DAA Assignment No. : 9
SOLVING THE NxN DETERMINANT PROBLEM USING PARALLEL PROGRAMMING

Lovely Digra
IIT2017090

Abhishek Vishwakarma
IIT2017091

Anuj Raj
IIT2017092

***Abstract*—Determinants has been used intensively in a variety of applications through history. It also influenced many fields of mathematics like linear algebra. Finding the determinants of a squared matrix can be done using a variety of methods, including well-known methods of Leibniz formula and Laplace expansion which calculate the determinant of any NxN matrix in O(n!) The goal of this paper is to optimize the algorithm for finding determinant of a NxN matrix. To achieve this goal, the concepts of a multithreading of a process is used. While designing the problem time complexity for the algorithm is calculated which comes to be $O(n-1!)$ but is better than brute force method by a factor of n i.e n times faster where n is the order of matrix.This is also observed from the comparison plots.**

***Keywords*—** sequential programming, recursion, discriminants, determinant, matrix, cofactor, minor, Laplace expansion, LU decomposition, choelsky decomposition, divide and conquer, C++ thread library ,parallel programming , multi threading

I. INTRODUCTION

Finding the determinant value of an NxN matrix has been an important topic that is useful in many fields of mathematics. In fact, determinants have led to the study of linear algebra. Although the attitude towards determinants has been changed, but however, the theoretical importance of them can still be found. Through history, many applications have used determinants to solve some mathematical problem, for example:
Solving linear equation systems by applying Cramers Rule.
Defining resultants and discriminants as particular determinants.
Finding the area of a triangle, testing for collinear points, finding the equation of a line, and other geometric applications.
Various applications in abstract mathematics, quantum chemistry, etc.
There are several methods for finding the determinant of a matrix, including the native methods of Leibniz formula and Laplace expansion (cofactor expansion) which are both of order (n!). There are also some decomposition methods, including, LU decomposition, Cholesky decomposition and QR decomposition (applied for symmetric positive definite matrices), which improve the complexity to $O(n^3)$. Designing parallel algorithms involves using some design patterns. Those well-known patterns are implemented using parallel processing techniques, which is available in many different programming languages. The patterns are:

A. Bag of Tasks

A number of workers maintain a dynamic bag of homogeneous tasks. All tasks are independent from each other so every worker can get a task, complete it, and return to get another one, and so on until the bag is empty.

B. Divide and Conquer

Breaking a big task into sub-tasks, which in turn, break the sub-tasks into smaller ones until we reach the base case that can be used to solve the parent tasks. Recursion is usually used in this pattern [5]. An example for this pattern is Merge Sort. We have described the design and implementation of a C++ thread library that solves problems recursively using parallel programming by forking them into sub-tasks and then waiting them to be joined.

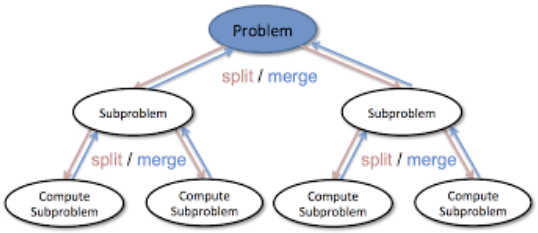

Fig. 1: divide and conquer strategy

In this paper, we are applying the Laplace expansion method of finding the determinant value of any NxN matrix using both sequential and parallel algorithms. Laplace expansion, which applies the divide-and-conquer pattern.

II. ALGORITHM DESIGN AND EXPLANATION

A. Sequential Approach

Laplace expansion is a method for finding the determinant by recursively expanding each matrix to sub-matrices until reaching a base case of 2x2 matrix. It expresses a determinant

$|A|$ of an NxN matrix A as the summation of all determinants of its N-1xN-1 sub-matrices.
Given an NxN matrix A, where a_{ij} is the value of the element located in the ith row and jth column. The cofactor is then given by:
$C_{ij} = (-1)^{i+j}.M_{ij}$
where M_{ij} is the minor of A in a_{ij}. Hence, the determinant $|A|$ is then given by:
$|A| = \sum_{j=1}^{n} a_{ij}C_{ij} = \sum_{j=1}^{n} a_{ij}C_{ij}$
Depending on Laplace expansion method, we follow the following scheme to calculate $|A|$:
Find the cofactor by multiplying the minor by either 1 if the sum of the column and row numbers is even, or -1 if the sum of the column and row numbers is odd .
Recursively, and for each column of a certain row, find the minor determinant of A by eliminating the elements that are in the same row and column of a_{ij} to consist a sub-matrix.
If you reach the base case, calculate the determinant directly. Multiply the result by the value of aij.
Add up all sub-determinants to calculate the total determinant of A.
The expansion is working in the following manner:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

$$|A| = a_{11}.\begin{vmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{vmatrix} - a_{12}.\begin{vmatrix} a_{21} & a_{23} \\ a_{31} & a_{33} \end{vmatrix} + a_{13}.\begin{vmatrix} a_{21} & a_{22} \\ a_{31} & a_{32} \end{vmatrix}$$

This method has a complexity of $O(n!)$ so it is usually not feasible to be applied in real applications. However, we will design sequential and parallel algorithms to find determinants using this method.

B. Parallel Approach

We observe that in the divide and conquer sequential approach the determinant of submatrices can be calculated independently. Thus it is faster to keep separate thread for separate determinant calculation of submatrices. Here we divide the whole process of finding determinant of a NxN matrix into N threads which runs simultaneously. In a parallel environmnent with N processors each thread runs parallely. This speeds up the whole process by a factor of N.

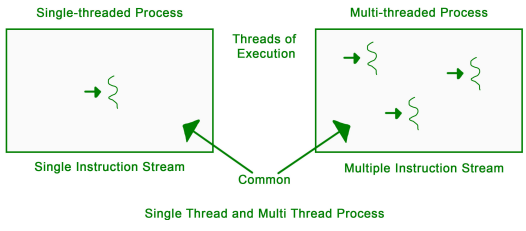

Fig. 2: parallel approach using multi-threading

Algorithm 1 Sequential Approach

procedure DETERMINANT($int\ \ mat2[][INT\_MAX]$, $int\ size$)
  $det \leftarrow 0$
  if $size == 2$ **then**
    $return\ \ mat2[0][0] * mat2[1][1] - mat2[0][1] * mat2[1][0]$
  else
    $mat3[MAX][MAX] \leftarrow 0$
    for x **from** $0 \rightarrow size$ **do**
      $subi \leftarrow 0$
      for i **from** $1 \rightarrow size$ **do**
        $subj \leftarrow 0$
        for j **from** $0 \rightarrow size$ **do**
          if $j\ != x$ **then**
            $mat3[subi][subj] \leftarrow mat2[i][j]$
            $subj \leftarrow subj1$
        $subi \leftarrow subi1$
      $det \leftarrow det$+$pow(-1, x) * mat2[0][x] * determinant(mat3, s-1)$
    $return\ det$
procedure CREATETHREAD($int\ x, int\ size$)
  $mat2[INT\_MAX][INT\_MAX] \leftarrow 0$
  $subi \leftarrow 0$
  for i **from** $0 \rightarrow size$ **do**
    $subj \leftarrow 0$
    for j **from** $0 \rightarrow size$ **do**
      if $j\ != x$ **then**
        $mat2[subi][subj] \leftarrow mat[i][j]$
        $subj \leftarrow subj1$
    $subi \leftarrow subi1$
  $cofactor[x] \leftarrow pow(-1, x) * mat[0][x] * determinant(mat2, s-1)$

procedure MAIN
  $Taking\ the\ inputs$
  $finalDet \leftarrow 0$
  $cofactor[INT_MAX] \leftarrow storecofactorforeachrow$
  $size \leftarrow size\_of\_matrix$
  $mat[i][j] \leftarrow store\_value\_of\_matrix$
  for I **from** $0 \rightarrow size$ **do**
    $th.push\_back(thread(createThread, x, size))$
  for I **from** $0 \rightarrow size$ **do**
    $th[i].join()$
  for I **from** $0 \rightarrow size$ **do**
    $finalDet \leftarrow finalDetcofactor[i]$
  // Print finalDet

III. EXPERIMENTAL STUDY

For the experimental analysis of this algorithm, order of matrix N is varied, to see how the execution time varies with increasing order. The value of each a_{ij} is taken from the standard random generator in the range $1 < a_{ij} < 100$. The graph of order vs execution time is plotted using .dat file generated from the source code. see Fig 1.

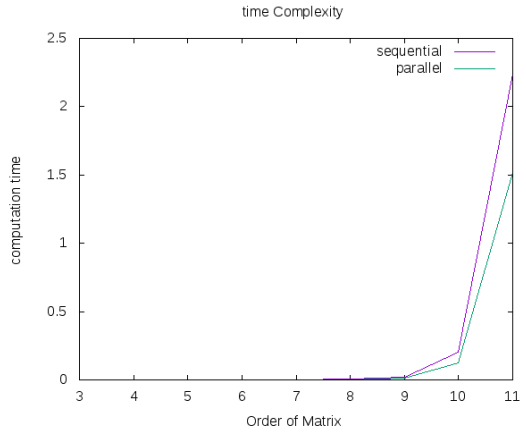

Fig. 3: comparision plot of sequential and parallel approach

IV. TIME COMPLEXITY AND DISCUSSION

In the algorithm design section, it is discussed that the sequential method has time complexity of order $O(N!)$ where N is the order of the square matrix. The parallel approach is in essence should be N times faster than the sequential one if there is large amount of processors available. When N is smaller than total number of processors available.

Space Complexity: Stack overhead of the recursion is $O(N!)$ for the proposed algorithm thus the complexity has the order $O(N!)$.

V. CONCLUSION

In this paper, we have applied two methods to find the determinants of NxN matrices using both sequential and parallel algorithms. The new parallel algorithm of Laplace expansion provided much better results than the sequential algorithm. We also plotted the graph of order vs computation time and found that the the proposed algorithm is faster than the sequential algorithm . In future, we plan to parallelize other LU decompostion methods to find determinant. We also aim to formalize our experiments.

REFERENCES

[1] https://www.tutorialspoint.com/cplusplus-program-to-compute-determinant-of-a-matrix
[2] https://thispointer.com//c-11-multithreading-part-1-three-different-ways-to-create-threads/
[3] https://matrix.reshish.com/