Solving system of linear equations using Crout LU decomposition

CS646 Term Paper

Virajith Jalaparti (Y5510)
Dilip Kumar Kola (Y5827169)

1. Problem Statement

In this project, we aim to solve a given system of linear equations of the form \( Ax = b \), where \( A \) is a matrix of size \( n \times n \), \( x \) and \( b \) are vectors of size \( n \). Each element of \( A \) and \( b \) is a number from a Galois Field of size \( p \) (a prime number), denoted by \( GF(p) \). We intend to find vector \( x \) such that it satisfies the matrix equation \( Ax = b \) in the field \( GF(p) \). In this implementation, we assume that \( p \) is a 128-bit prime and that \( n \) is of the order of \( 10^4 \).

2. Algorithm Used

To solve the problem stated above, we decompose \( A \) into a product of an upper triangular matrix \( U \) and a lower triangular matrix \( L \) (and possibly a permutation matrix \( P \)). This LU decomposition is done using the Crout Algorithm, which decomposes a matrix into a unit upper triangular matrix and a lower triangular matrix. Fig.1 shows the pseudo code for the Crout algorithm. Once \( L \) and \( U \) are obtained, the solution of the system is obtained using the forward substitution and the backward substitution.

1. for \( k = 1 \) to \( N \)
2.     for \( i = k \) to \( N \)
3.         for \( h = 0 \) to \( k-1 \)
5.         for \( j = k+1 \) to \( N \)
6.             for \( h = 0 \) to \( k-1 \)

Fig. 1: Shows the pseudo-code of the Crout LU decomposition. \( A \) is the matrix to be decomposed and the result is stored in \( A \) itself.

3. Computation Model Used

In this project, we have developed a parallel algorithm to work on a cluster machine. We used a cluster of 7 machines to test the implementation of the algorithm discussed above. Each of the machines used an Intel Pentium 4 (3.4GHz) processor with 1024KB cache.
4. Implementation

We have implemented the Crout LU decomposition and the subsequent solving of the equations for a cluster. Fig. 2 shows the various data dependencies that the various loops in the algorithm, given in Fig. 1, impose. From Fig. 2, we can see that the elements in the $i^{th}$ row and $j^{th}$ row ($i < j$), can be updated parallel, as far as their dependencies upon the rows1 to $i$ are considered. It is this parallelism that we exploit while implementing the parallel version of the Crout algorithm.

The data is divided equally among all the processors. Processor $p_i$ stores the rows $i, i + \text{numprocs}, i + 2 \times \text{numprocs}$ . . . of the matrix $A$ and $B$. This data distribution is shown in Fig. 3. Each processor reads the rows that are assigned to it, directly from the file. Another implementation could have been that a single processor reads the entire file and broadcasts the data to the various processors. The Crout LU decomposition takes $n$ steps. In step $i$, the processor, say $p$, with id equal to $\%\text{number of proc}$, divides the elements of row $i$, above the diagonal, by the diagonal element $A[i][i]$, provided it is non-zero and broadcasts (using MPI_Bcast) the modified part of the $i^{th}$ row ($n - i$ elements $A[i][i + 1]$ to $A[i][n]$) to all the other processors, after the receipt of which, each processor (including the one that has sent the row) updates the elements that it has stored and are present in a row $j$, $j > i$.

Handling Pivoting:

On the other hand if $A[i][i] == 0$, then a processor starts pivoting. At first, the processor $p$, examines the elements of the $i^{th}$ column, that are present with it and belong to a row greater than $i$, to find a non-zero element. If it finds such an element, say in row $k$, it swaps the rows $i$ and $k$, and the new $i^{th}$ row takes the place of the old one in the process discussed above (being broadcasted and being used for updating the rows of the other processors). This is shown in Fig. 4. However, if the processor $p$ does not find a non-zero element in the part the $i^{th}$ column, that are stored with, it needs to get a pivot row from another processor. It sends this information to the other processors, by sending a special character (“$” in our implementation) in the broadcast array that is supposed to contain the row that is to be sent by $p$ to all the other processors. This is shown in Fig. 5. If such a special character is receive by the other processors, they look into their $i^{th}$ column for a non-zero element and send the row number of that element to processor $p$ (using MPI_Send), as shown in Fig 6. If no such element exists, the respective processor sends a number greater than $n$ (the dimension). It is mandatory for each processor to send a row number to $p$. Now $p$ receives all the probable pivot row numbers from the various processors (using MPI_Recv) and chooses the one (smaller than $n$) which belongs to processor with the least index. If all the received numbers are greater than $n$, then it can concluded that the matrix has no LU decomposition.
Now processor $p$ informs all the others the id of the processor, say $q$, which has been chosen to provide the new pivot row, say $l$. Now processors $p$ and $q$ swap their $i^{th}$ and $l^{th}$ rows and subsequently the new pivot row is broadcasted to all the other processors. Now each processor uses this broadcasted row similar to the way described above to update the elements in the part of the array that they have.

This entire process is repeated $n$ (=dimension of array) times, after which we get the LU decomposition of $A$. Further, we maintain a permutation array $P$, which is used to save the swapping of the elements of the $B$ array. The forward substitution phase follows the LU decomposition, which again has $n$ stages. In the $i^{th}$ stage, the processor with id equal to $i \% \text{numberOfProcs}$, divides $B[i]$, with $A[i][i]$, to get the value of $X[i]$, and sends this value to all the other processors. Each processor now subtracts $A[j][i] \times X[i]$ from $X[j]$ for $j > i$ and $j \% \text{numprocs}$ is equal to the processor id. This step is done in parallel by all the processors. The backward substitution phase is similar to the forward substitution phase except for the fact that no division is required (as the diagonal elements of $U$ are 1) and instead of proceeding from the top of the array to the bottom, we proceed from the bottom of the array to the top.
Another implementation issue to be considered is the way the matrix is stored and transferred between the processors. We store the matrix using \texttt{mpz_t} (GMP integers) and before the elements of a row are transferred, they are converted to strings (char arrays) using \texttt{mpz_get_str}. Finally after the various processors receive the row of elements, they convert them back to \texttt{mpz_t} data to perform the required mathematical operations. One optimization we had considered here was to create strings in base 62 (this is the largest base that GMP provides) instead of the normal base 10, as the former uses the space more efficiently (1 byte (char) stores 6 bits of useful information in the former case, while in the latter it stores only about 3 bits of useful information).

5. **Results**

Table 1 shows the results obtained when this implementation was run on a cluster of 7 machines with each of the machines having an Intel Pentium 4 (3.4GHz) processor with 1024KB cache.

Fig. 7 shows the plot of the time taken by the various processors vs the number of processors that are used. This shows that the time of execution increases exponentially with increase in size of the matrix. Thus we give Fig. 8, which is a similar graph but shows the time in a logarithm scale, as it helps us better understand the variation of time with number of processors. It can be seen that for all sizes of the arrays a similar pattern is variation of time with number of processors is followed.

Fig. 9 shows the speed up achieved for various number of processors. We can see that as the size of the array is increasing the speed-up obtained is decreasing. This is because of the increased communication cost, with increase in size of array. For 100*100 the speed up is low, this is because of the small size of the array, which does not achieve much gain with increase in number of processors, as the communication cost overshadows the speedup due to parallelism.

6. **Limitations**

The current implementation was done keeping in mind that the array size is of the order of $10^4$ and the elements are from $GF(p)$ where $p$ is a 128-bit prime. The size occupied by such an array on a single processor is around 2.2GB. Thus as the number of processors of the cluster are increased, the current algorithm also scales for larger size matrices as the data is distributed evenly on all the processors. If $n$ is the dimension of the array and $p$ is the number of the processors used, then the space per processor required for storing this matrix is $(\frac{n^2 n}{p} + 2 \cdot n) \cdot \text{(size of one element of the matrix)} + \delta$, where $\delta$ is a much smaller quantity (at least 3 orders of magnitude) compared to the other term. Thus our implementation scales very well, with increased number of processors. However, if the number of processors is fixed, it fixes the maximum dimension of the array that can be stored (depending on the kind of addressing of the machine, of course) and hence it is limitation.

7. **Libraries used**

In this project, we have used MPICH and GMP, the former being used for message passing between the various processors while the latter was used for the handling of the big number arithmetic.
Table 1: This table shows the time of execution of the various implementations in seconds for varying size of matrix and varying number of processors.

<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>100*100</th>
<th>500*500</th>
<th>1000*1000</th>
<th>2000*2000</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.28786</td>
<td>35.813059</td>
<td>289.449771</td>
<td>2237.747407</td>
</tr>
<tr>
<td>2</td>
<td>0.18465</td>
<td>18.838455</td>
<td>202.673171</td>
<td>2181.143618</td>
</tr>
<tr>
<td>3</td>
<td>0.18028</td>
<td>13.348548</td>
<td>185.741592</td>
<td>1715.668600</td>
</tr>
<tr>
<td>4</td>
<td>0.17458</td>
<td>10.361999</td>
<td>145.447725</td>
<td>1188.620965</td>
</tr>
<tr>
<td>5</td>
<td>0.166</td>
<td>8.848662</td>
<td>118.735002</td>
<td>928.907118</td>
</tr>
<tr>
<td>6</td>
<td>0.14888</td>
<td>7.582422</td>
<td>99.475132</td>
<td>799.447776</td>
</tr>
<tr>
<td>7</td>
<td>0.10839</td>
<td>6.887935</td>
<td>84.914339</td>
<td>662.691515</td>
</tr>
</tbody>
</table>

Figure 7: This plot shows the variation time of execution with number of processors for various array sizes, y axis is on a linear scale.

Figure 8: This graph shows the variation of the time of execution with the number of processors used, with the y axis being in a logarithmic scale.

Figure 9: This plot shows the variation of Speed up with number of processors for various matrix sizes.