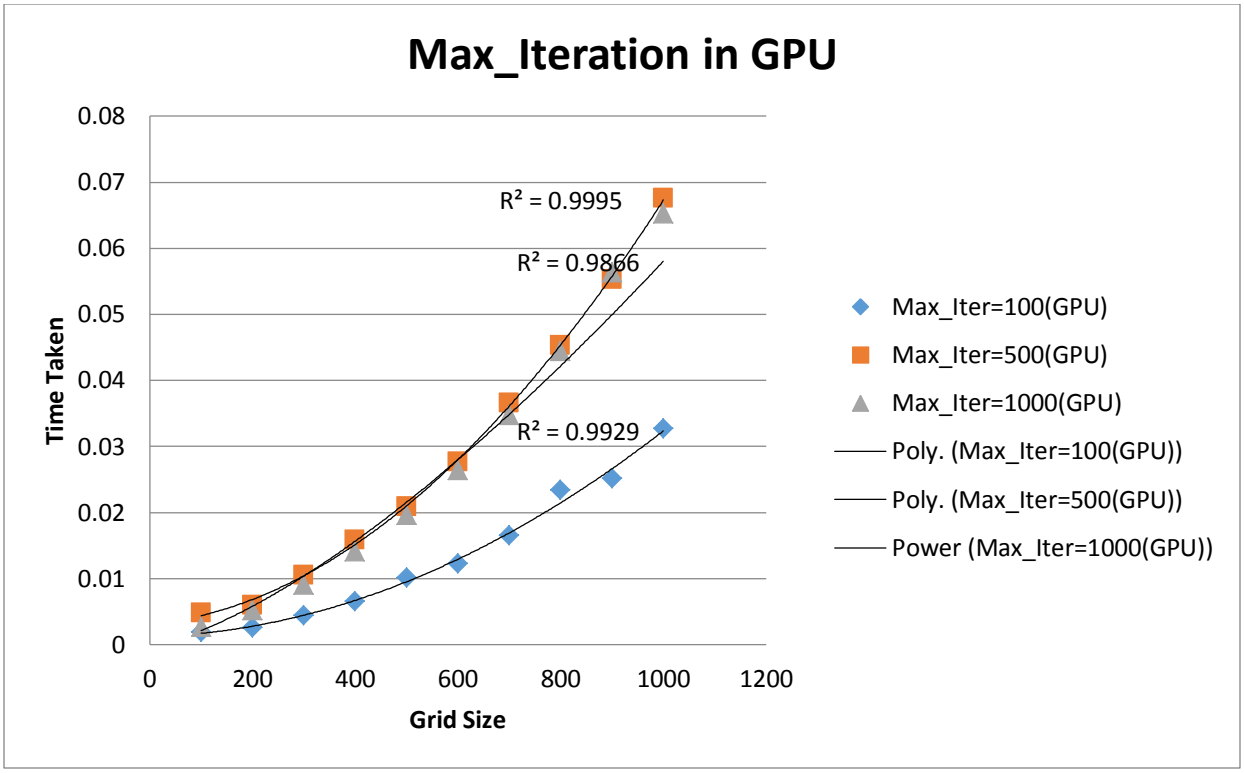


Figure 6: Time taken vs. Grid size for different maximum iterations (GPU)

- Figure 5 shows how the time taken for execution on the GPU varies with the increase in the grid size, as well as how varying the maximum iterations affects the time taken. Similar to the execution of the serial code on the CPU, the time taken has a second degree relation with the grid size, which is expected as explained in the earlier section. What is more interesting however is that the time taken for maximum iterations equal to 500 and 1000 is almost the same, and in some cases, it is even lower for n=1000. This can be a random error, though quite large. However, a very interesting point is that the time taken for execution on the GPU is almost a hundred times less than that on the CPU. This is an enormous speedup, and is explored in Figures 6 and 7.

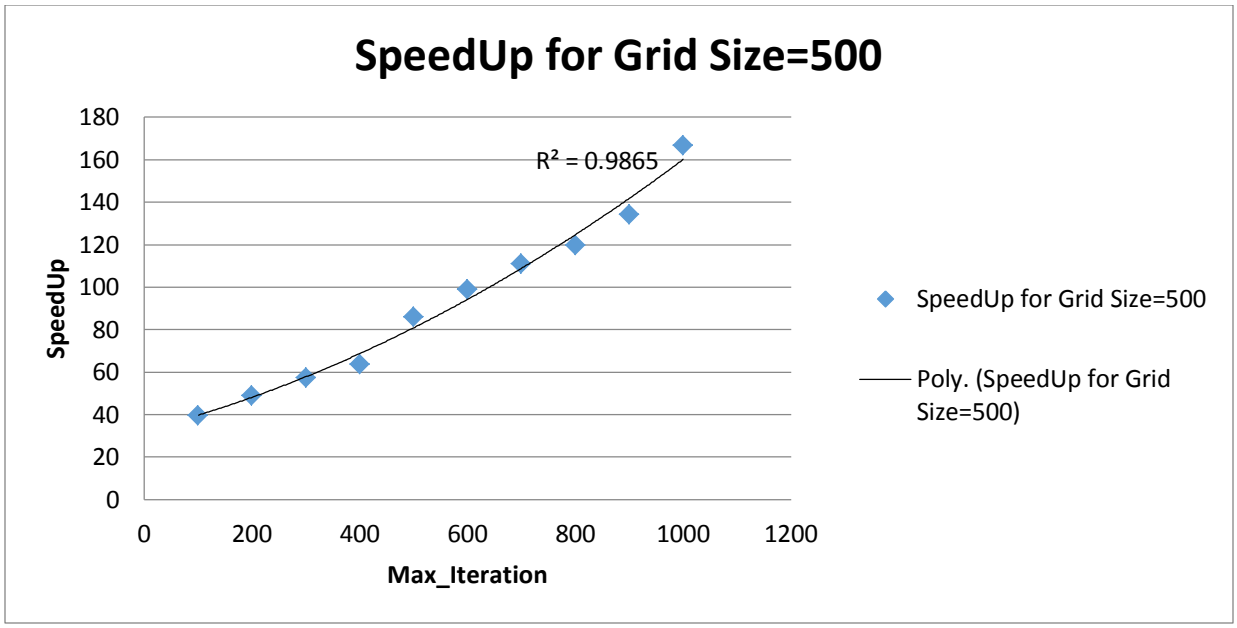


Figure 7: Speedup of the GPU code as compared to the serial code for fixed Grid size

- Figure 6 depicts an increase in speedup with increase in Max_Iteration when resolution is kept at 500 x 500. This phenomenon would not have been observed had it been done using CPU Cores or even a normal CUDA Program.
- The increase in speedup with change in Max_Iteration may be attributed to the intelligent distribution of work by PARALLEL MATLAB to the GPUs. It calculates the optimum number of GPU Threads required for such computation. Thus, as the value of Max_iteration is increased, it uses more number of threads with each increasing value, thus resulting into an overall increase in speedup.

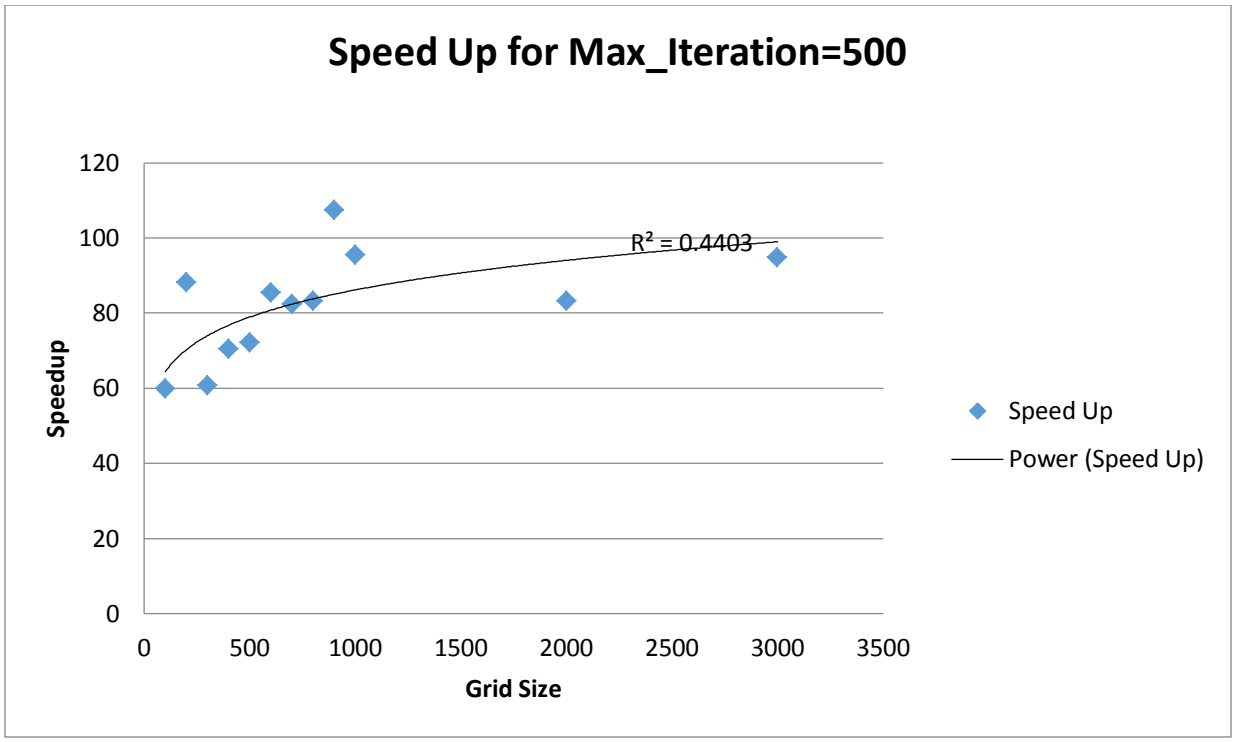


Figure 8: Speedup of the GPU code as compared to the serial code for fixed maximum iterations

- Figure 7 depicts an increase in speedup initially with increase in the width of the window (resolution) but speedup is found to get saturated as the Grid Size is further increased with Max_Iteration being kept at 500.
- The increase in speedup with change in Grid Size may be attributed to the intelligent distribution of work by PARALLEL MATLAB to the GPUs. It calculates the optimum number of GPU Threads required for such computation. Thus, as the value of Grid Size is increased, it uses more number of threads with each increasing value, thus resulting into an overall increase in speedup. But if the Grid Size is increased above 2000, it may be noticed that the speedup starts getting saturated which is similar to what we observe based on Amdahl's law.

CONCLUSIONS

Part-1: Thus, we have implemented the Parallel red-black Gauss Siedel method in MPI, and observed that the time taken for execution is exactly as per expectation, i.e. as the no. of the processors increases, the time taken decreases uniformly, as can be seen in Figure 1. Thus, we can say that MPI is a very efficient implementer of the Gauss-Seidel algorithm.

Part-2: The Parallel MATLAB is an excellent software with a perspective of its applicability in Electrical engineering. Especially, it becomes quite a lot computation friendly when specific areas like signal processing and image processing are explored using the Parallel MATLAB construct. We look forward to use this software for our future courses in the Electrical Engineering.

ACKNOWLEDGEMENT:

We, members of Group4, have taken efforts in this project. However, it would not have been possible without the kind support and help we received from Prof. Murali Damodaran. We would like to extend my sincere thanks to him. We are highly indebted to his guidance and constant supervision as well as for providing necessary information regarding the project & also for his support in completing the project. My thanks and appreciations also go to my friends of other groups with whom we constantly interacted and countered the problems we had in developing the project problems and people who have willingly helped us out with their abilities.