# OPEN MP PROGRAMMING: PROJECT ASSIGNMENT 3

Siddhartha Srinadhuni

9508286854

sisr16@student.bth.se

Madhukar Enugurthi

9506077537

maen16@student.bth.se

## I. INTRODUCTION:

Setting the goals as Standardization, Ease of use and portability for most major platforms, Open Multi-Processing is used to transform a sequential program to a parallel counter-part. Open MP is defined as an application programming interface (API) that supports most of the shared-memory multi-processing in various operating systems and processor architectures [1]. This yields in an enhanced performance when implemented on a multi-core system. Compiler directives, various library functions and environment variables allow the conversion of a sequential program to a parallel one. These directives have been structured differently for different programming languages (like C/C++ and FORTRAN). Further, the compiler directives can be used as extensions for the above sequential programming languages [2]. The following document is structured such that it describes our implementation of quick sort and Gaussian elimination in parallel implementations.

## II. QUICK SORT:

The logic behind the quick sort being an efficient sorting program is when given a random set of numbers are to be ordered in ascending manner. This also being a divide and conquer algorithm has the following steps involved.

- Picking an element from the array and labelling it to be a pivot element.
- Letting pivot be the pointer and positioning the element which value less than the pivot before and elements that value higher are positioned after the pivot.
- After this partition, the pivot is in its final position.
- This partition operation is applied recursively onto the subdivided arrays.
- This process is repeated until the elements are sorted in the way they are supposed to be.

**Parallel Implementation of QuickSort:**

- *initializeArray()* function is called when the array *ar[]* is to be initialized and yet to be sorted. This generates an array of random numbers.
- *parallel_qsort()*function makes the array undergo sorting.
- Using the directives *#pragma omp parallel* and *#pragma omp single* for making sure that the functions are executed by a single thread.
- The *parallel_qsort()* function sorts the array in the following way.
- Letting the element having the highest index be the pivot for the array.
- Comparisons starting from the lower index are done with the pivot in order to check if that particular element is less than that of the pivot.
- This is repeated until reaching the highest index value is reached. If the element is found to be lower, then the variable ‘low’ is incremented and the current element is swapped with *a[low]*.

- These comparisons are done in order to create two sub arrays out of the divided array.
- One sub array carrying elements greater than the pivot and the other sub array carrying the elements lesser than the pivot.
- Sorting these arrays in parallel using the directives *#pragma omp task* in which the divided subarrays call *parallel_qsort()* function individually.
- This sorting of sub arrays finally yields out a sorted array and is the output generated.
- The function *print_Array()* prints the desired output.

**III.**GAUSSIAN ELIMINATION:

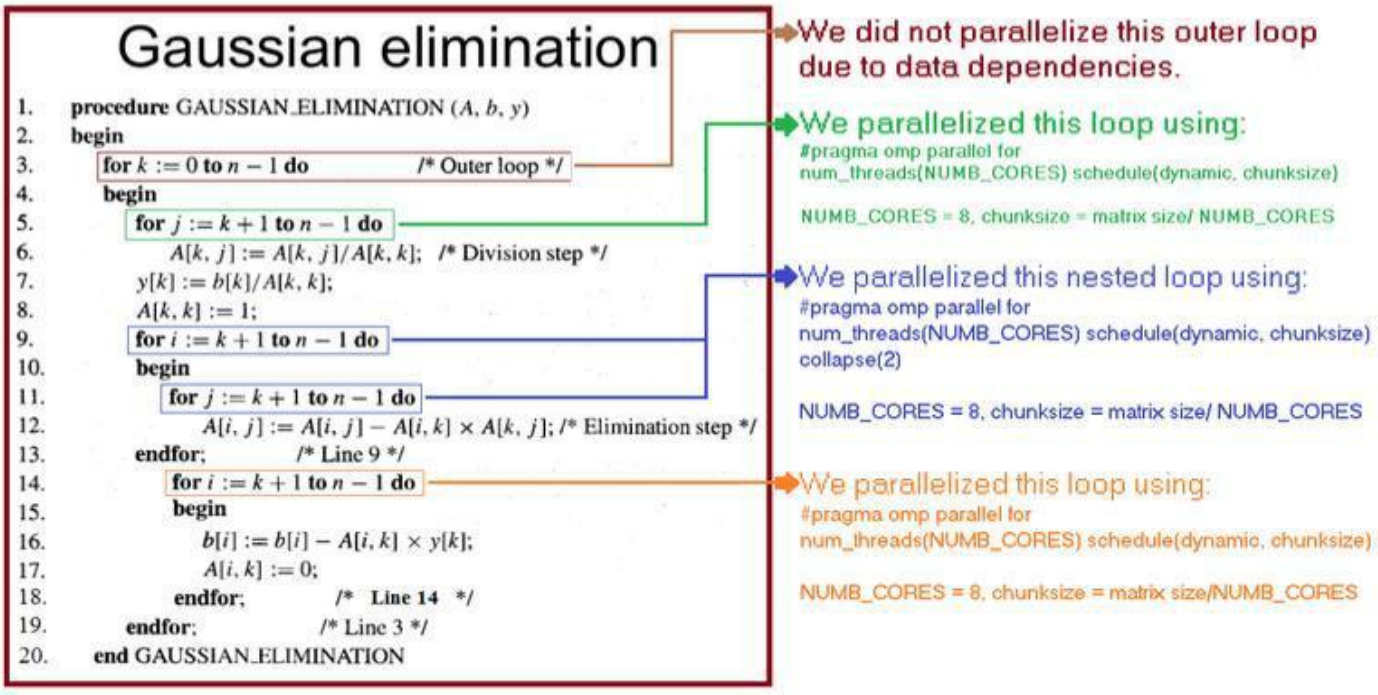
Gaussian elimination also called as the row reduction algorithm, solves a system of linear equations. It plays the basis for the classic algorithms for computing canonical forms of the integer. The start of the Gaussian elimination is the conversion of the given linear equations into matrices wherein the coefficients of the equations represent the elements of the matrices.

- Row and column transformations are applied and the matrices are now converted into upper triangular matrices with the unit elements as the diagonal elements.
- Now, Gaussian elimination is used to generate the resultant matrix and helps solving the given linear equations when the back substitution is done. [3]

**Parallel Implementation of Gaussian Elimination:**

We proposed to implement Gaussian elimination using OPENMP as follows

- Assuming that the size of the matrix is greater than that of the number of cores used.
- The size of the matrix can be given to the program as a command line parameter. The number of cores that the program is intended to run on, is defined by the *NUMB_CORES* macro.
- This macro makes it easy to record execution times on 1 core and 8 cores, as we do not change values at every



instance in our code. We parallelize the initialization of the matrix.

- Before every "*for"* loop in the initialization of the matrix, we use the compiler directive along with the clauses.
- We always create a number of threads as there are cores on the machine. The *chunksize* variable is defined as the size of the matrix divided by the number of cores. For nested *"for"* loops, we use the *collapse* clause at the end of the directive.
- We make a slight modification to the implementation of the Gaussian Elimination algorithm. As can be seen in **Figure**, we add an additional "*for"* loop before the end of the outer loop.
- This modification does not change the algorithm. It only changes the implementation of the algorithm. We then parallelize the algorithm as shown in Figure 1.

**Motivation:**

- The reason for choosing this implementation is after we have experimented with different scheduling policies and varying number of threads, we observed the following.
- The static scheduling makes the program slower and adding to that, optimal *chunksize* is necessary for making the execution times faster.
- The scheduling policy is dynamic. Further, the number of threads is equal to the number of cores that the program intends to run upon.

## **IV.**REFERENCES


[1] "OpenMP," *Wikipedia*. 12-Dec-2016.

[2] Blaise Barney, "OpenMP." [Online]. Available: https://bth.itslearning.com/ContentArea/ContentArea.aspx?LocationType=1&LocationID=6423. [Accessed: 23-Dec-2016].

[3] "Gaussian elimination," *Wikipedia*. 19-Dec-2016.