Project Assignment 3: OpenMP Programming
Project Report

Denim Deshmukh - dede19@student.bth.se
and
Koyyada Sai Pranav - saky19@student.bth.se

27'th May 2020

1 Introduction

The program use OpenMP (Open Multi-Processing) API for Implementing a parallel version of Quick sort and Gaussian elimination.
Quick sort [5] is a divide and conquer based efficient sorting algorithm with time complexity of O(n log n). The array is partitioned into smaller sub-arrays using a pivot is chosen from the array. All the elements greater then the pivot are swapped from their position to the right of pivot. All the elements smaller then the pivot are swapped from their position to the left of pivot. Thus, this arrangement ensures all the left elements are smaller and all right elements are larger than the pivot element. The left and right elements are further partitioned in sub-arrays. The quick sort algorithm is applied recursively on the sub-arrays till the number of elements in segmented sub-array is one which indicates the partition of the pivot is sorted. Fig 1 shows a simple example of quick sort.

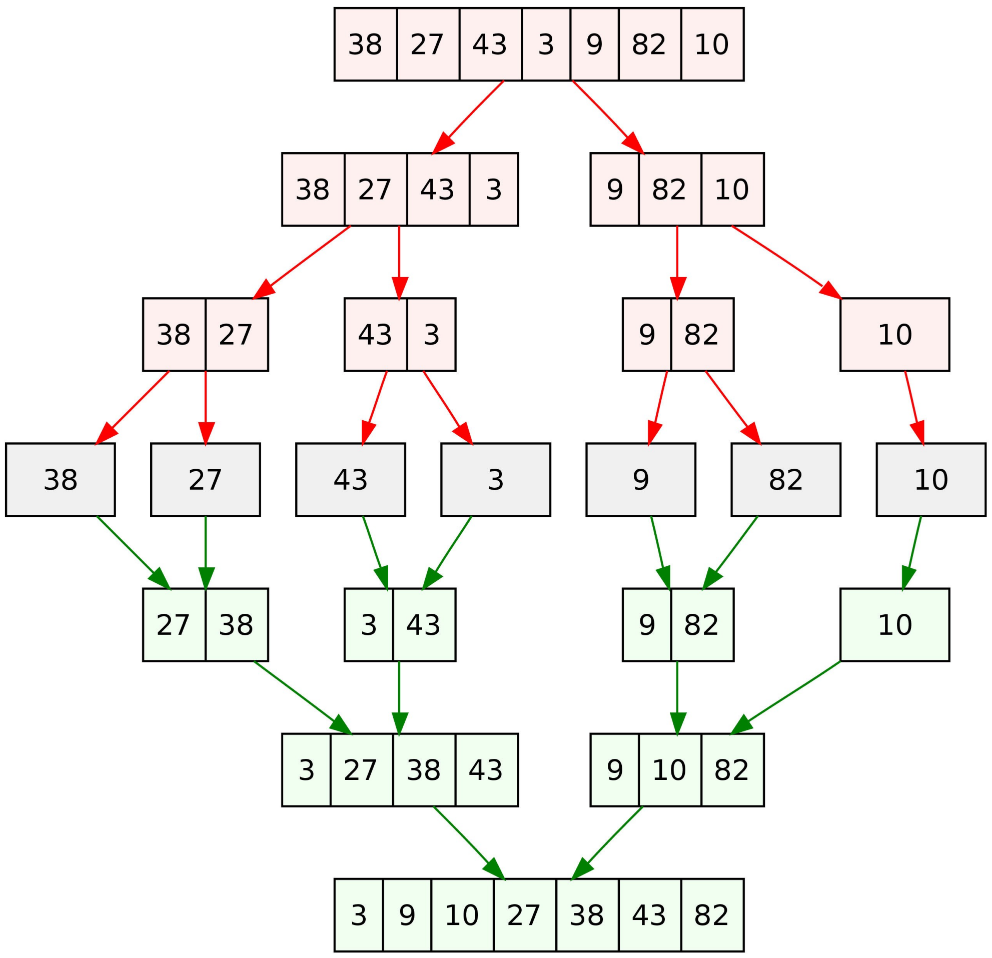

Fig1: QuickSort sorting technique illustration on array of size 7

Gauss Elimination is a method for solving Equation like
Ax=B(1)
Gauss Elimination takes a system of equation and transform the system of equation into a upper triangular matrix and the back substitution is used to solve for unknown variable and get solution.

$$a_{11}x_1 + a_{12}x_2 + ... + a_{1n}x_n = b_1$$

$$a_{21}x_1 + a_{22}x_2 + ... + a_{2n}x_n = b_2$$

$$.$$

$$.$$

$$a_{n1}x_1 + a_{n2}x_2 + ... + a_{nn}x_n = b_n$$

From above Equation we we create a matrix as

$$\text{A=}\begin{vmatrix} a_{11} & a_{12} & ... & a_{1n} \\ a_{21} & a_{22} & ... & a_{2n} \\ . & & & \\ . & & & \\ a_{n1} & a_{n2} & ... & a_{nn} \end{vmatrix} \text{X=}\begin{vmatrix} x_1 \\ x_2 \\ . \\ . \\ x_n \end{vmatrix} \text{B=}\begin{vmatrix} b_1 \\ b_2 \\ . \\ . \\ b_n \end{vmatrix}$$

By performing elementry operation as given in Figure 1 we transfor the matrix to upper triangular matrix

$$\text{A}^{'} = \begin{vmatrix} a_{11}^{'} & a_{12}^{'} & ... & a_{1n}^{'} \\ 0 & a_{22}^{'} & ... & a_{2n}^{'} \\ . & & & \\ . & & & \\ 0 & 0 & ... & a_{nn}^{'} \end{vmatrix} B = \begin{vmatrix} b_1^{'} \\ b_2^{'} \\ . \\ . \\ b_n^{'} \end{vmatrix}$$

We get solution for system of linear equations by back substitution by using given equation(1) as

$$x_i = \frac{1}{a_{ii}^{'}}\Big(b_i^{'} - \sum_{=i+1}^{n} a_{ij}^{'}x_j\Big) \tag{1}$$

2 Quick Sort

2.1 Implementation

```
QuickSort(A,low,high)
{
  if(low<high)
  {
    pivot_index = partition(A,high,end);
    QuickSort(A,low,pivot_index-1); //parallelize
    QuickSort(A,pivot_index+1,end); //parallelize
  }
}
```

Fig2: Parallel implementation of Quick Sort Algorithm

Fig 2 show parallel implementation of Quicksort algorithm. The initial call of QuickSort will be executed by one thread in the team using #pragma omp single nowait. The recursive calls of QuickSort will be executed in parallel. Thus available threads will execute the partitioned sub-arrays parallely until only one element remain in the recursively partitioned sub-array.

2.1.1 Measurement

Fig 3 and 4 shows the execution time and speed up comparison respectively. Parallel implementation of quick sort on 8 threads has a significant performance improvement over the serial with 7 times speed up.

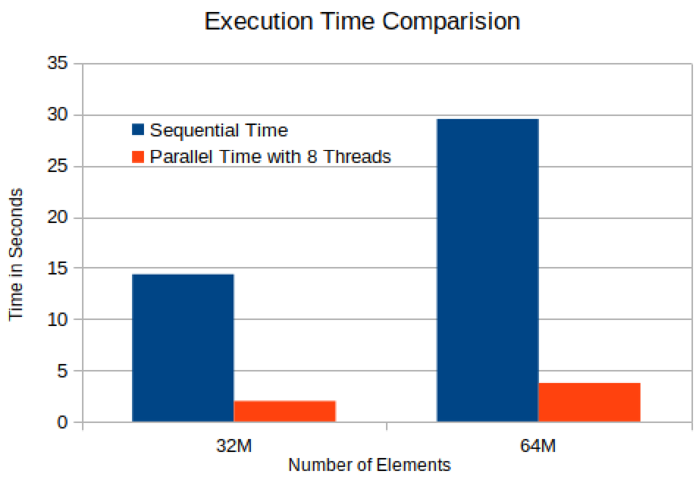

Fig3: QuickSort execution time comparison

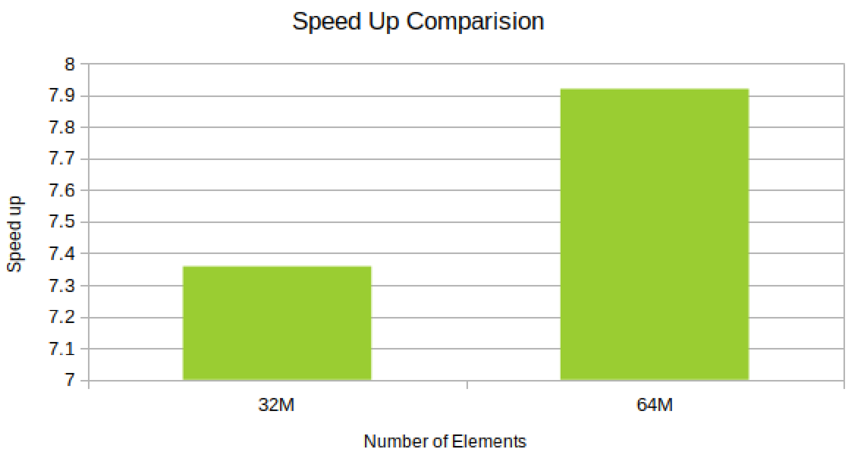

Fig3: QuickSort speed up comparison

3 Gauss Elimination

3.1 Implementation

```
Gaussian Elimination (no pivoting):
for (int i = 0; i < N-1; i++) {  /* Outer loop */
  // parallel part
  for (int j = i; j < N; j++) {
  //j loop is parallelizable.
   //  later iterations  do not depend on earlier ones, j iterations be
      computed in any order
    double ratio = A[j][i]/A[i][i]; /* Division step */
    for (int k = i; k < N; k++) {  //k loop is also parallelizable.
       // its number of iterations varies but is calculable for every i.
       // None of the later iterations depend on earlier ones, and they can
          all be computed in any order
      A[j][k] -= (ratio*A[i][k]);
      b[j] -= (ratio*b[i]);
    }
  }
}
```

Fig:1

The Gauss Elimination method is to solve system of equations in two major part first is to get the upper triangular form of matrix by doing row wise elimination ,following the algorithm in Figure 1 on give matrix and second is back substitution step.
The implementation is based on what part of code can be parallelized and what part cannot be parallelized.From fig.1 The inner loop is parallelizable since later iterations in the loop are not dependent on earlier ones and order of calculation does not matter in this loop.
The most inner loop k is also parallelizable , it's number of iterations varies but is calculated for every i and also later iterations in the loop are not dependent on earlier ones and order of calculation does not matter in this loop.[2][3]. The back substitution(equation-1) part is solved using collapse which allow for nested for loop threading.
For data distribution dynamic scheduling is used in program which is supported by OpenMP.

3.2 Measurements

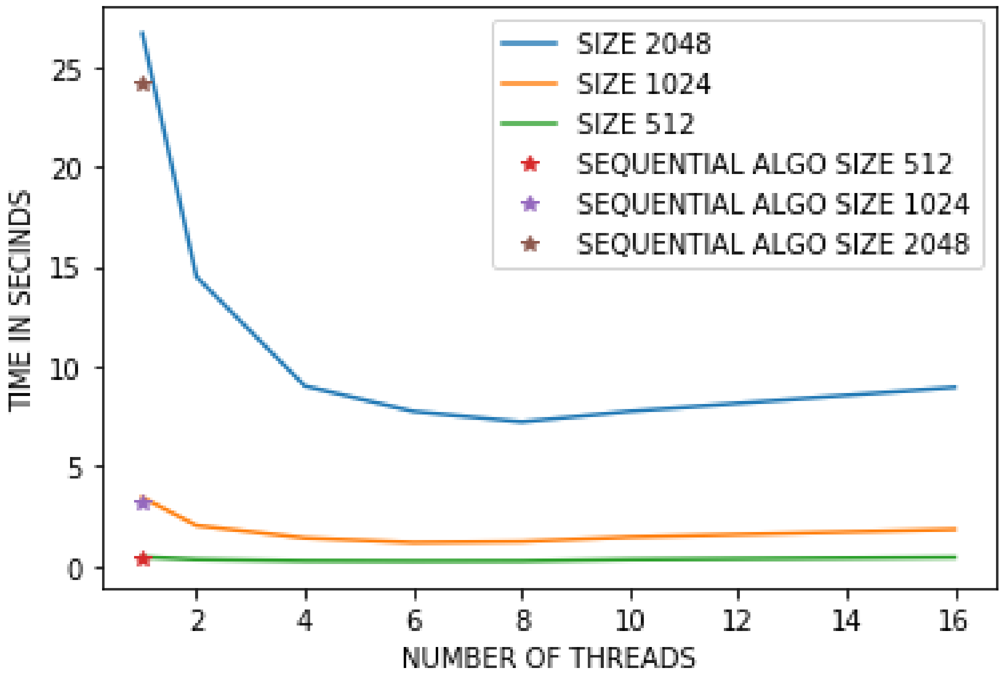

Figure 2:Time to execute V/S Number of Threads

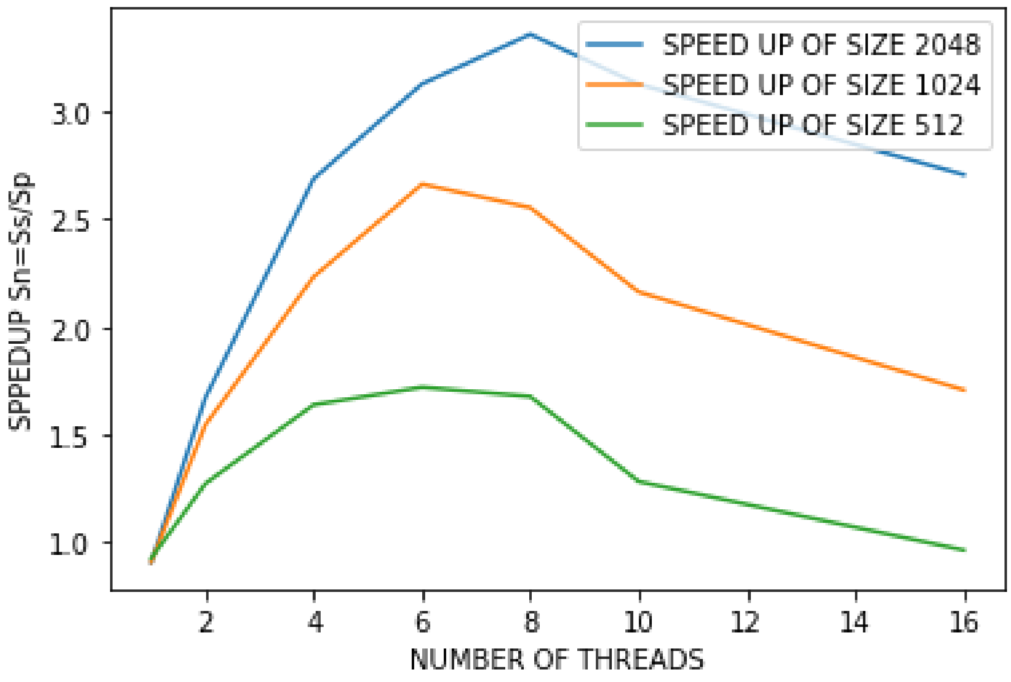

Figure 3:Speed Up V/S Number of Threads

As in Figure 2 we the program was executed on various size of 2048, 1024 and 512 size matrix and on various number of threads as 1, 2, 4, 6, 8, 10 and 16 and the extracted data is shown in figures and it was notices that there is a optimal point for every size which give best time of execution , that point was around 8 threads.
For speed up was calculate by equation 2[6] given below which is ratio of time to execute serially to time to execute parallelly. The speed up of positive sub-linear and have crossed 1.5 speedup mark for optimal number of threads according to size. The speed up can be seen to increase till the 8 number of threads and then a flat or decrease curve is noticed.

$$S = \frac{S_s}{Sp}$$

No of Threads	Time(Sec),SIZE=2048	Time(Sec),SIZE=1024	Time(Sec),SIZE=512
1	26.689	3.442	0.448
2	14.491	2.026	0.326
4	9.016	1.403	0.253
6	7.744	1.176	0.241
8	7.215	1.226	0.248
10	7.747	1.448	0.324
16	8.950	1.833	0.431

Table 1: Time for execution for various size and for number of threads

Size	Time in Seconds
512	0.416125
1024	3.132290
2048	24.223247

Table 2: Size V/s linear Execution time

No of Threads	SpeedUp,SIZE=2048	SpeedUp,SIZE=1024	SpeedUp ,SIZE=512
1	0.907	0.910	0.927
2	1.671	1.546	1.273
4	2.686	2.232	1.639
6	3.127	2.661	1.720
8	3.356	2.553	1.677
10	3.126	2.162	1.283
16	2.706	1.708	0.965

Table 3: Speed Up for various size and for number of threads

4 References

[1] https://bisqwit.iki.fi/story/howto/openmp

[2]https://riebecca.blogspot.com/2008/12/supercomputing-course-openmp-syntax-and$_1$5.*html*

[3]$https://www.cs.rutgers.edu/\ venugopa/parallel_summer2012/ge.html$

[4]$A.Grama, A.Gupta, G.Karypis, andV.Kumar, Introductiontoparallelcomputing, 2ndedition, Addison-Wesley, 2003.$

[5]$Al-Dabbagh, SinanSameerMahmoodHazim, Nawaf.(2016).ParallelQuicksortAlgorithmusingOpenMP.5.372-$
$382.10.6084/M9.FIGSHARE.3470033.$