Fall 2022 ME/CS/ECE759 Final Project Report
University of Wisconsin-Madison

Several Linear Solver's Parallel implementations

Zeyu Wang

December 14, 2022

Abstract

Parallel computing has emerged as a powerful tool for solving large-scale linear equations, which are commonly found in many scientific and engineering applications. Linear equations are fundamental to a wide range of problems, including data analysis, optimization, and modeling. However, their solution often requires significant computational resources, especially for large-scale problems with millions or billions of equations. Parallel computing provides a means to harness the power of multiple processing units to accelerate the solution of linear equations and reduce the computation time.

In this paper, we present a parallel algorithm for solving linear equations that combines the Jacobi method and LU decomposition. The Jacobi method is used as an iterative solver to efficiently approximate the solution of the linear equations, while LU decomposition is employed as a direct solver to accurately compute the exact solution. We implement our algorithm using different programming frameworks such as CUDA, OpenMP, cuBLAS, and thrust to achieve high performance and scalability on a wide range of architectures and problem sizes. Our results show that our approach is highly efficient and scalable, achieving significant speedups and good parallel efficiency on a wide range of problem sizes and architectures. Overall, our work provides a promising solution for solving large-scale linear equations using parallel computing.

Link to Final Project `git` repo:
https://github.com/Larrywangustc/HPC/upload/main/FinalProject759

Contents

1. Problem statement

I will accomplish Linear Solvers. Solving linear systems with high accuracy and time efficiency is an important aspect of many computational engineering applications. Jacobi method and LU decomposition are commonly used numerical methods to solve systems of linear equations, and can be optimized and accelerated by taking advantage of the parallel computing frameworks

2. Solution description

There are several implementations of different method on different frameworks. I will list them below and compare them in next part.

1. The Jacobi method is an iterative method for solving systems of linear equations, and it involves repeatedly updating the solution vector according to the following formula:

   x_new[i] = (b[i] - sum(A[i][j] * x_old[j] for j != i)) / A[i][i]

   where `A` is the matrix of coefficients, `b` is the vector of constants, and `x` is the solution vector. This process is repeated until the solution converges or a maximum number of iterations is reached.

   In ***Jacobi_cuda.cu***, I accomplished two versions of Jacobi method with CUDA by define two kernel functions.
   The two kernel functions perform the same basic operation: computing the next iteration of the Jacobi method for a system of linear equations. However, they do it in slightly different ways.

   The first kernel, jacobiOnDevice_v1, takes four arrays as inputs: x_next_u, A_u, x_now_u, and b_u. The kernel then uses a loop to iterate over the elements of the arrays and compute the next iteration of the Jacobi method using the input data. The result is stored in the array pointed to by x_next_u.
   [1]The second kernel, jacobiOnDevice_v2, is similar to the first, but it uses shared memory to store and manipulate some of the input data. It also uses a constant memory array, b_s, which is a copy of the b_u array from the first kernel. The kernel first copies the data from the d_x_now and d_x_next arrays into shared memory arrays, xdsn and xdsx. It then uses a loop to compute the next iteration of the Jacobi method using the shared memory arrays and the d_A array. The result is stored in the shared memory array xdsx. Finally, the kernel copies the data from the shared memory array back into the d_x_next array in device memory.

Both kernels compute the solution to the system of linear equations using the Jacobi method, which involves iteratively updating the solution vector x using the formula above. This process is repeated until the solution converges or a maximum number of iterations is reached. In order to concentrate on the computational cost of different implementation ways, we do not focus on when will the iteration stops, and we will set a fixed time of iterations. This is mainly because when to stop is a pure math problem, which is not what we want to talk about here.

2. In ***Jacobi_omp.cpp***, I finished a sequential implementation of the Jacobi method for solving systems of linear equations, JacobiOnHost.

   The jacobiOnHost function is parallelized using the #pragma omp parallel for directive, which tells the OpenMP library to run the loop in parallel using the specified number of threads. Inside the loop, the jacobiOnHost function is called to update the solution vector using the Jacobi method, and then the solution vector is copied to x_now for the next iteration.

3. In ***LU_cuda.cu***, I use the CUDA framework to perform the LU decomposition in parallel on the GPU. CUDA is a parallel computing platform and programming model that allows for efficient execution of parallel jobs on GPU.

   The lu_decomposition_kernel kernel is executed with a grid of thread blocks, where each block is composed of threads that compute the L and U matrices for a submatrix of the input matrix. The kernel uses a shared memory space to store partial sums that are used in the calculation of the L and U matrices.

   The luDecompositionOptimized_kernel kernel is like the lu_decomposition_kernel kernel, but it optimizes the calculation of the L and U matrices by using a tiled approach. In this approach, the input matrix is divided into smaller tiles, and each thread block processes a tile at a time. This can improve the performance of the kernel by reducing the number of memory accesses and increasing the amount of data that can be kept in the fast on-chip shared memory.

   Both kernels use a loop to iterate over the elements of the input matrix and compute the L and U matrices using the algorithm for LU decomposition. The resulting L and U matrices are stored in separate arrays that are passed to the kernel as arguments.

4. In ***LU_omp.cpp***, I use OpenMP to perform LU decomposition on a square matrix A of size n. The resulting lower and upper triangular matrices are stored in L and U, respectively. The number of threads to use is specified by the t command-line argument. The code also measures and prints the time taken to perform the decomposition.

5. In ***LU_cublas.cu***, the code appears to be a simple implementation of a batch LU decomposition using the CUBLAS library. It takes in two command-line arguments: n and batchSize. n is the

size of the square matrices that will be decomposed, and batchSize is the number of matrices to be decomposed.

We first initialize a CUBLAS handle, which is required to call any CUBLAS functions, and then allocates memory for the matrices that will be decomposed. Next, we allocate memory for several arrays that will be used in the LU decomposition: Aarray will hold pointers to the matrices that will be decomposed, ipiv will hold the pivot indices for each matrix, and info will store status information about the decomposition. The matrices in Aarray are then set to point to the appropriate locations in the A matrix.

Then we record the starting time and performs the batch LU decomposition using the cublasSgetrfBatched function. This function takes in the handle, the size of the matrices, the array of matrices to be decomposed, the leading dimension of the matrices, an array to store the pivot indices, an array to store status information, and the batch size.

6. In ***LU_thrust.cu***, the code computes the LU factorization of a square matrix A of size n. The matrix A is initially generated with random values and then copied to the device memory. U and L are initialized to 0 and copied to the device memory. Then, thrust::transform() is used to compute the LU factorization of A and store the resulting matrices L and U in the device memory. Finally, the time taken to compute the factorization is printed to the console.

   We use the Thrust library, which is a parallel algorithms library that is built on top of the CUDA programming model. This allows the code to run on an NVIDIA GPU to take advantage of its parallel processing capabilities. use Thrust's transform function to compute the LU decomposition of the matrix A in parallel on the GPU, and then copies the resulting matrices L and U back to the host for printing.

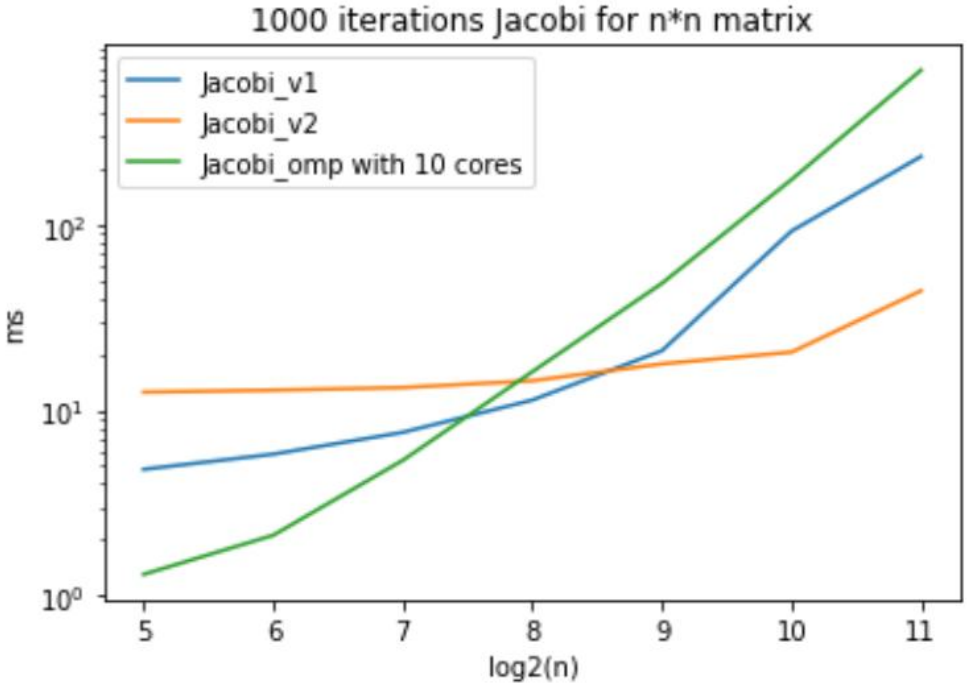
3. Overview of results.

To summarize the results, I generate several plots to explain them including comparison in different ways.

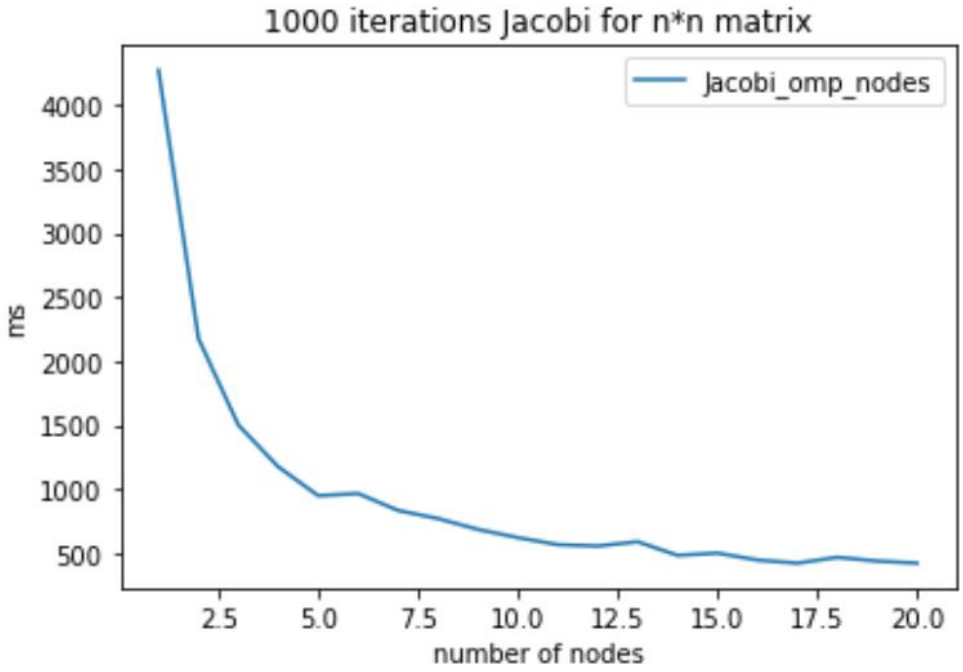
1. First, we compare the time cost of different ways to implement Jacobi method. Since the convergence condition is pure math problem, we do not need to concentrate on the iteration times. We fixed the iteration times to 1000, and we generate the ms-log2(n) plot for three different ways below.

   We can observe that, the updated version of Jacobi method on cuda has the lowest time cost than Jacobi_v1 while size of matrix became bigger. And we use Jacobi_omp with 10 cores CPU to compare the time cost with CUDA version.

In general, the tiled version of the Jacobi method implemented in the jacobiOnDevice_v2 kernel may perform better for larger matrix sizes because it takes advantage of the memory hierarchy on the GPU by storing a portion of the input matrix in shared memory. This can reduce the number of global memory accesses, which can improve performance. However, for small matrix sizes, the overhead of tiling the matrix and using shared memory may outweigh the performance benefits, which could make the jacobiOnDevice_v1 kernel faster.

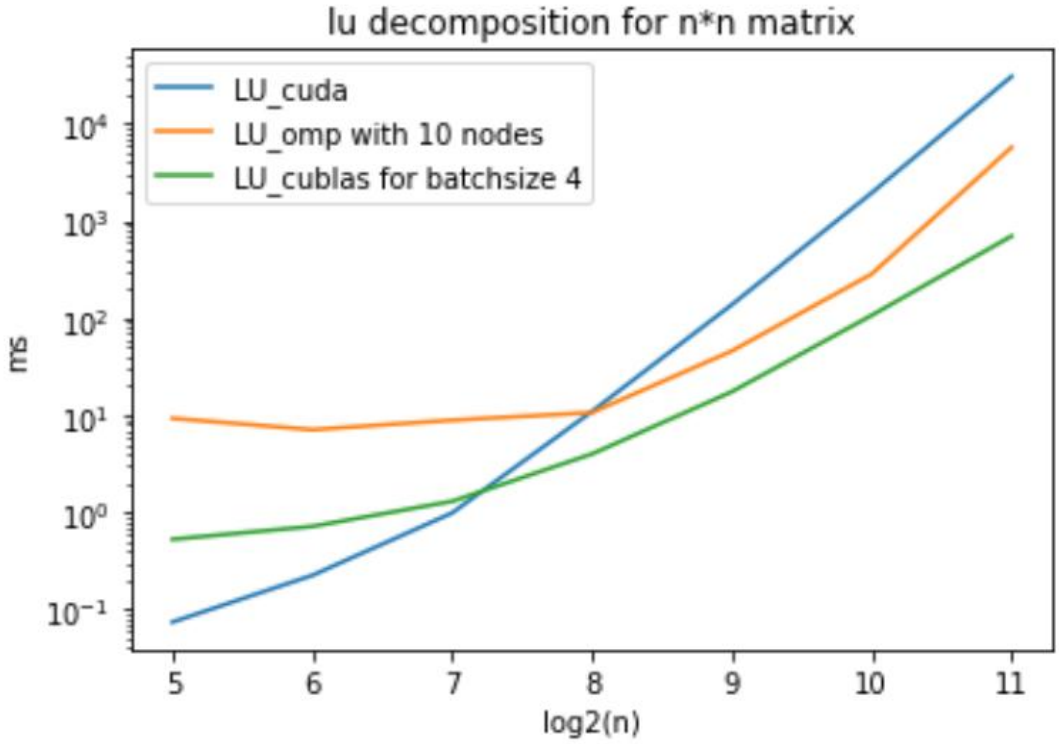


2. Then we generated a plot to show how the time cost decrease while the number of CPUs used in OpenMP. We have done the similar comparison in homework. The result can be observed as below.

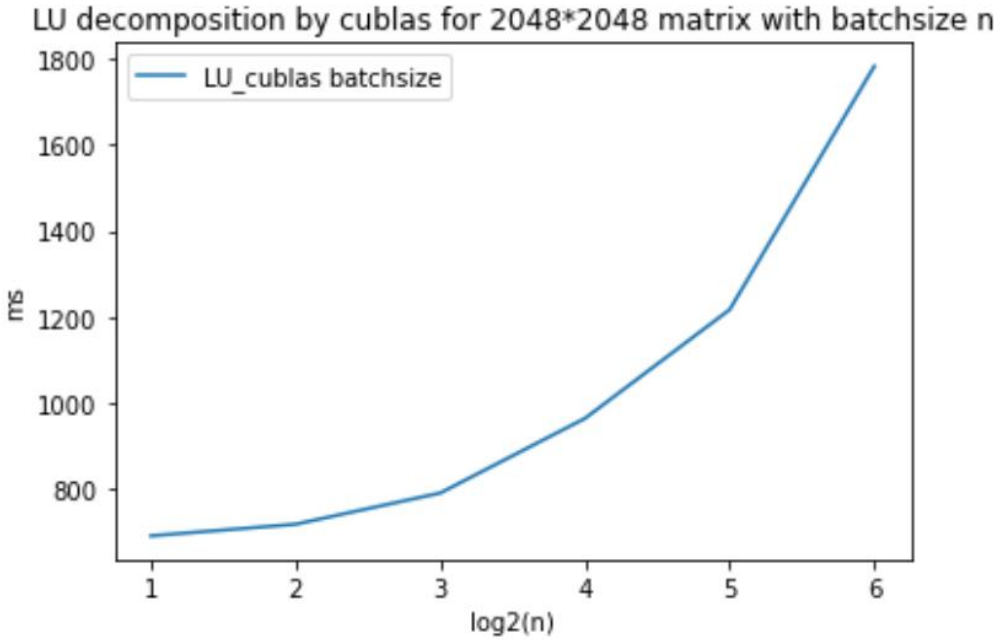


3. We compare the time cost of different ways to implement LU decomposition, which include a simple implementation of CUDA, OpenMP with 10 CPUs, and implementation of batch by cuBLAS. For solving linear system by LU decomposition, the time cost of LU decomposition is O(n^3), and solving the lower/upper triangular system is O(n^2), so we only implement LU decomposition and use the time cost of LU decomposition to represent the speed of solving linear system by LU decomposition.

   cuBLAS is designed to be a highly optimized library for performing common linear algebra operations on GPUs. It is written in low-level languages like CUDA and C++, which allows it to take advantage of the full capabilities of the GPU hardware. It is also heavily parallelized, which allows it to take advantage of the many cores on modern GPUs. Additionally, it is regularly updated and optimized by a team of experts in Nvidia, which helps to ensure that it remains one of the fastest ways to perform linear algebra on a GPU.



4. In the plot below, we generate a plot to compare the time cost with the different batch size. The function we used in LU_cublas, cublasSgetrfBatched is a function in the CUBLAS library that performs a batched LU decomposition on a set of square matrices. We found that the time cost increases is not as faster as the batch size increases. Note that the x axe is log2(n) where n is the batch size.



4. Deliverables:

My FinalProject759 folder has a clear structure, we can find the name of the file representing the method and implementation method, such as Jacobi_cuda.cu. And for file name ***.cu/cpp, the compile and execution command can be easily found in ***.sh, and for LU decomposition, the output file is Slurm_***.out, and for Jacobi method, the output file is Slurm_***_J.out. And error message can be all found in Slurm_err. CPU_lib provides other potential useful numerical method implemented by cpp.

5. Conclusions and Future Work

I implemented Jacobi and LU decomposition in many ways of parallel computing, some of them is based on the method we learned in class, some of them is newly learned besides coursework. For example, Jacobi_v2 is implemented by ideas of tile the matrix and improve the performance significantly. And batched cuBLAS implementation is used to decompose a batch of matrix together. These are all related to ME759 material.

There are some other things that is not finished in my work. The first one is I want to update a new version of LU decomposition by CUDA but failed to make it right. The kernel can not be executed successfully may caused by the index is over the range, I am not sure about the reason here. The second one is I try to implement LU_thrust, but use thrust::transform, this part takes me a lot of effort but still can not run successfully. This part is quite different with call a simple thrust function compared to what we done in homework, but is use thrust structure to do what we what. These are two parts we have not finished in this project.

References

[1] https://forums.developer.nvidia.com/t/cuda-jacobi-method-using-shared-memory/69113/3