Project Report: Parallelizing Matrix Inversion

15-418: Parallel Computer Architecture and Programming

By Allen Xie (axie) and Chi Nguyen (clnguyen)

Summary

For this research project, we implemented matrix inversion through two popular methods of decomposition/elimination. We parallelized our implementations on the language platform of C++ (with OpenMP) and measured performance on the 8-core machines in GHC Labs. The parallelization efforts of this project focused on two key areas. The first area of focus is parallelizing LU-Decomposition and Gauss-Jordan Elimination. These are the two methods we used to generate the inverse of an invertible matrix. We then focused on parallelizing the workload through block recursion. We decomposed the matrix into smaller submatrices that can be used to find the original inversion.

Background

Matrix inversion is a crucial operation utilized in various industries and disciplines. Whether it is simply used to help solve linear equations or integrated as a part of some complicated 3-D graphic rendering, matrix inversion is often an expensive operation that can benefit greatly from performance optimization. Given an invertible matrix of size *n* by *n,* the cost of a generic matrix inversion algorithm is typically in $O(n^3)$.

One aspect of the matrix inversion problem that we find extremely interesting is the numerous ways to calculate the inverse. Methods can range from calculating the determinant of submatrices, performing various matrix row operations, to decomposing the matrix into special matrices. For this project, we picked out Gauss-Jordan Elimination and LU-Decomposition as the two core methods to explore and parallelize.

Gauss-Jordan Elimination

$$\left[A \quad | \quad I\right] \longrightarrow \left[I \quad | \quad A^{-1}\right]$$

The standard Gauss-Jordan method of finding matrix inverse involves performing a series of matrix row operations to convert the original input matrix (if invertible) into the identity matrix. These row operations can be performed, in the same order, on an identity matrix to obtain the inverse of the original matrix. Based on the previous description, the two key data structures are a copy of the original matrix and an identity matrix of the same dimensions. Normally there are three basic matrix row operations we can perform, which include multiplying the row by a constant, switching two rows (element-wise exchange), and adding the multiple of a row to another row (allowing negative multiples for subtraction). During implementation, we observed that the only times we need to exchange is when the matrix diagonal element in the current row becomes 0 (so indices (1,1), (2,2), … (n,n)). In this case, we can just take a lower row for which the element on the same column is not 0 and add it to the current row.

LU-Decomposition

LU-Decomposition is a method that allows us to factorize an invertible matrix A into matrices L and U such that L is a lower triangular matrix and U an upper triangular matrix where $A = LU$. To obtain A^{-1}, we only need to compute $U^{-1}L^{-1}$ because $I = AA^{-1} = LUU^{-1}L^{-1}$. One advantage of using triangular matrices is when the diagonal elements are all 1, finding the inverse requires only changing the signs of the off-diagonal elements.

Block-Recursion

$$\begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix}^{-1} = \begin{bmatrix} \mathbf{A}^{-1} + \mathbf{A}^{-1}\mathbf{B}\left(\mathbf{D} - \mathbf{C}\mathbf{A}^{-1}\mathbf{B}\right)^{-1}\mathbf{C}\mathbf{A}^{-1} & -\mathbf{A}^{-1}\mathbf{B}\left(\mathbf{D} - \mathbf{C}\mathbf{A}^{-1}\mathbf{B}\right)^{-1} \\ -\left(\mathbf{D} - \mathbf{C}\mathbf{A}^{-1}\mathbf{B}\right)^{-1}\mathbf{C}\mathbf{A}^{-1} & \left(\mathbf{D} - \mathbf{C}\mathbf{A}^{-1}\mathbf{B}\right)^{-1} \end{bmatrix}$$

A major challenge in breaking down many matrix operations' problem size is that we often can not directly reconstruct the solution to the original problem from the solutions of the subproblems. The above figure highlights the reconstruction necessary to obtain the inverse of the original matrix. There are many calculations that can be reused during the reconstruction process like A^{-1}, CA^{-1}, $CA^{-1}B$, and $(D - CA^{-1}B)$. Reusing the outputs from a lot of these calculations can greatly reduce the overall workload. However, one drawback/restriction in using block-recursion is that our sub-matrix A must be invertible. This adds extra complications to the recursive procedure and can limit use cases on certain matrices.

Approach

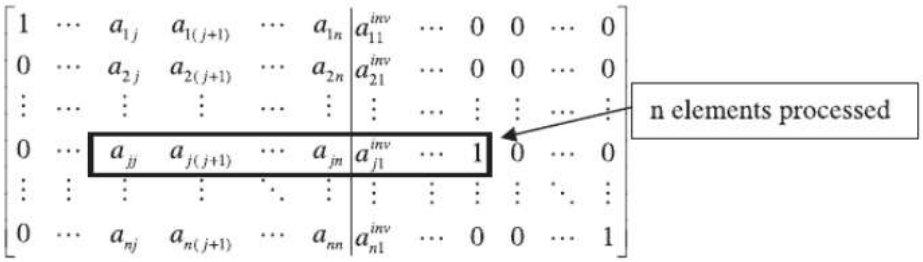
Inversion through Gauss-Jordan Elimination

High-level Pseudocode of actual implementation:

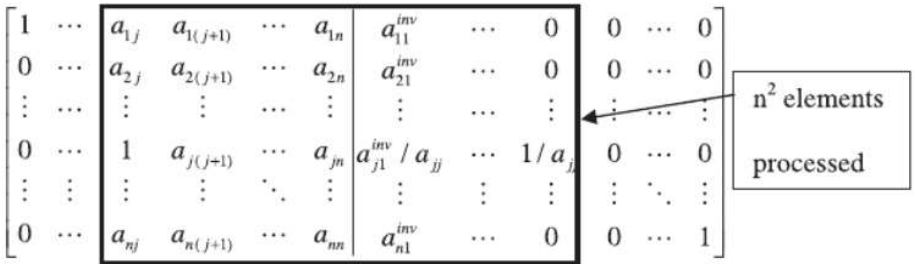
1. Begin by generating the identity matrix I to copy row operations done on original matrix A
2. We wish to perform row operations such that A becomes an identity matrix. So we perform the following for each row i
   (a) if $A[i][i] == 0$, we will find a row j where $i < j$ and $A[j][i] \neq 0$. We know such row exist from the matrix A being invertible.
   (b) We divide all elements on row i by $A[i][i]$
   (c) for every other row, j, in A, we either add or subtract a multiple of row i such that $A[j][i]$ equals to 0
3. Now our matrix A should be an identity matrix and our matrix I should contain inverse of original A

It is obvious to see that the sequential implementation of the above algorithm has $O(n^3)$ where our matrix *A* has *n* x *n* elements. Fortunately, there are multiple areas where we can exploit parallelism. On the low level, we can parallelize the matrix row operation. Because for each element of row *j*, we are adding or subtracting a multiple of another element from row *i,* this element-wise update can be done safely in parallel. In addition, we can also safely apply this row update in parallel (step 2c) across all other rows that are not *i* since row *i* does not change once we make the leftmost element 1 (corresponding to the 1 on the diagonal crossing that row). Following similar logic, the element-wise update done in step 2b can also be parallelized once we store the original value of *A[i][i]*.

To achieve better spatial locality, we stored the 2-dimensional matrices with 1-dimensional representation. While looking for further opportunities to improve performance, we noticed that we can slightly reduce the workload on the number of elements updated in steps 2b and 2c.



The number of elements that actually need to be processed in step 2b



The number of elements that actually need to be processed in step 2c

Because the row operations done for matrix A unit out a column for every row and the identity matrix only updates in a triangular fashion, we can actually reduce the work done from *2n* to *n* in step 2b and reduce the work done from $2n^2$ to n^2 in step 2c.

Inversion through LU-Decomposition

High-level Pseudocode of actual implementation of LU-Decomposition:

1. Begin by allocating square matrices L and U to store results
2. For each column i, we breakdown the process into two parts
   (a) for each row j, we perform the following updates to L, our lower triangular matrix
      i. if $j < i$, we simply write 0 to the cell $L[j][i]$ and continue
      ii. we set $L[j][i]$ to be $A[j][i]$
      iii. for each index $k < i$:
         A. $L[j][i] = L[j][i] - L[j][k] * U[k][i]$
   (b) for each row j, we perform the following updates to U, our upper triangular matrix
      i. if $j < i$, we simply write 0 to the cell $U[i][j$ and continue
      ii. if $j = i$ we set $U[i][j]$ to be 1 and continue
      iii. we set $U[i][j]$ to be $A[i][j]/L[i][i]$
      iv. for each index $k < i$:
         A. $U[i][j] = U[i][j] - L[i][k] * U[k][j]/L[i][i]$

To obtain the matrix inversion, we then follow the procedure previously mentioned in the background section. We take the inverse of the upper triangular matrix *U* and multiply it with the inverse of the lower triangular matrix *L* to obtain the inverse of the original matrix *A.* Because row updates for the two triangular matrices depend on the previously processed columns, we can not parallelize the outer level looping. Fortunately, we identified that the inner row iteration updates can be parallelized as well as matrix operations used for computing A^{-1} from $U^{-1}L^{-1}$. Last but not least, once we obtain the L and U matrices, we can safely compute their inverses in parallel, and the operations within the inverse calculations were parallelized across matrix elements.

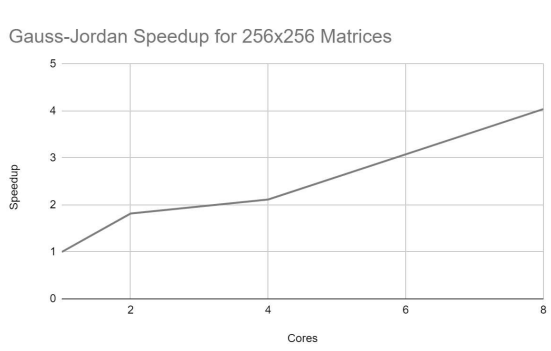
Inversion through Block-Recursion

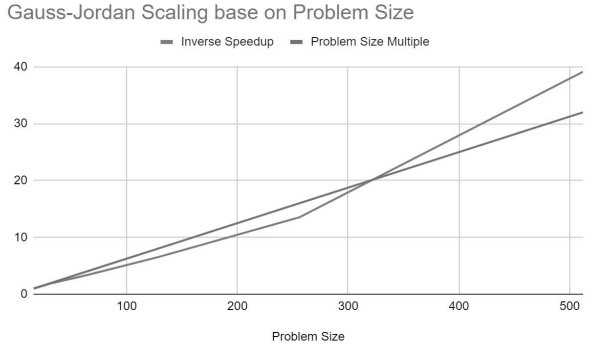
The Block-recursive implementation closely followed the algorithm and formula mentioned in the background section. However, instead of recursing down to a 2x2 invertible matrix base case, we set a threshold where matrix dimensions below it will simply be solved with either Gauss-Jordan or LU-Decomposition. The major source of parallelism comes from the recursive nature of the function, which we utilized OpenMP tasks to exploit. In addition, we parallelized the matrix operations to greatly improve the performance of solving subproblems and temporary variables.

Results

Figures and Interpretations

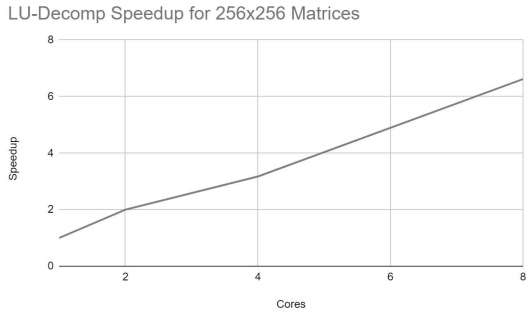
Performance measured in terms of speedup

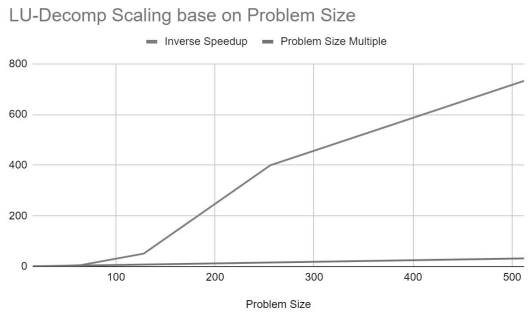




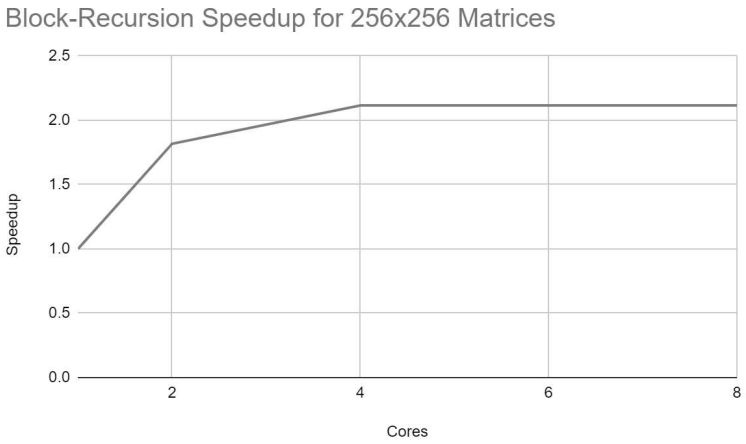
We can see from the collected data that the program scales almost linearly with the number of threads used for parallelization. The slower than expected to perfect speedup is likely due to the overhead from creating parallelization through the various loop segments needed for row operations. In addition, because we essentially need to atomically swap a row when the diagonal

element on the row is 0, that has the chance to make that code segment extremely sequential if the randomly generated matrix requires multiple swaps.





Unfortunately, our implementation does not scale as well given the problem size. When the dimension of the matrix increases by a factor, the total number of elements increases by a factor squared. This problem is probably due to the fact that all the nested loops that are dependent on each other and share variables that constantly change depending on the algorithms of the loops. Hence, this gives little space for parallelization. As a result, the inverse of our speedup (multiple for slowdown compared to the base case) increases by a factor squared in the results.



For the block recursive implementation, we did not observe significant speed up. Despite the problem size breaking down smaller, the performance gain quickly plateaued, likely due to the overhead and limitations from parallelizing the high number of matrix operations and reconstruction.

Limitations and Analysis

Even though we were able to break down the problem size utilizing block recursion, much of the performance was still limited by the nature of the algorithm (matrix operations for reconstruction) rather than the decomposition and elimination algorithms. Because block

recursion only switches to one of the two methods after a set threshold, varying the threshold only brings the performance closer or further to the respective methods. This can be observed from the two extremes: when the threshold is extremely large (larger than the matrix dimension), we go straight into solving the inverse with the two methods rather than block recursion; when the threshold is extremely small (2x2 min), we keep on recursing and only solve with the two methods on the base case. When we originally decided on exploring block-recursion, we had hoped that it can leverage the benefits of breaking down problem size as well as a strong decomposition/elimination method. However, in actual practice, the added matrix operations for reconstructing the original matrix leveled out whatever performance gains from the previously hoped areas. We believe that if we were able to further improve the performances of the matrix operations and reconstruction steps, block-recursion to a certain threshold level can outperform inversion through purely Gauss-Jordan or LU-Decomposition.

References

A Note On Parallel Matrix Inversion
Quintana, Enrique S., Gregorio Quintana, Xiaobai Sun, and Robert van de Geijn. “A Note on Parallel Matrix Inversion.” *SIAM Journal on Scientific Computing* 22, no. 5 (2001): 1762–71. https://doi.org/10.1137/s1064827598345679.

A Method of Ultra-Large-Scale Matrix Inversion Using Block Recursion
Wang, HouZhen, Yan Guo, and HuanGuo Zhang. 2020. "A Method of Ultra-Large-Scale Matrix Inversion Using Block Recursion" *Information 11*, no. 11: 523. https://doi.org/10.3390/info11110523

Work

Week	Date	Tasks
1	Mar 28 - Apr 3	1. Read research papers on parallelizing LU Decomposition and Gauss-Jordan Elimination (Allen/Chi) 2. Setup testing for matrices and implement sequential versions of two algorithms mentioned above (Allen/Chi)
2	Apr 4 - Apr 10	1. Implement Matrix Inversion using Gauss-Jordan (Allen) 2. Implement Matrix Inversion using LU-Decomposition (Chi)

Week	Date	Tasks
1	Mar 28 - Apr 3	1. Read research papers on parallelizing LU Decomposition and Gauss-Jordan Elimination (Allen/Chi) 2. Setup testing for matrices and implement sequential versions of two algorithms mentioned above (Allen/Chi)
3 (checkpoint)	Apr 11 - Apr 17	1. Parallelize Inversion through Gauss-Jordan (Allen) 2. Parallelize Inversion through Lu-Decomposition (Chi) 3. Implement block parallelization useable with both approaches (Allen/Chi)
4	Apr 18 - Apr 24	1. Parallelize inversion through block recursion (Chi) 2. Begin work on Project Report, all sections except performance analysis (Allen)
5 (Final)	Apr 25 - Apr 29	1. Finish performance analysis (Chi) 2. Create materials needed for the final report and presentation (Allen)

The work distribution was 50-50