Date of publication xxxx 00, 0000, date of current version xxxx 09, 2022.

Digital Object Identifier 10.1109/ACCESS.2017.DOI

Parallelization of Linear Equations

BAILIE DELPIRE[1], JOSEPH MAY[2]
[1]Department of Computer Science, Middle Tennessee State University, Murfreesboro, TN 37132 USA (e-mail: bcd3q@mtmail.mtsu.edu)
[2]Department of Computer Science, Middle Tennessee State University, Murfreesboro, TN 37132 USA (e-mail: jam2ft@mtmail.mtsu.edu)

ABSTRACT Linear algebra is a branch of mathematics dealing with vectors and matrices. The methodologies and insights of the field are particularly useful for the computation-heavy problems of today's programs. As the need for data processing has increased and the size of storage structures has grown rapidly, the ability to process these structures and find solutions to linear systems has become more important in fields such as statistics and graphics processing. The goal of our project is to solve a system of linear equations using parallel methods to introduce better speed and efficiency for applied problems in the linear algebra space.

INDEX TERMS Linear Algebra, Parallelization, Programming, Mathematics

I. INTRODUCTION

TO better understand our problem, we looked for research in two main areas, mathematical methods of solving linear systems, and applied parallelization methods and strategies for linear systems.

The literature suggested a number of mathematical algorithms such as Gaussian Elimination, Conjugate Gradient, Sherman-Morrison, Woodberry, and the Newtonian and Chebyshev methods to solve linear equations. Each algorithm comes with unique computational costs and constraints. Our first task was choosing an algorithm that we could feasibly code, but also had acceptable run times and the capability to be parallelized.

After algorithmic selection, our second task was parallelization. We had to decide where parallelization was most appropriate, from the instruction stream, the data stream, or both, based on the algorithm we chose. And from there, we had to decide on messaging protocols, data sharing, and look to optimize our algorithm.

II. SELECTING AN ALGORITHM

There are many algorithms for solving linear systems, and a diverse array of linear systems. A particular algorithm will work well on one type of matrix, but performs poorly on another type of matrix. Our desire was to select an algorithm that performs fairly well on many different types of matrices and be a generically useful linear system solving tool, rather than an optimized tool to solve for a specific matrix configuration.

A. DISCARDING GAUSSIAN

The search for an optimal algorithm began with the familiar sequential Gaussian elimination algorithm. The main reason we pursued Gaussian methods were because Gaussian elimination will arrive at an exact solution for a linear system, and because it was familiar and thus more comfortable to work with. After research and an examination into the Gaussian process, we discovered that Gauss would not be the best algorithm. Gauss requires a lot of matrix processing. The matrix must be in a specific upper triangular form. Firstly the matrix must be manipulated into such a form that the first non-zero entry of each row must be equal to one (known as a pivot), and each pivot must be one column over and one row from the previous pivot, such that the first pivot occurs at [0][0], the second at [1][1], the third at [2][2] etc such that the main diagonal values are all equal to one. Values above the main diagonal may take on any real value. Once the matrix has been processed, values of unknowns can be read starting from the bottom and proceeding up solving for the next unknown in the sequence.

This means that all the results start from the bottom, and the answer to one value depends on the value solved for the one below it. This creates a data dependency in the algorithm, and Gauss must be tweaked or another algorithm chosen to increase parallel gains. The effort to adapt Gauss for parallelism would not be worth it, and so we found an iterative approach that is easily parallelized and works for different data types.

B. JACOBIAN ITERATION

The Jacobian iterative algorithm does not yeild an exact solution. Each iteration will progress the values of the system

closer to the exact answer, and the algorithm will be set to stop once there is an acceptable level of error reached. The time complexity of solving linear systems can be reduced through this approximation.

The Jacobi method has a few requirements. Namely, the matrix must be square, the system must be linear, and the matrix must be non-singular. Our domain is linear systems, so the second requirement is not a drawback but rather a feature! The system in the ideal case, will be diagonally dominant. Diagonally dominant is defined such that for each row, the value on the main diagonal must be greater than or equal to the sum of the other values in the row, and one diagonal value must be strictly greater than the sum of the other values in it's row. In a diagonally dominant matrix, the Jacobi method will always converge! The generic steps of the Jacobi Method are as follows:

1) Guess an estimated value for x values in the matrix, the more accurate the guess, the faster the convergence as less iterations are required.
2) Set all system equations to solve for a different unknown.
3) Plugin the assumed value for the unknown variable into the equations from step.
4) With each new iteration use the value from the previous iteration as the new input.
5) Repeat this step until the error value is within a set threshold tolerance.

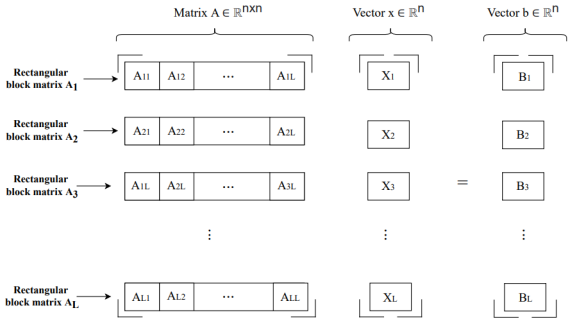

FIGURE 1. The multisplitting method used in [10] splits the matrix A, and vectors x and b into L different blocks, each of which is processed by a group of processors [10].

III. PROJECT OVERVIEW

Our task is to implement Jacobi iteration in code. This will require a list of matrix problems to solve, some preset and others supplied by the user. The array must be validated for Jacobi conditions as discussed above (square, non-singular), and then the data must be interpreted. Each row must be adjusted to solve for one unknown value, and we must assume some value for each unknown. After that, the iteration can happen to solve for the value of each variable. In the initially stages, the program will be implemented sequentially to validate accuracy and correct implementation of the Jacobi method. From there we apply parallel methods learned in class to improve the speed of the algorithm. The likely target of parallelization is the iterative solve loop where we can use threads for each different equation in the system and hopefully achieve significant gains in speed.

IV. CREATING A DATASET

The goal of this project is to parallaelize and optimize the Jacobi Iteration algorithm to solve linear systems. In order to see any gains from parallelization the linear system must be large enough that we can notice parallel gains. To this end, we set about creating large linear systems to feed into our Jacobi algorithm.We broke the data set creation into 3 different functions, and also made another function to read in array data.

A. DIAGONAL DOMINANCE

Jacobi Iteration will only converge for systems that are diagonally dominant. An array is defined to be diagonally dominant is the diagonal value is greater than the sum of other values in it's row. Before attempting arrays not guaranteed to converge, we want to test out parallelization methods on systems we know have answers. As such, we implemented a function that checks for diagonal dominance in a given array. The function prototype looks like

```
bool checkDiagDominance(int
    arr[NUMROW][NUMCOL], int row, int col);
```

Given an array, and its row and column dimensions, the function will check to see that every diagonal value is greater than the sum of other values in its row. In practice for large non-sparse arrays, this often means that diagonal values are substantially larger than other row values.

B. CREATING THE LINEAR SYSTEM

Another function we created will make an array of predetermined size with random numbers 0-7 (negative values allowed). It's function prototype looks like:

```
void createRandArray(int row, int col, int
    arr[NUMROW][NUMCOL], int* sumArr);
```

NUMROW and NUMCOL parameters are defined as global variables, changing these will change the size of the array that is made. Given that Jacobi only converges for diagonally dominant arrays, the function is currently setup to only create such arrays. If there is time after parallelization and optimizations, we can revisit non diagonally dominant arrays.

C. WRITE ARRAY TO FILE

This function simply accepts the newly generated random array and writes the array information into a file. The first line of the file contains row number, a space, then column number. Every line after that contains array data in the form of value space value. Each line of the file is equivalent to one row of the matrix. Because the files first line is row and column number this means the total number of lines in

the file should be rows + 1. The filename is of the format matrixsize + Matrix + random number. So if the matrix is a 500 by 500 matrix, the file name would be 500Matrix12. The ending number is a random value so that the program can be run multiple times and create new array files, rather than overwrite the same file with different data.

D. CREATE ARRAY FOR JACOBI ITERATION

This is a function for our main program. It will read in the column and row size of the array from some file input. It will dynamically allocate an array of the appropriate size, and add the values into the array. Additionally it will generate three other arrays. One array to hold the right hand side values.One array to hold guess values. Lastly, one array to hold the final answer values for parameters. A typical linear equation takes the form 2x + 3y = 6. Once this function is done there will be an array with values [2 3] which is the array containing the matrix coefficients, an array containing [0 0] which is the initial estimated value for variables x and y, an array with value(s) [6] which is the right hand side of the equation and what we will use to compare the accuracy of variable estimates, and finally an uninitialized array of row size which will hold the final answer from Jacobi iteration.

V. WHERE WE ARE GOING

By the next time we report, we hope to have successfully parallelized the algorithm and to have performed some data analysis on how much faster parallel is compared to sequential. And if time permits, add in more matrix capabilities such as bigger values, more sparse matrices, and compare different parallel implementations.

REFERENCES

[1] V. P. Gergel, "9. Parallel Methods for Solving Linear Equation Systems," University of Nizhni Novgorod. [Online]. Available: http://www.hpcc.unn.ru/mskurs/ENG/PPT/pp09.pdf

[2] S. C. S. Rao and R. Kamra, "A computational technique for parallel solution of diagonally dominant banded linear systems," in 2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC), Dec. 2021, pp. 448–453. doi: 10.1109/HiPC53243.2021.00064.

[3] H. Gu, Y. Luo, Y. Qiu, and J. Hou, "A Fast Solution Method for Large Scale Linear Sparse Equations Based on Parallelism," in 2022 IEEE 5th International Conference on Electronics Technology (ICET), May 2022, pp. 1337–1340. doi: 10.1109/ICET55676.2022.9824027.

[4] V. Pan and J. Reif, "Efficient parallel solution of linear systems," in Proceedings of the seventeenth annual ACM symposium on Theory of computing, New York, NY, USA, Dec. 1985, pp. 143–152. doi: 10.1145/22145.22161.

[5] S. Maruster, V. Negru, and L. O. Mafteiu-Scai, "Experimental Study on Parallel Methods for Solving Systems of Equations," in 2012 14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, Sep. 2012, pp. 103–107. doi: 10.1109/SYNASC.2012.73.

[6] Q. Chunawala, "Fast Algorithms for Solving a System of Linear Equations | Baeldung on Computer Science," Apr. 22, 2022. https://www.baeldung.com/cs/solving-system-linear-equations (accessed Sep. 02, 2022).

[7] T. J. Dekker, W. Hoffmann, and K. Potma, "Parallel algorithms for solving large linear systems," Journal of Computational and Applied Mathematics, vol. 50, no. 1, pp. 221–232, May 1994, doi: 10.1016/0377-0427(94)90302-6.

[8] L. Yao, X. Ji, S. Liu, and J. Yang, "Parallel Implementation and Performance Comparison of BiCGStab for Massive Sparse Linear System of Equations on GPU Libraries," in 2015 IEEE 12th Intl Conf on Ubiquitous Intelligence and Computing and 2015 IEEE 12th Intl Conf on Autonomic and Trusted Computing and 2015 IEEE 15th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom), Aug. 2015, pp. 603–608. doi: 10.1109/UIC-ATC-ScalCom-CBDCom-IoP.2015.119.

[9] R. Peng and S. Vempala, "Solving Sparse Linear Systems Faster than Matrix Multiplication," in Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA), Society for Industrial and Applied Mathematics, 2021, pp. 504–521. doi: 10.1137/1.9781611976465.31.

[10] M. A. Tchakorom, R. Couturier, and J.-C. Charr, "Synchronous parallel multisplitting method with convergence acceleration using a local Krylov-based minimization for solving linear systems," in 2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), May 2022, pp. 900–906. doi: 10.1109/IPDPSW55747.2022.00146.