A Parallel Jacobi Gauss-Seidel Method with Dynamic Parallelization

Nirav C. Pansuriya
School of Computer Science
Carleton University
Ottawa, Canada K1S 5B6
niravchhaganbhaipan@cmail.carleton.ca

December 21, 2021

Abstract

Methods for solving large system of linear equations have always been a topic of interest for researchers. The Jacobi and Gauss-Seidel methods are both very well-known methods for solving linear equations. While the Jacobi method is easy to implemented in a parallel environment, it is extremely slow at solving equations; on the other hand, the GS method is extremely quick at solving equations but extremely complex to implement in a parallel environment. To address this issue, the PJG (Parallel Jacobian Gauss-Seidel Method) was introduced. This method is capable of solving large system of linear equations in small number of iterations and is extremely simple to apply in a parallel environment too. The primary objective of this research is to further enhance the performance of the PJG method by implementing the concept of dynamic parallelization. Additionally, performance of the proposed method (PJG method with dynamic parallelization) is compared with the Jacobi method, the GS method, and the PJG method by implementing these algorithms in a parallel system.

1 Introduction

Computer architecture has become increasingly parallel in recent years. Modern GPUs, such as the NVIDIA GForce GTX 280 provide enormous parallelism. So, the focus of researchers has moved to parallelism rather than continuous improvements to the single unit speed of computation. We regularly encounter several linear systems in physics, mathematics, and engineering. In many scientific simulations, we have to solve large-scale linear equations. However, large-scale linear equations require a significant amount of time and resources, such as memory. It is always a trending topic among researchers to find an algorithm that can solve large-scale linear equations in less time and with fewer resources. Many scientific methods, such as iterative and direct methods, exist for solving large-scale, computationally expensive linear systems of equations. When solving large systems of linear equations, iterative methods are typically preferred over direct methods. This is because direct methods are very computationally expensive for large linear systems, while iterative methods require lower memory and shorter execution times. The Jacobi and Gauss-Seidel methods are two famous and well-known iterative methods for solving systems of linear equations. Although the Jacobi approach is extremely simple to implement in a parallel environment, it requires an excessive number of iterations to solve a large system of linear equations. The Gauss-Seidel method is an improved version of the Jacobi method. The GS method is capable of solving a large system of linear equations in a small number of iterations. However, because this method is sequential in nature, it is extremely difficult to implement in a parallel environment. One method is easily implemented in a parallel environment but requires an excessive number of iterations to solve, whereas the other method can solve a large system of linear equations in a very few iterations but cannot be implemented in a parallel environment. To address this issue, the PJG (Parallel Jacobian Gauss-Seidel Method) was introduced. This method is capable of solving large systems of linear equations in a small number of iterations and is extremely simple to apply in a parallel environment too. The primary objective of this research is to further enhance the performance of the PJG method by implementing the concept of dynamic parallelization. I compared the proposed method (PJG method with dynamic parallelization) with the Jacobi method, the GS method, and the PJG method. I implemented all of these algorithms in CUDA and executed them on a GPU in order to compare their performance with the proposed technique. Among the Jacobi method, the GS method, and the PJG method, the PJG method took the least time to solve a large system of linear equations. The PJG method achieved up to a 7x faster speed compared to the GS method. I solved the same system of linear equations using my proposed method. Results indicate an improvement in runtime, reaching up to a 10x faster speed compared with the GS method and up to a 1.3x faster speed compared with the PJG method.

2 Literature Review

Large linear equation systems are used in many complex scientific simulators. Solving such huge systems requires a significant amount of processing resources, such as memory and CPU. Solving a large system of linear equations is a very time consuming procedure. That is why developing effective algorithms to solve such big systems in a limited amount of time and with limited resources is always a research interest.

There are two approaches to solving linear equations: a direct approach and an iterative approach. In a direct approach, equations are solved in finite steps, but the time complexity is $O(n^3)$ [8], which is quite high when the system of linear equations is large. This is because direct approaches scale intensively with respect to the size of system of linear equations. The indirect technique has a time complexity of $O(n^2)$. The variables in the linear equation are initially initialized with random values in an indirect way. Following that, we find a close approximation with each iteration. Additionally, indirect approaches consume less memory and resources than direct approaches. That is why the majority of research is conducted using iterative approaches, as they need fewer computational resources than direct methods.

The Jacobi method and the Gauss-Seidel method are two well-known iterative approaches. [4]. Both methods are used to solve linear equations represented in matrix form. Any system of linear equations can be $AX = B$, where A is the coefficent matrix, B is a vector of competitors, and X is a vector of variables. In the Jacobi method, variables (vector X) initialised with random values. In the Jacobi method, variables (vector X) initialised with random values. After that, to solve each diagonal element in matrix A, values of variables (vector X) are plugged into the linear equation. By doing this, the new values of variables are calculated. All of these steps are repeated until convergence occurs. The

Gauss-Seidel method is similar to the Jacobi method. The only difference is that variables in the Jacobi approach are updated after each iteration, but variables in the GS method are updated immediately after each diagonal element is solved. That is why, in comparison to the Jacobi method, the GS method has a higher convergence speed. The only disadvantage of the GS method is that it is very hard to implement in parallel systems compared to the Jacobi method, because the GS method is sequential in nature.

There are many parallel algorithms for the GS method. The Red-Black GS algorithm is very popular [6]. This algorithm is based on the ordering of multiple colours. The specific matrix structure require to work this this method, and so it is dependent on the sparsity pattern of a matrix. As a result, we cannot use it with all systems of linear equations.

In the paper [3], the author has come up with a row-based parallel method. This method is derived from the GS method. To solve every diagonal element, we do the multiplication of constants and current variables. For each linear equation, this method performs all of these multiplication operations simultaneously in a parallel environment. After all these multiplication operations are completed for a full row of matrix A, variables get updated. This procedure is repeated until convergence occurs.

In paper [1], the author has come up with a new parallel method to solve linear equations by combining two other algorithms. This new method can achieve a good parallelization and can solve large systems of linear equations in very few iterations. The author has merged the Jacobi method's parallelism with the GS method's rapid convergence. The main advantage of this method is that there is no need to have any special pattern in matrix. And so this method can work with a sparse matrix as well as a dense matrix. In this new approach, linear equations are divided into blocks. Every equation within the same block is solved in a parallel environment at the same time, as the Jacobi method is very easy to implement in a parallel environment. After that, variables get updated as one would do in the GS method, and equations within the next block will use the updated values of variables. This new approach is called the PJG method. This method is being improved in this research work.

3 Algorithms

3.1 Jacobi Method (Sequential version)

The Jacobi method is a well-known iterative method for solving system of linear equations[2]. The Jacobi method is based on the following principle: take an equation and arrange it in terms of $X_{n+1} = F(X_n)$. By initializing X_n with some value and then plugging it into the $F(X_n)$ equation, a new value X_{n+1} is calculated and then it is used as the next X_n value. These operations are repeated until convergence is achieved. During this iterative process, when the value of variable X_n is set such a way that both sides of the linear equation are nearly equal, one can say that convergence has achieved.

Algorithm 1: Convergence

```
input  : matrix A = coefficient matrix
         vector X = variables
         vector B = constant vector
output: whether convergence is achived or not
$error \leftarrow 0$
for $i \leftarrow 0$ to $size$ do
    $prediction \leftarrow 0$
    for $j \leftarrow 0$ to $size$ do
        $prediction \leftarrow predction + A[i][j] \times X[j]$
    end
    $error \leftarrow error + |prediction - B[i]|$                /* absolute value */
end
if $error \leq threshold$ then
    convergence is true
else
    convergence is false
end
```

Algorithm 2: Jacobi (Sequential)

```
input  : matrix A = coefficient matrix
         vector B = constant vector
output: Solution vector X
$X \leftarrow (0, 0, ..., 0)_n$
$X_{new} \leftarrow$ array of size $X$
while until convergence do
    for $i \leftarrow 0$ to $size$ do
        $X_{new}[i] \leftarrow B[i]$
        for $j \leftarrow 0$ to $size$ do
            if $i \neq j$ then
                $X_{new}[i] \leftarrow X_{new}[i] - (A[i][j] \times X[j])$
            end
        end
        $X_{new}[i] \leftarrow X_{new}[i]/A[i][i]$
    end
    $X \leftarrow X_{new}$
end
```

Linear system is given as follow:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + ... + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + ... + a_{2n}x_n = b_2 \\ \vdots \\ a_{n1}x_1 + a_{n2}x_2 + ... + a_{nn}x_n = b_n \end{aligned}$$

Above system can be reprensented as follow:

$$AX = B$$

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}, X = \begin{pmatrix} x_1 \\ x_2 \\ \cdots \\ x_n \end{pmatrix}, B = \begin{pmatrix} b_1 \\ b_2 \\ \cdots \\ b_n \end{pmatrix}$$

are the matrix of coefficients, the solution of the system and the column-matrix with the constant terms respectively.

The variable x_1 can be solved from the first equation as follows:

$$x_1 = \frac{b_1 - a_{12}x_2 - a_{13}x_3 - ... - a_{1n}x_n}{a_{11}}$$

In general, one can write above equation for iteration k as follows:

$$x_i^{(k)} = \frac{b_i - \Sigma_{i \neq j}(a_{ij} \times x_j^{(k-1)})}{a_{ii}} \tag{1}$$

Initially vector X is initialized with some intial values. After that, for each iteration k, x_i, where $i \in \{1, 2, ..., n\}$ is solved using equation 1 with help of $X^{(k-1)}$. As shown in an alogorithm 2, these operations are repeated until convergence is obtained.

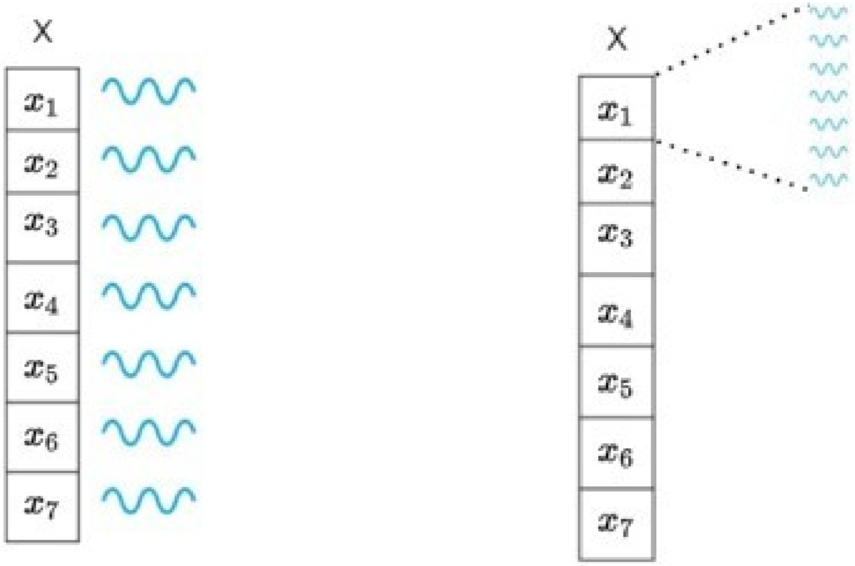
3.2 Jacobi Method (Parallel version)

Parallel computing is a technique that allows the simultaneous execution of several processes or calculations. To convert sequential problems to parallel problems, they are broken into smaller ones that can be solved concurrently. The critical point here is that all of those little problems should be independent of each other. In other words, any operations or changes done by one small problem should have no effect on the execution or outcome of another small problem. And if this is not the case, then one small problem must has to wait for another small problem to complete the execution. As a result, not all problems can be resolved concurrently in a parallel system.

For any iteration k of the Jacobi method, the calculations to solve x_i are dependent on the vector X of iteration $k - 1$. Once all equations have been solved in iteration k, then only vector X gets updated. As a result, the calculations required to solve each equation are independent of one another in this method. As a result, this method is inherently divisible into small parallel tasks. Thus, it is easy to solve linear equations in a parallel system using the Jacobi approach.

Total n parallel processes are required to solve linear equations using the Jacobi method in a parallel system, where n is the number of equations. This is because each equation's calculations can be performed concurrently. Each process i solves an i^{th} equation as shown in figure 1a. Vector X is initially initialized with random values. Total n threads are triggered in each iteration. Each thread i solves the i^{th} equation in parellel environment at the same time. Each thread solves the equation as per formula(1). Following that, we synchronise the threads (which means all threads will wait for every thread till every thread reaches this point). Thread i will then updates the value of $X[i]$ with the newly calculated value. Convergence is achieved by performing these operations iteratively as shown in algorithm 3.

While the Jacobi method is extremely simple to apply in a parallel system, it has a significant disadvantage. The Jacobi algorithm modifies the value of vector X in such a way that the difference between two sides of the equations decreases with each iteration. And after a certain number of iterations, this difference approaches zero, which is the algorithm's main goal. Due to the fact that this algorithm updates variable vector X after solving each equation in an iteration, it takes an excessive number of iterations to obtain a almost zero difference between the two sides of equations. In other words, this method requires too many iterations to attain convergence, and in worst case it becomes stuck and never achieves convergence for bigger systems[5].



(a) Parallel Jacobi Method

(b) Row Based Method

Figure 1: Parallel Threads

```
Algorithm 3: Jacobi (Parallel)
input  : matrix A = coefficient matrix
         vector B = constant vector
output: Solution vector X
$X \leftarrow (0, 0, ..., 0)_n$
while until convergence do
    do in parallel
        (Copy matrix A, vector B and vector X in shared memory)
        $i \leftarrow blockIdx.x \times blockDim.x + threadIdx.x$
        if $i < size$ then
            $prediction = B[i]$
            for $j \leftarrow 0$ to $size$ do
                if $i \neq j$ then
                    $prediction \leftarrow prediction - (A[i][j] \times X[j])$
                end
            end
            $prediction \leftarrow prediction / A[i][i]$
            (Synchronize threads)
            $X[i] = prediction$                    /* Update in global memory */
        end
    end
end
```

3.3 Gauss-Seidel Method (Sequential version)

The Gauss-Sidel method is an iterative method for solving linear equations [7]. This is also referred as the Liebmann method. The Gauss-Seidel and Jacobi methods are nearly identical. The only difference is that the Jacobian method is synchronous, whereas the GS method is asynchronous in nature. The Jacobi method calculaes any variable x_i using the values obtained in the previous iteration, but the GS method always uses the most recently updated values during the iterative process. In other words, this method updates vector X immediately after solving any equation i (as given in formula(2)), but the Jacobian method updates vector X only after each iteration. Thus, the GS method begins the approximation process far earlier than the Jacobi method. As a result, this method, even for enormous systems, can achieve convergence in very few iterations compared to the Jacobi method. Thus, this method overcame the Jacobi method's slow convergence issue. The variable vector X is initialized with some initial values in this algorithm. Then, as demonstrated in the algorithm 4, equation i is solved as per formula (2) and $X[i]$ is updated immediately. Here, variable x_i is dependent on variables $x_1, x_2, ...x_{i-1}$. These processes are carried out until convergence occurs.

$$x_i = \frac{b_i - \Sigma_{i \neq j}(a_{ij} \times x_j)}{a_{ii}} \tag{2}$$

Algorithm 4: Gauss Seidel (Sequential)

```
input  : matrix A = coefficient matrix
         vector B = constant vector
output: Solution vector X
$X \leftarrow (0, 0, ..., 0)_n$
while until convergence do
    for $i \leftarrow 0$ to size do
        $X[i] \leftarrow B[i]$
        for $j \leftarrow 0$ to size do
            if $i \neq j$ then
                $X[i] \leftarrow X[[i] - (A[i][j] \times X[j])$
            end
        end
        $X[i] \leftarrow X[i]/A[i][i]$        /* Immediate update variable vector X */
    end
end
```

3.4 Row-Based Method (Parallel version)

As discussed above, the GS algorithm can achieve convergence in very few iterations, as it uses the most recently updated values. Thus, the calculation of variable x_i is dependent on $x_1, x_2, ..., x_{i-1}$. In other words, x_i cannot be calculated without solving x_{i-1}. As a result, the GS algorithm cannot calculate all variables $x_1, x_2, ..., x_{i-1}$ concurrently in a parallel system. The author proposed a row-based parallel method in the paper [3]. In the formula (2), the author calculated $\Sigma_{i \neq j} a_{ij} x_j^{(k-1)}$ this part in parallel system. There are $n-1$ multiplications and $n-1$ summations in expression (3). In a parallel system, n threads are required to solve expression (3). Here n denotes the number of equations. As seen in the figure 1b, n threads are launched to solve the i^{th} equation. Each thread j multiplies $A[i][j]$ and $X[j]$ and saves the result in the multiplications vector. Then, as shown in the algorithm 5, there will be a summation of multiplication vectors in the parallel system. And the value of $X[i]$ is updated based on the sum of the multiplication vector. These processes are carried out until convergence occurs.

$$\Sigma_{i \neq j} a_{ij} x_j^{(k-1)} \tag{3}$$

3.5 Parallel Jacobi Gauss-Seidel Method

So far, two iterative techniques have been discussed: the Jacobi and the Gauss-Seidel. While the Jacobi method can achieve a high degree of parallelization, it requires too many iterations to achieve convergence, and in the worst-case method, it becomes stuck and never achieves convergence for big systems. The GS method requires a small number of iterations to reach convergence, even for huge systems, but it is extremely difficult to apply in a parallel system because any variable i is dependent on variable $i-1$. In other words , if good parallelization is desired, the time required for the algorithm to reach convergence must be compromised, and vice versa. In other words, the Jacobi and GS methods cannot accomplish both high parallelization and fast convergence.

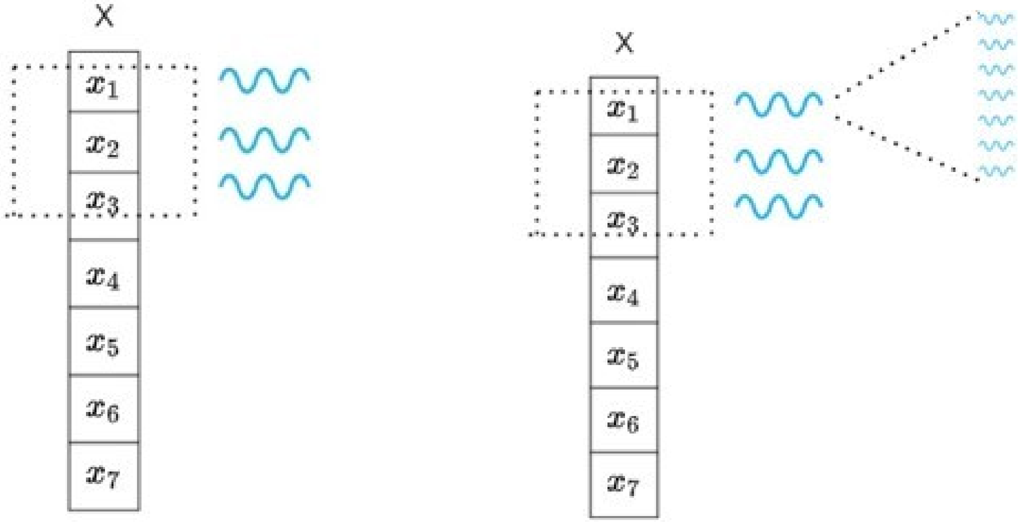
Algorithm 5: Row Based Parallel Method

```
input  : matrix A = coefficient matrix
         vector B = constant vector
output: Solution vector X
$X \leftarrow (0, 0, ..., 0)_n$
$multiplications \leftarrow$ array of size X                /* GPU Global Memory */
while until convergence do
    for $i \leftarrow 0$ to size do
        do in parallel
            (Copy matrix A and vector X in shared memory)
            $j \leftarrow blockIdx.x \times blockDim.x + threadIdx.x$
            $multiplications[j] = A[i][j] \times X[j]$
        end
        do in parallel
            Do sum of array multiplications
        end
        do in parallel
            (Only one CUDA thread)
            $j \leftarrow blockIdx.x \times blockDim.x + threadIdx.x$
            if $j \neq 0$ then
                $X[i] \leftarrow (B[i] - multiplications[0])/A[i][i]$
            end
        end
    end
end
```

Algorithm 6: PJG Method

```
input  : matrix A = coefficient matrix
         vector B = constant vector
         P = block size
output: Solution vector X
$X \leftarrow (0, 0, ..., 0)_n$
while until convergence do
    for $blockIndex \leftarrow 0$ to $ceil(size/p)$ do
        do in parallel
            (Copy matrix A, vector B and vector X in shared memory)
            $threadId \leftarrow blockIdx.x \times blockDim.x + threadIdx.x$
            if $threadId < P$ then
                $prediction \leftarrow 0$
                $i \leftarrow blockIndex \times P + threadId$
                if $i < size$ then
                    $prediction \leftarrow B[i]$
                    for $j \leftarrow 0$ to $size$ do
                        if $i \neq j$ then
                            $prediction \leftarrow prediction - (A[i][j] \times X[j])$
                        end
                    end
                    $prediction \leftarrow prediction/A[i][i]$
                end
            end
            (Synchronize threads)
            $X[i] \leftarrow prediction$                /* Update in global memory */
        end
        $blockIndex \leftarrow blockIndex + 1$
    end
end
```

To address this issue, the author of paper [1] developed a new method called the PJG (Parallel Jacobi Gauss-Seidel) method. The PJG method combines the Jacobi and GS methods. The author of the paper [1] combined the good characteristics of both methods to develop the PJG method. The good feature of the Jacobi method is that it updates variable vector X after every iteration, which is why the Jacobi method is very easy to implement in a parallel system. The advantage of the GS method is that after solving any variable xi, it immediately updates variable vector X and uses the most recently updated values of variable vector X, which explains why convergence occurs very quickly. The PJG method divides equations into blocks of size p. All equations within the same block are solved simultaneously in a parallel system (a Jacobi method feature), and once one batch is solved, variable vector X is updated (a feature of the GS method). The remaining batches' equations will now use the updated vector X values. Due to the fact that all equations in the same batch size are independent of one another, they are solved parallelly. After solving all equations in a batch, the variable vector X is updated. Thus, in comparison to the Jacobi method, there will now be more frequent updates to variable vector X. As a result, the PJG algorithm converges in a very few iterations. Therefore, the PJG algorithm achieves both high parallelism and convergence.



(a) Parallel Jacobi Gauss-Seidel Method

(b) PJG with Dynamic Parallelization

Figure 2: Parallel Threads for PJG method

To solve linear equations in a parallel system using the PJG method, p threads are required. Initially variable vector X initialized with some values. As shown in the figure 2a, each thread i will solve equation i in the batch. Once all equations have been solved in a batch, the variable vector X will be updated in the as shown in algorithm 6. Then the next p equations will be solved, and so on until all of the equations are solved. All of these processes will be repeated until the algorithm reaches convergence.

The primary objective of this research is to archive more parallelization in the PJG method. I hypothesised that by integrating the dynamic parallelization concept with the PJG method, the performance of the PJG method can be improved. To test this hypothesis, I have made appropriate changes in the PJG algorithm by applying the dynamic parallelization concept to it and then implementing it in a parallel environment. I have compared the performance of the PJG method with dynamic parallelization, with all the above algorithms discussed so far.

4 Proposed Method

4.1 Dynamic Parallelization

In CUDA, a kernel is a function that runs on the GPU. Generally, the kernel is invoked by host code, but in some situations, more parallelization may be achieved by calling a kernel from another kernel. This is called ”dynamic parallelization”. A set of threads is called a ”block” in CUDA. A grid is a set of blocks in the CUDA programming architecture. A kernel operates on a grid. A parent grid launches kernels known as child grids in CUDA Dynamic Parallelism. Certain properties and restrictions, such as the L1 cache/shared memory configuration and stack size, are inherited by a child grid from the parent grid. The most important thing to remember about dynamic parallelization is that any thread that meets a kernel launch will execute it. So, if the parent grid contains 128 blocks, each with 64 threads, there will be 8192 kernel launches. It can degrade the performance. It is critical to control the number of child kernels that are invoked. Grid launches are completely nested in dynamic parallelization. To put it another way, child grids are always completed before parent grids. There is no need for explicit synchronization. Exexution of dynamic parallelization is shown in figure 3. If the parent kernel requires the results computed by the child kernel in order to conduct its own work, the parent kernel must explicitly synchronize in order to ensure that the child grid has finished execution before proceeding with the work.

A parent grid frequently relies on a child grid to read and write to global memory. To accomplish this, the CUDA Device Runtime ensures that the parent and child grids have a fully consistent view of global memory when the child starts and stops. In other words, if a parent writes in memory and then launches a child grid, CUDA will make sure that the child grid will see the value written by the parent grid and vice versa. This also means that if more than one child grid is run consecutively, any writes made by earlier child grids are seen by later child grids, even if no synchronization has occured between them. Memory consistancy is shown in figure‘4.

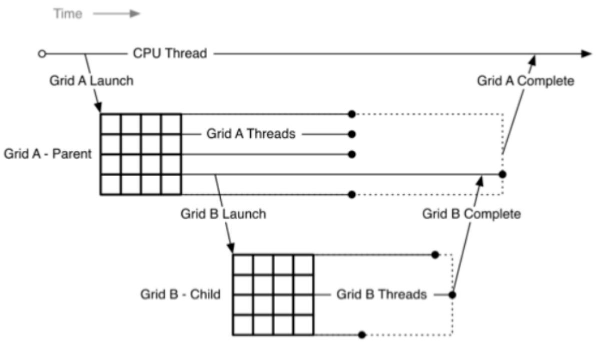


Figure 3: Execution of dynamic parallelization

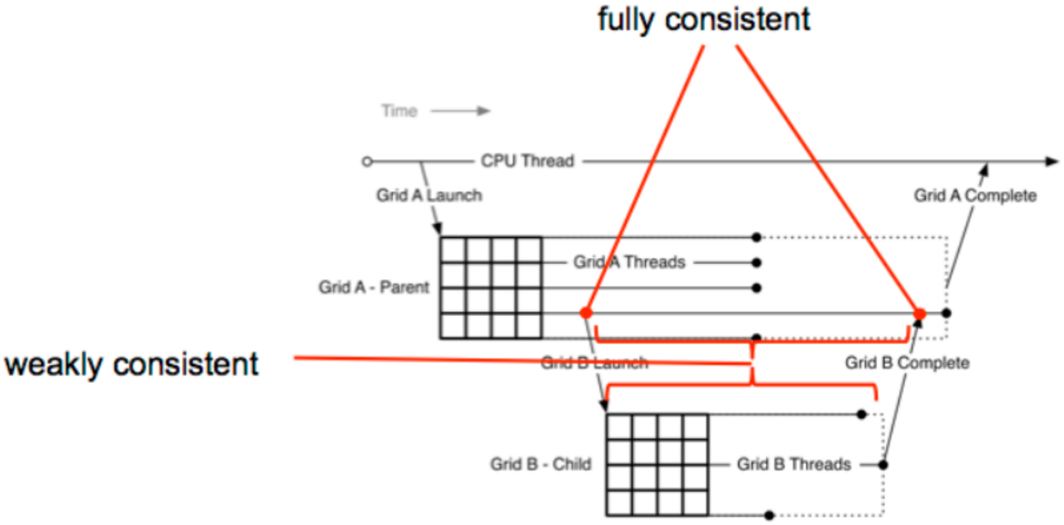


Figure 4: Memory consistency

During the experiment, it is observed that performance of the JPG algorithm degrades rather than improves. This is because kernel launching, whether parent or child, is a difficult process [nvidia documentation]. If child kernels do not extract much parallelism, then the child kernel launch overheads may cancel out any benefit. To overcome this, I implemented dynamic parallelism without the use of a nested kernel.

4.2 The PJG algorithm with dynamic parallelization

As we have discussed above, dynamic parallelization is a nested kernel execution. As shown in the figure 2b, thread i solves the i^{th} equation in block. In other words, thread i calculates the expression 3 in formula 2 for linear equation i . All threads perform these calculations simultaneously in order to solve all of the equations contained within the same block. For block of size p, the PJG algorithm requires p threads. The initial idea was to invoke n child threads from every p parent thread, as shown in the figure 2b. All these child threads of parent thread i, calculate expression 3 in formula 2 for equation i, in a parallel environment. However, as noted previously, executing a child kernel is a difficult and time consuming process. As a result, I attempted to simulate dynamic parallelization without using a nested kernel to avoid this limitation.

Instead of executing p parent threads first and then n child threads from every p parent thread, one can execute $p \times n$ threads from host code only. Then every thread is assigned the equation id as shown in the algorithm 7. The equation id i indicates that the thread is solving i^{th} equation in the block. So, for the first n threads, the equation id will be 0, then for the next n threads, it will be 1 and so on. For each thread, i and j variables are assigned as shown in the algorithm 7. An variable i indicates that thread is solving equation i and variable j indicates that thread multiple $A[i][j]$ and $X[i]$. All these multiplications are stored in a matrix named "multiplications". After that, every thread will do a summation of each row of multiplications matrix simultaneously, and then variable vector X will be updated. These operations are performed until the algorithm achieves convergence. Such that dynamic parallization is simulated without using nested kernel execution.

Algorithm 7: PJG Method (Dynamic Parallelization)

```
input  : matrix A = coefficient matrix
         vector B = constant vector
         P = block size
output: Solution vector X
$X \leftarrow (0, 0, ..., 0)_n$
$multiplications \leftarrow$ matrix of dimensions P X size    /* GPU Global Memory
 */
while $until\ convergence$ do
    for $blockIndex \leftarrow 0$ to $ceil(size/p)$ do
        do in parallel
            (Copy matrix A, vector B and vector X in shared memory)
             $threadId \leftarrow blockIdx.x \times blockDim.x + threadIdx.x$
            $equationId \leftarrow floor(threadId/size)$
            if $equationId < P$ then
                $i \leftarrow blockIndex \times P + equationId$
                $j \leftarrow threadId \pmod{size}$
                if $i == j$ then
                    $multiplications[equationId][j] \leftarrow 0$
                else
                    $multiplications[equationId][j] \leftarrow A[i][j] \times X[j]$
                end
            end
        end
        do in parallel
            (Parallel summation of elements of each row in multiplications matrix)
        end
        do in parallel
            (P threads)
            (Copy matrix A, vector B and matrix multiplications in
             shared memory)
            $threadId \leftarrow blockIdx.x \times blockDim.x + threadIdx.x$
            $equationId \leftarrow floor(threadId/size)$
            if $threadId < P$ then
                $i \leftarrow blockIndex \times P + threadId$
                $X[i] \leftarrow \frac{B[i] - sum(multiplications[threadId])}{A[i][i]}$
            end
        end
        $blockIndex \leftarrow blockIndex + 1$
    end
end
```

5 Experiment

5.1 Hardware and Software Configuration

The computer hardware used in this experiment is equipped with used an NVIDEA T4 GPU. The GPU features 320 tensor cores and 2560 CUDA cores. This GPU has 16 GB of GDDR6 global memory. Cuda is used to implement algorithms in GPU. Cuda is an extension in C++.

5.2 Simulation Settings

To provide a comprehensive performance report, simulation settings cover two types of systems with liner equations: large systems and small systems. A total of seven algorithms were used in this experiment. Those algorithms are the Jacobi Algorithm, the Jacobi algorithm with a parallel version of the Jacobi algorithm, the GS algorithm, the Row Based method, the sequantial version of the PJG method, the parallel version of the PJG method, and the proposed method (the PJG method with dynamic parallelization). All algorithms are tested on the same inputs.

The first test is conducted with a small system of linear equations. A total of 50 equations are used. The block size is 10 for the PJG method. The second test is conducted with a large system of linear equations. A total of 1000 equations are used. The block size is 35 for the PJG method. The third test is also conducted with 1000 linear equations and with block size 35. But in this particular test, all algorithms are forced to run for 1000 iterations instead of until convergence is achieved.

5.3 Results

Method	Time (Seconds)	Iterations
Jacobi	0.0125762	478
Jacobi (Parallel)	0.0268081	478
Gauss Seidel	0.0001888	7
Row Based Parallel Method	0.00842755	7
PJG (Sequential)	0.000267712	9
PJG (Parallel)	0.00077216	9
PJG (Dynamic Parallelization)	0.00100070906	9

Table 1:
Iteraions : until convergence
Number of equations : 50
Block size : 10

Table 1 summarizes the results of the first test (small system of linear equations). The Jacobi method required 478 iterations to solve 50 linear equations, whereas the GS and PJG methods required just 7 and 9 iterations, respectively. A critical point to note here is

Method	Time (Seconds)	Iterations
Jacobi	99.802	1000 (no convergence)
Jacobi (Parallel)	5.69737	1000 (no convergence)
Gauss Seidel	0.0992823	10
Row Based Parallel Method	0.0930212	10
PJG (Sequential)	0.267712	10
PJG (Parallel)	0.056479	10
PJG (Dynamic Parallelization)	0.039485	10

Table 2:
Iteraions : until convergence
Number of equations : 1000
Block size : 35

Method	Time (Seconds)
Jacobi	5.562
Jacobi (Parallel)	0.270
Gauss Seidel	5.518
Row Based Parallel Method	3.389
PJG (Sequential)	5.549
PJG (Parallel)	0.770
PJG (Dynamic Parallelization)	0.587

Table 3:
Iteraions : 1000
Number of equations : 1000
Block size : 35

that all parallel algorithms take longer than their sequential counterparts for small system. This is because the degree of parallelization highly depends on the number of equations. If the number of equations is small, kernel launch and data operations cancel out any benefit gained by parallel operations.

Results for the second test (a large system of linear equations) are shown in Table 2. The number of equations is 1000, and the block size for the PJG is 35. The Jacobi method (parallel version) achieved a 17.5x speedup compared to its sequential version. But even after 1000 iterations, it does not solve equations. All other methods take only 10 iterations to solve 1000 linear equations. The GS algorithm took almost the same time in the parallel and sequential versions. No significant speedup was observed in this test for the GS algorithm. The proposed algorithm (the PJG with dynamic parallelization) takes the least amount of time of all the other algorithms. The PJG algorithm (parallel version) and the proposed algorithm achieved 5.74x and 6.78x speedup compared to the PJG (sequential version). When compared to the GS method, the proposed method achieved a 2.5x speedup, while the PJG method (parallel version) achieved a 1.75x speedup. Here, there is no significant performance enhancement observed, this is because the GS algorithm and the PJG algorithm solved 1000 equations in just 10 iterations.

Results for the test (large system but all algorithms are forced to run for 1000 iterations) are shown in table 3. The number of equations is 1000 and the block size is 35 for the PJG method. The Jacobi algorithm achieved a 20.6x speedup, but in this test also, the algorithm does not solve the equations after 1000 iterations. The proposed algorithm took the least time (the Jacobi method does not solve equations, so its results are not considered). The PJG method (parallel version) and the proposed method achieved 7.2x and 9.45x speedup, respectively, when compared with the sequential version of the PJG method. When compared to the sequential version of the GS method, the parallel version of the PJG method achieved a 7.1x speedup, while the proposed method achieved a 9.4x speedup.

5.4 Conclusion

In this research work, I have tried to improve the performance of the PJG method by applying dynamic parallelization. The proposed method achieved a 9.45x speedup compared to the sequential PJG method, which is higher than the 7.2x speedup achieved by the parallel PJG method. The proposed method achieved a 9.4x speedup compared to the GS method. Additionally, I have compared the performance of various algorithms like the Jacobi algorithm, the GS algorithm, and the PJG algorithm by executing them on the GPU. The Jacobi method achieved the highest speedup, which is 20.6x, but it didn't achieve convergence. The GS method achieved good convergence, but it did not achieve significant speedup. The PJG method achieved good parallelization and convergence as well. Further work will focus on developing an algorithm for the proposed method to implement it in a distributed memory architecture.

References

[1] Afshin Ahmadi, Felice Manganiello, Amin Khademi, and Melissa C. Smith. A parallel jacobi-embedded gauss-seidel method. *IEEE Transactions on Parallel and Distributed Systems*, 32(6):1452–1464, 2021.

[2] I.N. Bronshtein and K.A. Semendyayev. *Handbook of Mathematics*, page 892. Springer Berlin Heidelberg, 2013.

[3] Hadrien Courtecuisse and Jérémie Allard. Parallel dense gauss-seidel algorithm on many-core processors. In *2009 11th IEEE International Conference on High Performance Computing and Communications*, pages 139–147, 2009.

[4] L.A. Hageman. *Applied Iterative Methods.* Elsevier Science, 2016.

[5] Louis A. Hageman and David M. Young. *Applied Iterative Methods*, page 140. Dover Publications, 2004.

[6] Kawai, Masatoshi, Iwashita, Takeshi, Nakashima, Hiroshi, and Osni Marques. Parallel smoother based on block red-black ordering for multigrid poisson solver. *Lecture Notes in Computer Science High Performance Computing for Computational Science - VECPAR 2012*, page 292–299, 2013.

[7] Hongxia Liu and Tianxiang Feng. Study on the convergence of solving linear equations by gauss-seidel and jacobi method. In *2015 11th International Conference on Computational Intelligence and Security (CIS)*, pages 100–103, 2015.

[8] A. Quarteroni, R. Sacco, , and F. Saleri. *Numerical Mathematics*, volume 37. Springer, 2010.