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.

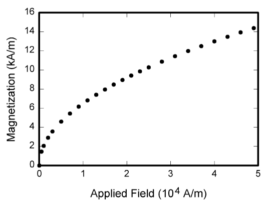


FIGURE 1. The multisplitting method used in [10] splits the matrix A, and vectors x and b into L different blocks, each of which is processed by a group of processors [10].

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 initial 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 out to create a 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 if 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 makes an array of predetermined size with random numbers between 0 and 7, with negative values allowed. Its 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 set up 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 file's 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. METHODS

A. DIVIDING THE PROBLEM

The data structures we used for holding the data, matrices, are natural candidates for parallel processing. The typical methods for this are to divide the matrices into a number of rows or columns depending on whether the language implementing the solution uses row-major order or column-major order matrix structures. As we implemented our solutions in C and C++, we divided our matrices up into groups of rows. Figure 2 shows what a matrix divided into groups of rows might look like.

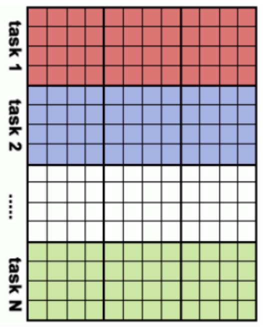


FIGURE 2. Dividing a matrix into groups of rows.

This can easily be done by dividing the number of rows in the matrix by the number of threads or processes that will be used. For example, if the matrix has 100 rows and 4 processes will be used, each process will operate on 25 rows of the matrix.

B. SERIAL

Before implementing any parallel solutions, we implemented a serial solution to use a a baseline. The serial solution takes the following steps to find a solution to a system of linear equations using the Jacobi method.

1) Read in the randomly generated, diagonally dominant coefficient matrix from a file, randomly generate a constant matrix, and initialize a variable solution matrix with all zeros.
2) Loop to find a solution for each row in the system.
3) Calculate the error for each solution.
4) Find the maximum error.
5) Compare the maximum error to the desired error threshold, epsilon
6) Break if the error is less than epsilon, otherwise continue searching the solution space.

Shown below is the function written to update the solutions. This function is used across all implementations, with different methods employed for deciding which rows to operate on and how the data is sent back out of the function.

```
updateSolution(arguments) {

   // decide which rows will be handled
   // will be different for each
      implementation
   ...

   // find solutions
   for (i=begin; i <= end; i++) {
      sum = 0;
      for (j=0; j < N; j++) {
         if (i != j)
            sum += matrix[i][j] * xOld[j];
      }
      x[i] = (b[i] - (sum)) / matrix[i][i];
   }
   // calculate the maximum error for any
      of the equations
   for (int i=begin; i <= end; i++) {
      error = b[i];
      for (j=0; j<N; j++) {
         error -= matrix[i][j] * x[j];
      }
      if (abs(error) > lmax) {
         lmax = error;
      }
   }
   // send the maximum error out
}
```

C. PTHREADS

Threads are independent streams of instructions that can be scheduled to run independently by the operating system. POSIX Threads is an API defined by the standard POSIX.1c and used for the creation and control over these threads.

For our Pthread solution, we create a number of threads, with each finding solutions for its group of linear equations. When each thread has finished finding a solution, the threads are joined, and we check if the maximum error their solutions had is less than our error threshold. If it is, we stop searching for solutions, otherwise we continue. The code for this process is shown below.

```
while (1) {
   for (i=0; i < num_threads; i++) {
      pthread_create(&pids[i], NULL,
          updateSolution, (void *) (long
          int)i );
   }
   for (i=1; i < num_threads; i++) {
      pthread_join(pids[i], NULL);
   }
   if (max_error <= epsilon) {
      break;
   }
}
```

The update solution function, shown previously, decides which rows the thread will be operating on, finds solutions for those rows, and updates the solution matrix. Then it finds the maximum error out of all the equations it operated on.

In Pthreads, a mutex variable takes the role of a “lock”, protecting access to a shared data resource. We use the maximum error as a mutex variable and use mutex lock and unlock in the updateSolution function to safely update the max error variable, as shown below.

```
   max_eps[lcount] = lmax;
   pthread_mutex_lock(&lock_m);
   if (max_eps[lcount] >= max_error) {
      max_error = max_eps[lcount];
   }
   pthread_mutex_unlock(&lock_m);
```

D. OPENMP

OpenMP, or Open Multi-Processing, is an API that is used to explicitly direct multi-threaded, shared memory parallelism. OpenMP is very easy to implement over a serial implementation. For this we use the pragma omp parallel directive, shown below.

```
#pragma omp parallel
    num_threads(numThreads)
    private(threadID, i, j, sum, error,
    begin, end, rowStep) shared (maxerror)
```

This makes the variables threadID, i, j, sum, error, begin, end, and rowStep all private to a particular thread and the variable maxerror shared among threads. Following this directive, the updateSolutions function can determine which rows a thread will operate on based on its thread number, and while the maximum error is larger than epsilon, it will look for solutions and calculate their errors. We use the critical directive to safely update the maximum error for each iteration and we use the barrier directive to make sure that the threads are synchronized and do not move on to the next iteration before all threads have updated their solution matrices. This process is shown below.

```
#pragma omp critical
{
   if(maxeps[threadID] >= maxerror)
      maxerror = maxeps[threadID];
}
#pragma omp barrier
```

E. MPI

MPI, or Message Passing Interface, is a specification for the use of message passing libraries. It addresses a parallel programming model where data is moved from the address space of one process to that of another process through cooperative operations on each process. To begin an MPI implementation, the following lines of code are used.

```
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&numranks);
MPI_Comm_rank(MPI_COMM_WORLD,&myrank);
```

In our MPI implementation, we have each worker process running the same updateSolution function to find solutions for a number of rows in the matrix. The max error of those solutions is returned. Then we use MPI Reduce to find the maximum error from all the processes and use MPI Broadcast to broadcast this error to the main process. Again, if the error is less than epsilon we can stop searching. The worker process is shown below.

```
 if (myrank != 0) {
   rowCount = M / (numranks - 1);
   ...
   while (1) {
      error = updateSolution(matrix, b, x,
          xOld, rowCount, myrank, numranks,
          N);
      ...
      MPI_Reduce(&error, NULL, 1,
          MPI_FLOAT, MPI_MAX, root,
          MPI_COMM_WORLD);
      MPI_Bcast(&error, 1, MPI_FLOAT, root,
          MPI_COMM_WORLD);
      MPI_Barrier(MPI_COMM_WORLD);
      if (error < epsilon) {
         break;
      }
   }
```

The main process uses MPI Reduce to find the global maximum error. If the global maximum error is less than epsilon, the desired solution has been found and the program terminates. The main process is shown below.

```
} else {
   while(1) {
      MPI_Reduce(&lerror, &maxerror, 1,
          MPI_FLOAT, MPI_MAX, root,
          MPI_COMM_WORLD);
      MPI_Bcast(&maxerror, 1, MPI_FLOAT,
          root, MPI_COMM_WORLD);
      MPI_Barrier(MPI_COMM_WORLD);
      if (maxerror < epsilon) {
         printf("# Iterations: %d\n", step);
         t2 = time1();
         printf("\n\nTOTAL TIME %f\n",
             t2-t1);
         break;
      }
```

```
        step++;
    }
}
MPI_Finalize();
```

VI. RESULTS

A. METHODOLOGY

To evaluate our programs, we recorded and charted the run times of each implementation. Run time started after each program read in the matrix file and initialized the arrays, and stopped once the epsilon point was reached. The run time of each program is printed to standard output. When testing this output was sent into a file using a shell command.

In order to maintain standards, all parallel implementations were tested using the matrix 1000Matrix20.txt. This is a non-sparse array of size 1000x1000.

Each implementation was tested with 4 thread values: 1,2,4,8. This is enough threads to gather a good trend to examine if the program speeds up as more threads are added.

Each implementation was run 100 times for each possible thread number. For example, the pthread implementation ran 100 times with 1 thread, 100 times with 2 threads, 100 times with 3 threads, 100 times with 4 threads. The results of each test were sent into a file. The results of pthread with 1 thread was stored in pthread-1.txt, pthread with 2 threads was stored into pthread-2.txt, etc. The end result of these tests thread[1-4].txt were compiled into a single file. This process was repeated for both openMP and MPI implementations.

The following shell command is an example of how to run a test like this.

```
For val in 1 2 4 8; do ((x=0; x<100; x++));
    do ./openMP ${val} 1000Matrix20.txt >>
    mpi-Data-${val}.txt; done; done
```

val in 1 2 4 8 represents the number of threads to pass into each implementation. The second loop using variable x ensures each program is run 100 times. The » is a shell command which takes the output of the program and appends it to a file.

Each test generates four files, one for each thread. An example shell command to compile all the thread files into a single file is:

```
 ls pthread*.txt | xargs paste >
     pthreadtotal.txt.
```

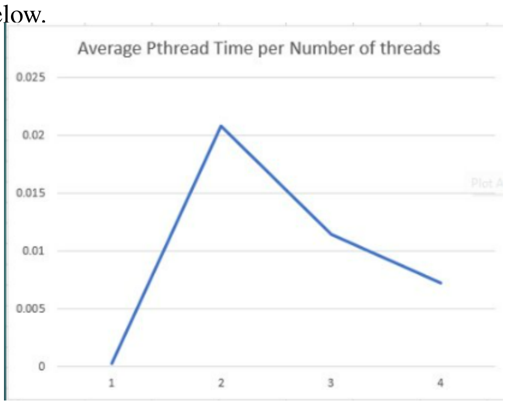
ls pthread*.txt lists all files in a directory beginning with pthread and ending in .txt, then the command xargs paste combines all those files with a tab separating each value and stores that tab separated output in a file called pthreadtotal.txt.

Once all the data had been compiled, it was exported into Microsoft excel, where I gathered the average results of each thread, for each implementation. These results were graphed and are shown below.

One small error in each graph is that the thread axis has been mislabeled. Each graph has numbers 1,2,3,4. This is incorrect. The 3 on each graph represents the time for 4 threads, and the 4 of each graph represents the time for 8 threads. Apologies for the labelling error.

B. PTHREAD

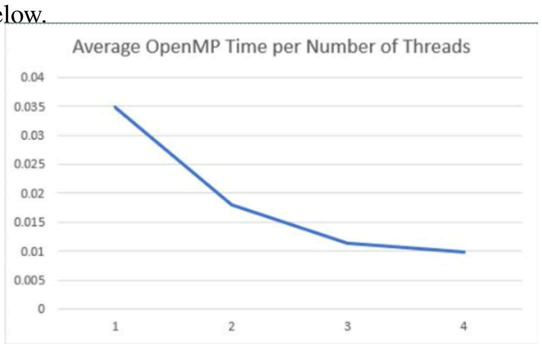
The average time for our pthread implementation is shown below.



We would expect the time it takes to process the matrix to go down as more threads are added as more threads mean the work should be done faster. This is true as the graph extends outwards to 2, 4 and 8 threads. So 3/4 of the graph meet our expectations. However, this graph has an atypical trend line in the first section where the program is run using 1 thread. Interestingly, the lowest time is achieved with 1 thread. This runs counter to our assumptions that more threads equates to a faster time. This tells us that while adding threads does increase the efficiency at a certain point, that our parallelization is ultimately not efficiently implemented because a single thread outperforms multiple threads. Ignoring the threads centered data, and looking to just the time data, the pthread implementation performed remarkably well. The highest average time for any thread number being .021, which is exceptionally fast to process an array of 1000x1000. Other than this abnormality at 1 thread, the graph behaves in line with our predictions.

C. OPENMP

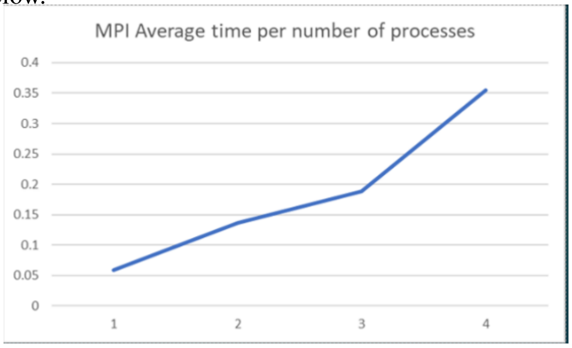
The average time for our openMP implementation is shown below.



This graph behaves much more in line with our expectations than the pthread graph. It begins with the highest execution time at 1 thread, and as more threads are added the average execution time of the program goes down. This aligns perfectly with our hypothesis about more threads making for greater efficiency. The OpenMp implementation performs comparably to the pthread implementation. The maximum average time taken is .035, which is .014 seconds longer than the slowest time of pthread, but at such small fractional seconds the time difference is unlikely to matter. At its fastest the openMP program achieved an average time of .01 seconds with 4 threads, which is exceptionally fast for a large 1000x1000 array.

D. MPI

The average time for our MPI implementation is shown below.



Unlike our pthread and openMP implementation, this graph never behaves the way we would expect. It performs best with just one process, and almost linearally performs worse as more processes are added. This is an indication that our MPI implementation needs to be reworked. The pthread and openMP implementations are close mirrors of one another, but our MPI implementation is the most unique implementation that we have. Such dramatic slow downs in execution time indicate that the program is not setup to handle more processes. However, this implementation is still very fast. Its fastest average time was .05, and its slowest average time was .037, which is quite similar to the fastest and slowest times of our openMP program, albeit the MPI implementation achieved these times at unexpected ends of the graph.

E. RESULTS SUMMARIZED

All implementations ended up solving a 1000x1000 matrix in almost no time at all even at their slowest average times. Both the pthread and MPI achieved their fastest executions times at only a single thread, which indicates that our parallelization of those implementations is imperfect as a single thread can outperform multiple threads. The openMP implementation fit our hypothesis perfectly and showed that more threads can increase the speed of a program, and increased the execution time of the Jacobi Iteration near linearally as the number of threads was doubled. Our MPI program was the outlier amongst all three implementations, not in the time it took to execute but in how it completely provided evidence against our hypotheses as adding threads slowed down the program. Regardless of the thread trends, each program achieved a similar speed profile, with pthread performing the fastest overall.

VII. FUTURE WORK

Given more time on this project areas of research we would pursue are:

1) Improving parallelization. We would rework how threads communicate and synchronize in order to reevaluate the execution time of all implementations until the average run time of each program meets with the expected result that more threads results in a faster run time, or until we can conclude that there are some barriers inherent to the nature of array processing / Jacobi iteration that inhibit this expected result.
2) Compare Jacobi Iteration against different types of matrices. In addition to making multiple implementations, we also generated multiple types of matrices, such as sparse, very-sparse, and upper-triangular. Given more time we would conduct a study to see which type of matrix is most conducive to increasing the speed of convergence for Jacobi Iteration.
3) Examining algorithm performance for non diagonally dominant matrices. The matrices we used for testing and generated are all guaranteed to have a find-able result. This is useful for bench marking the algorithm, but doesn't capture the performance of Jacobi Iteration as it would perform in a broad context. Real life data matrices don't come pre-scrubbed and formatted into diagonally dominant arrays. To establish how Jacobi Iteration could perform in a real work context, we would need to examine how the algorithm performs on data that may not contain a solution that can be solved iterably, and generate ways to handle those edge cases to make a more robust linear equation solver.

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 Ap-

plied 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.