CS5300 Project Report
Parallelizing Gauss-Siedel method

Aaryan, CO21BTECH11001
Aayush, CO21BTECH11002
Darpan, CO21BTECH11004

1 Introduction

In the context of a two-dimensional space (2D), the heat transfer problem involves analyzing the temporal evolution of surface temperatures at various points until a stable condition is achieved, where the temperatures become constant (referred to as equilibrium temperature). Initially set at specific temperatures, each point undergoes changes influenced by heat sources positioned along the boundaries of the plane, leading to the dispersion of heat across the surface.

1.1 Problem Modelling

The heat transfer problem, as mentioned earlier, involves the diffusion of heat across a surface until it reaches a stable state (equilibrium). In a two-dimensional plane with coordinates (x, y), the problem can be represented by the heat equation:

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}$$

This equation describes the temporal variation of temperature $u(x, y)$ at a specific point. To determine the temperature at each point when the steady state is reached (i.e., when heat transfer ceases, $\frac{\partial u}{\partial t} = 0$), the problem can be expressed as:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 0$$

It's important to note that the steady-state equation doesn't imply uniform temperature across the entire plane. Temperature gradients can exist in the plate, but these gradients remain constant after reaching steady state as long as the heat sources are constant. To ascertain the temperature at any point in the plate, the presented equation needs to be applied to each point. A common approach is to use a system of linear equations, where each line represents the steady-state temperature at a specific point. Using the previous equation, the temperature at a given point when steady state is reached can be determined with the following expression:

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \Rightarrow \frac{u(x+h/2, y) + u(x-h/2, y) - 2u(x, y)}{h^2} + \frac{u(x, y+k/2) + u(x, y-k/2) - 2u(x, y)}{k^2}$$

To calculate the temperature difference between two points, different steps h and k can be applied for the coordinates x and y, introducing errors of magnitude $O(h^2)$ and $O(k^2)$ respectively.

1.2 Approach

The adoption of iterative methods for solving computational problems is driven by the necessity to address large-scale problems, particularly those involving systems of linear equations with a substantial number of variables. In such cases, employing a direct method, while yielding precise solutions, demands significant computational resources and time for execution. Consequently, iterative methods become more favorable for handling these scenarios. An illustrative example is the Gauss-Seidel iterative method, which continuously updates a matrix containing the coefficients of each equation.

The update for each element, denoted by x_i^{k+1} in the k-th iteration, follows the expression:

$$x_i^{k+1} = \frac{b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{k+1} - \sum_{j=i+1}^{n} a_{ij} x_j^k}{a_{ii}}$$

Here, k represents the current iteration. The method terminates when the differences between consecutive iterations fall below a predefined threshold or tolerance.

Given that the coefficients of numerous terms in the resulting linear equations are zero, the task simplifies to computing the mean of four neighboring points. These coefficients of the neighbouring points are denoted as $aE[i], aW[i], aN[i]$, and $aS[i]$.

Hence, our final equation becomes

$$x_{i,j}^{k+1} = \frac{b[i][j] + aE_{ij} x_{i+1,j}^k + aW_{ij} x_{i-1,j}^{k+1} + aN_{ij} x_{i,j+1}^k + aE_{ij} x_{i,j-1}^{k+1}}{aP_{i,j}}$$

This method enhances the Jacobi method by utilizing previously computed element values to update the current element, resulting in a faster convergence rate and eliminating the need to store two vectors. However, the reliance on the most recent values introduces data dependencies in each iteration, posing challenges for parallel development.

1.3 Discretization

Determining the temperature at every point across a continuous 2D plane is impractical due to the infinite number of points involved. Consequently, it becomes essential to transform the domain from continuous to discrete, a procedure known as discretization. This involves applying a mesh to the continuous domain and evaluating the function (under study) at each mesh point. In the context of the heat equation, the mesh comprises four boundaries referred to as heat sources, each with known temperature values, and a set of internal points with unknown temperatures. The discretization process introduces a truncation error because fewer points than those present in the initial problem are used to characterize the 2D plane. The magnitude of the truncation error is inversely proportional to the number of points in the mesh.

1.4 Exact Solution

We assume that homogeneous Dirichlet conditions are specified at the left, right and top boundaries, i.e. $T(x = 0) = T(x = Lx) = T(y = Ly) = 0$, and homogeneous Neumann condition is specified at the bottom, i.e. $(\partial T/\partial y)(y = 0) = 0$. The exact solution for the given problem and boundary conditions is expressed as

$$T(x, y) = x(1 - x)cos(\pi y)$$

This analytical solution provides a precise description of the temperature distribution across the two-dimensional domain. In this formula, x and y represent the spatial coordinates within the domain.

2 Sequential Algorithm

A pseudocode for sequential solver is given in Listing 1

Listing 1: Pseudocode for sequential implementation of Gauss-Siedel method

```
// T -> (nx+2, ny+2) matrix
// aW, aS, aP, aN, aE, b -> each of size (nx, ny)

aW, aS, aP, aN, aE, b = get_coefficients(nx, ny, xst, xen, yst, yen);

void sequentialGS() {
    for iter in [1, max_iter] {
        err = 0.0;
        for i in [0, nx) {
            for j in [0, ny) {
                ip = i + 1, jp = j+1;

                T_prev = T[ip][jp];

                T[ip][jp] = ( b[i][j] + aE[i][j]*Tpnew[ip+1][jp] + aW[i][j]*
                    Tpnew[ip-1][jp] +
                          aN[i][j]*Tpnew[ip][jp+1] + aS[i][j]*Tpnew[ip][jp-1] )
                              / aP[i][j];

                err += (T[ip][j] - T_prev)^2;
            }
        }

        if (err < threshold) break;
    }
}
```

3 Parallel Algorithms using Barrier

The general structure for implementing multithreaded algorithms using barriers is given in Listing 2.

Listing 2: Pseudocode for parallel implementation of Gauss-Siedel method

```
err = MAX_VALUE; // global variable
thread_err[num_threads]; // global array

void updateVals() {
    temp_err = 0.0; // Thread local variable
    for iter in [1, max_iter] {
        // Update T and calculate temp_err

        thread_err[ThreadId.get()] = temp_err;

        Barrier_1.wait();
        // Now calculate error thread will run

        Barrier_2.wait();

        if (err < threshold) break;
    }
}

void calcError() {
    for iter in [1, max_iter] {
        // Wait for updating threads to complete the iteration
        Barrier_1.wait();

        // Calculate error and update the variable err

        Barrier_2.wait();

        if (err < threshold) break;
    }
}

void parallelGS(n) {
    // Create n updateVals threads

    // Create 1 calcError thread
}
```

3.1 Wave Method

A grid of size (nx, ny) has $nx + ny - 1$ number of diagonals. Each diagonal is allocated to one thread. Precisely, if the number of threads are n, then a thread with id i is allocated the diagonals $\{i, i+n, i+2n, ...\}$.

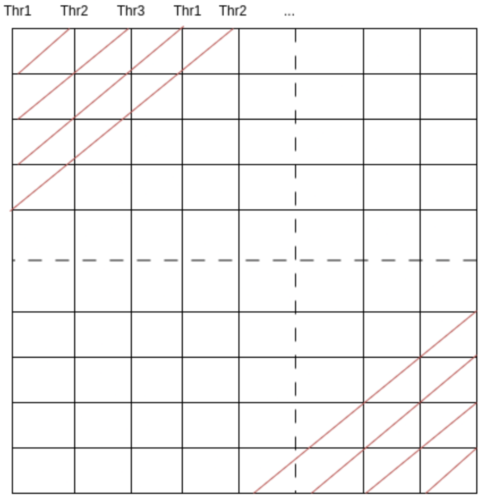


Figure 1: Representation of the distribution of diagonals to 3 threads

A pseudocode for wave method is given in Listing 3.

Listing 3: Pseudocode for updating values using Wave Method

```
thrId = ThreadId.get();

for (iter = 0; iter < max_iter; iter++) {
    diagonal_number = thrId;

    while (diagonal_number <= num_diagonals) {
        // Update every point in this diagonal

        diagonal_number += num_threads;
    }
}
```

3.2 Red Black Method

We split the grid points into red and black color scheme (shown in Figure 2).

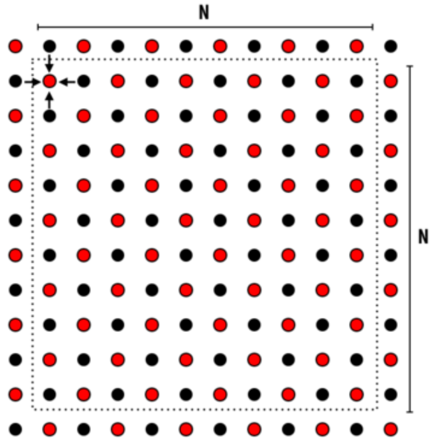


Figure 2: Splitting of grid points into red and black points

Notice that the calculation for a red point is dependent only on black points and vice versa. Each iteration of this algorithm can be divided into these steps:

- Distribute all the red points among the threads. Since the calculation of red points are dependent on black points, which are not calculated yet, the values of the previous iteration will be used for them.
- Wait for all the threads to complete the first step. A barrier is used to achieve this.
- Distribute all the black points among the threads. Since the calculation of black points are dependent on red points, which are calculated in the first step, the values of the same iteration will be used for them.

The distribution of points at each step is done by statically allocating rows to the threads. A pseudocode for red-black method is given in Listing 4.

Listing 4: Pseudocode for updating values using Red-Black Method

```
thrId = ThreadId.get();

for (iter = 0; iter < max_iter; iter++) {
    for (row = thrId; row <= nx; row += num_threads) {
        for (col = 1; col <= ny; col++) {
            if (row+col) is even {
                // Update T[row][col]
            }
        }
    }

    // Wait for all threads to complete step 1.
    Barrier.wait();

    for (row = thrId; row <= nx; row += num_threads) {
        for (col = 1; col <= ny; col++) {
            if (row+col) is odd {
                // Update T[row][col]
            }
        }
    }
}
```

3.3 Static Row Allocation

A grid of size (nx, ny) has nx rows. Distribution of rows among threads follows a specific pattern based on thread id and number of threads available. Thread i is allocated with rows $\{i, i+k, i+2k, ...\}$, where k is number of threads.

A pseudocode for wave method is given in Listing 5.

Listing 5: Pseudocode for updating values using Static Row Allocation

```
thrId = ThreadId.get();

for (iter = 0; iter < max_iter; iter++) {
    for (i = thrId; i < nx ; i += num_threads) {
        for (j = 0; j < ny ; j++) {
            // update T[i+1][j+1]
        }
    }
    // wait for all threads to complete
    barrier.wait();
}
```

3.4 Static Chunk Allocation

A grid of size (nx, ny) has nx rows. Distribution of rows among threads follows a specific pattern based on thread id, nx and number of threads available. Thread i is allocated with rows $\{i * c, i * c + 1, ..., (i+1) * c - 1, (i+k) * c, (i+k) * c + 1, ..., (i+k+1) * c - 1, ...\}$, where k is number of threads and c is chunk size.

A pseudocode for wave method is given in Listing 6.

Listing 6: Pseudocode for updating values using Static Chunk Allocation

```
thrId = ThreadId.get();

for (iter = 0; iter < max_iter; iter++) {
    id = thrId;
    while (true) {
        for (i = id*chunkSize; i < nx && i < (id+1)*chunkSize ; i += num_threads
            ) {
            for (j = 0; j < ny ; j++) {
                // update T[i+1][j+1]
            }
        }
        if (i>=nx) break;
        id += num_threads;
    }
    // wait for all threads to complete
    barrier.wait();
}
```

3.5 Dynamic Row Allocation

An atomic counter *row* is initialized as 1. Each thread gets the current row id and increments *row* by 1. If a thread obtains a row id $> nx$, it breaks out of the loop. After each iteration, row is set to 1 by the error calculation thread.

The pseudocode for a thread which updates the values is given in Listing 7.

Listing 7: Pseudocode for iteration in batches

```
void dynamicRow() {
    // Thread local variables

    for(iter = 0; iter < max_iter; iter++) {
        while(true) {
            // get row no. to operate on
            currRow = row.fetchAdd(1);

            // iteration has completed
            if(currRow > nx) break;

            for(col = 1; col <= ny; col++) {
                // update T[currRow][col]
            }
        }

        // wait for all threads to complete current iteration
        barrier.wait();
    }
}
```

3.6 Dynamic Chunk Allocation

An atomic counter *row* is initialized as 1. Each thread gets the starting row id and increments *row* by *chunkSize*. The thread then updates the rows $\{startRow, startRow{+}1, ..., startRow{+}chunkSize{-}1\}$. If a thread obtains a row id $> nx$, it breaks out of the loop. After each iteration, row is set to 1 by the error calculation thread. The optimal size of chunk can depend on cache size of the machine.

The pseudocode for a thread which updates the values is given in Listing 8.

Listing 8: Pseudocode for iteration in batches

```
void dynamicRow() {
    // Thread local variables

    for(iter = 0; iter < max_iter; iter++) {
        while(true) {
            // get row no. to operate on
            startRow = row.fetchAdd(chunkSize);

            // iteration has completed
            if(currRow > nx) break;

            for(currRow = startRow; currRow < min(startRow+chunkSize, nx+1);
                currRow++) {
                for(col = 1; col <= ny; col++) {
                    // update T[currRow][col]
                }
            }
        }

        // wait for all threads to complete current iteration
        barrier.wait();
    }
}
```

3.7 Iteration in Batches

Recognizing the inherent inefficiency introduced by barriers, we devised a modified implementation wherein threads execute iterations in batches rather than synchronizing at every iteration. This method has proven effective in mitigating synchronization overhead and, consequently, reducing the overall computational time.

In particular, the approach involves grouping iterations into batches, allowing threads to operate autonomously within each batch. By adopting this strategy, threads are relieved of the burden of constant synchronization, which becomes especially beneficial when dealing with a large number of iterations. Consequently, the computational workload is distributed more efficiently, resulting in an appreciable improvement in the overall speedup.

This optimization becomes particularly noteworthy when considering the substantial scale of computational iterations—often numbering in the order of tens of thousands. Therefore, it makes perfect sense to perform a few hundred iterations in batches.
The pseudocode for a thread which updates the values is given in Listing 9.

Listing 9: Pseudocode for iteration in batches

```
void batchUpdateVals() {
    // Thread local variables
    batch_iter_count = 0;
    temp_err = 0.0;

    for (iter = 0; iter < max_iter; iter++) {
        // Update T
        // Calculate error only when (batch_iter_count = iteration_batch_size)

        if (batch_iter_count == iteration_batch_size) {
            thread_err[ThreadId.get()] = temp_err;

            Barrier_1.wait();
            // Now calculate error thread will run

            Barrier_2.wait();

            if (err < threshold) break;
        }

        if (batch_iter_count == iteration_batch_size) batch_iter_count = 0;
        else batch_iter_count++;
    }
}
```

4 Barrier Free Algorithm

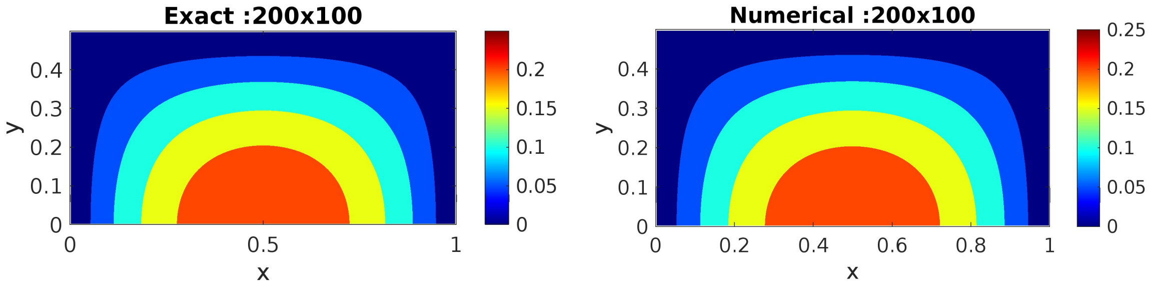
We devised an alternative parallel algorithm for the Gauss-Seidel method that eliminates the need for barriers, simplifying thread coordination. In this approach, each thread computes the error during every iteration and records it in a 2D array (size: `(max_iter, num_threads)`). Simultaneously, each thread fetches and increments an atomic integer related to the iteration within a designated array, `count_arr`. When the fetched value equals `num_threads - 1`, the thread calculates the error. If the computed error falls below a specified threshold, the thread signals other threads to terminate through a shared memory boolean variable. This strategy streamlines the parallel computation process without the reliance on barriers, contributing to improved efficiency.
The pseudocode for this approach is given in Listing 10

Listing 10: Pseudocode for barrier free algorithm

```
// global arrays
count_arr[max_iter] = {0};
thread_err[max_iter][num_threads];

done_calculation = false; // global variable

void barrierFreeUpdateVals() {
    for (iter = 0; iter < max_iter; iter++) {
        if (done_calculation) break;

        // Update T and calculate temp_err
        // Keep checking the value of done_calculation

        if (done_calculation) break;
        thread_err[iter][ThreadId.get()] = temp_err;

        get_count_value = count_arr[iter].fetchAndAdd(1);

        if (get_count_value == num_threads - 1) {
            // Calculate error

            if (err < threshold) {
                done_calculation = true;
                break;
            }
        }
    }
}
```

Note: Link for Github repository.

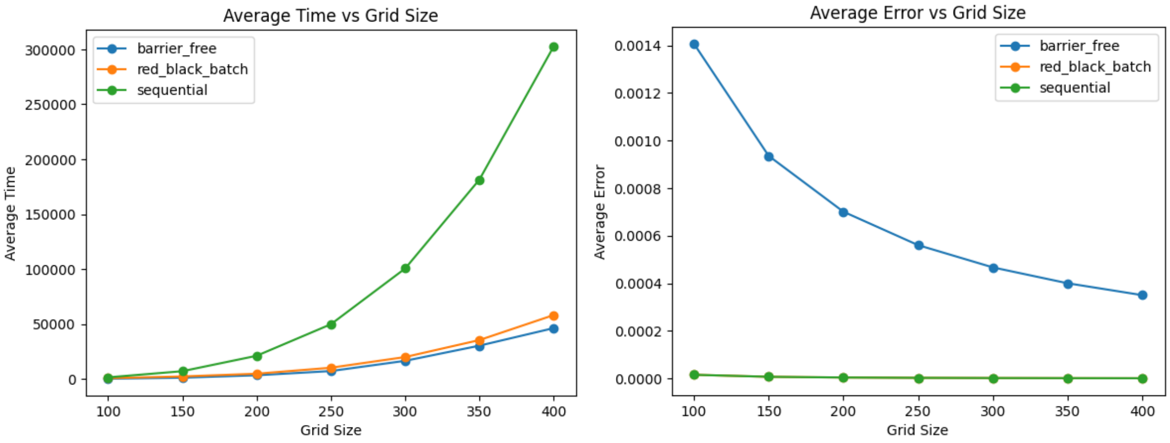
5 Results



(a) Exact Solution

(b) Solution using BarrierWithBatch method

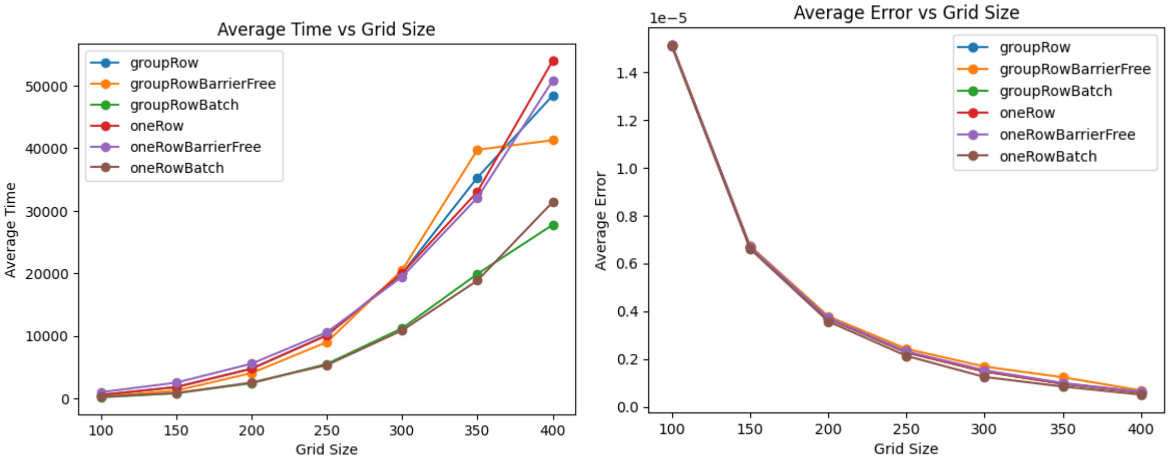
Figure 3: Temperature contours for exact and numerical solution for a grid size of $(200, 100)$



(a) Time vs Grid Size

(b) Error vs Grid Size

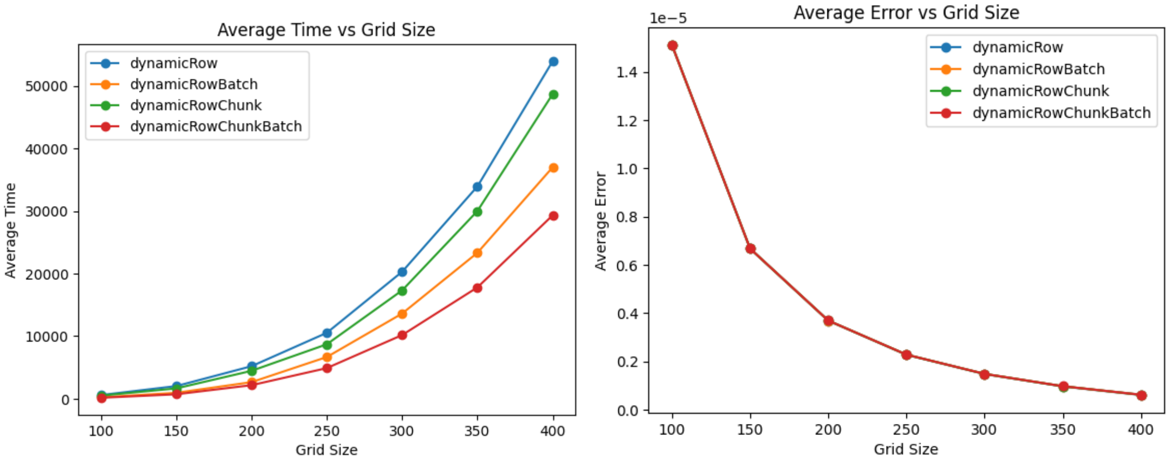
Figure 4: Sequential vs Red-Black-Batch vs Barrier-Free (# threads = 16)



(a) Time vs Grid Size

(b) Error vs Grid Size

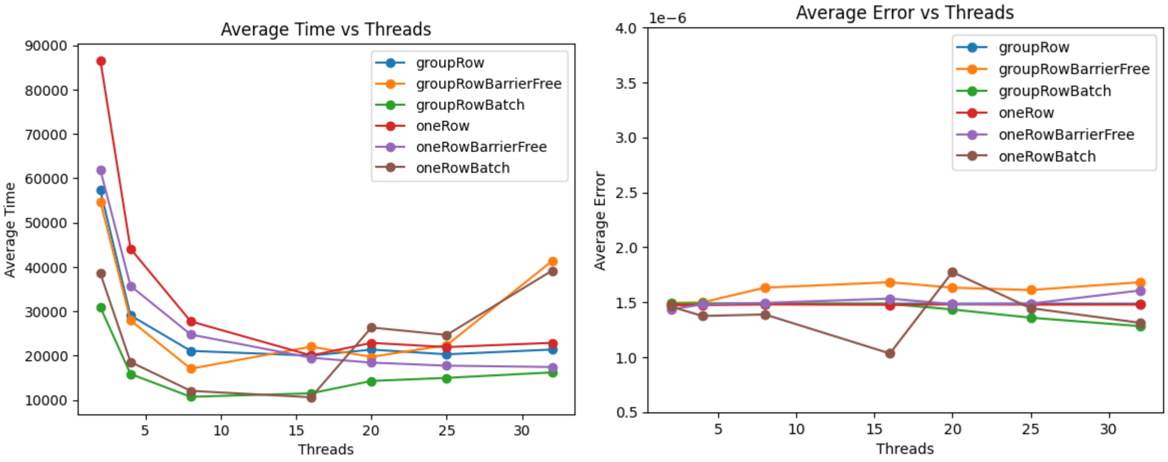
Figure 5: Barrier vs BarrierWithBatch vs BarrierFree using Static Allocation (# threads = 16)



(a) Time vs Grid Size

(b) Error vs Grid Size

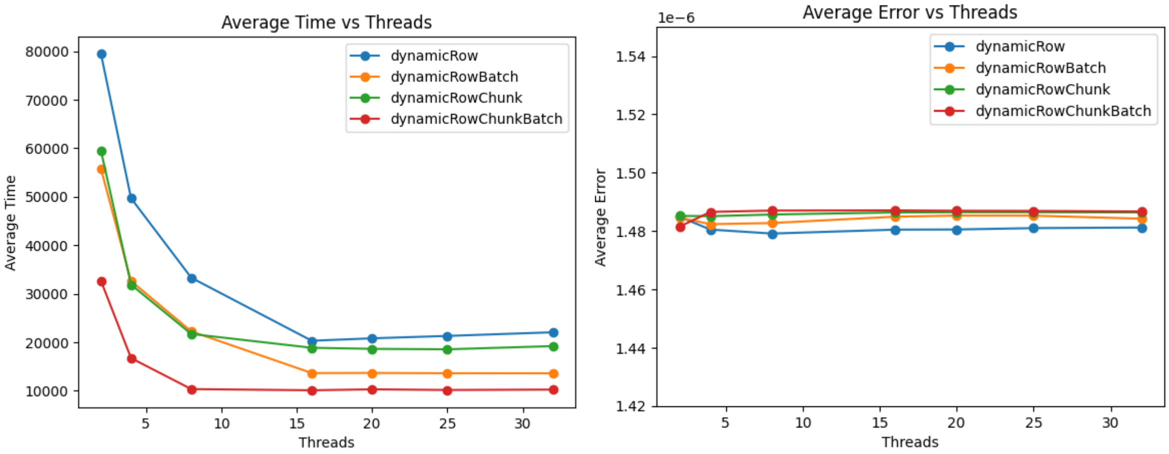
Figure 6: Comparison of Barrier and BarrierWithBatch using dynamic allocation (# threads = 16)



(a) Time vs Threads

(b) Error vs Threads

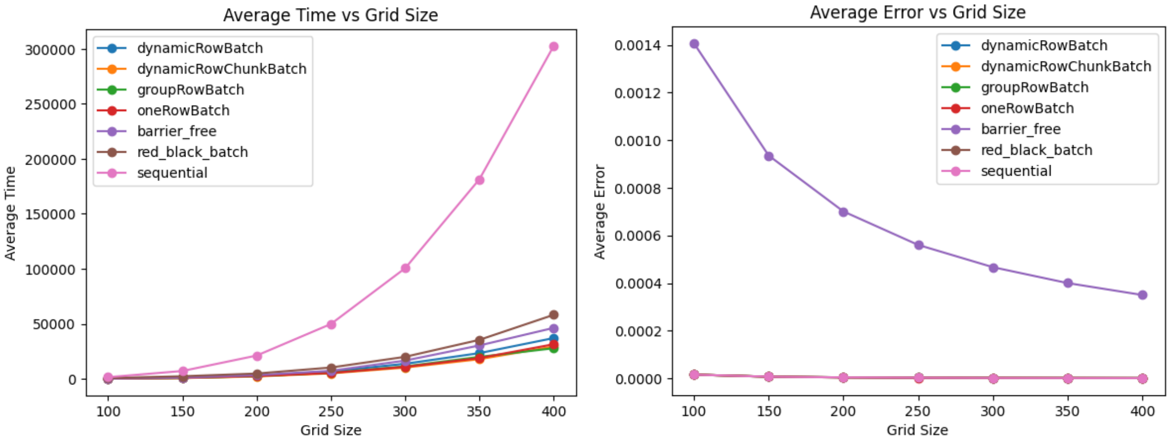
Figure 7: Barrier vs BarrierWithBatch vs BarrierFree using Static Allocation (GridSize: 300 x 150)



(a) Time vs Threads

(b) Error vs Threads

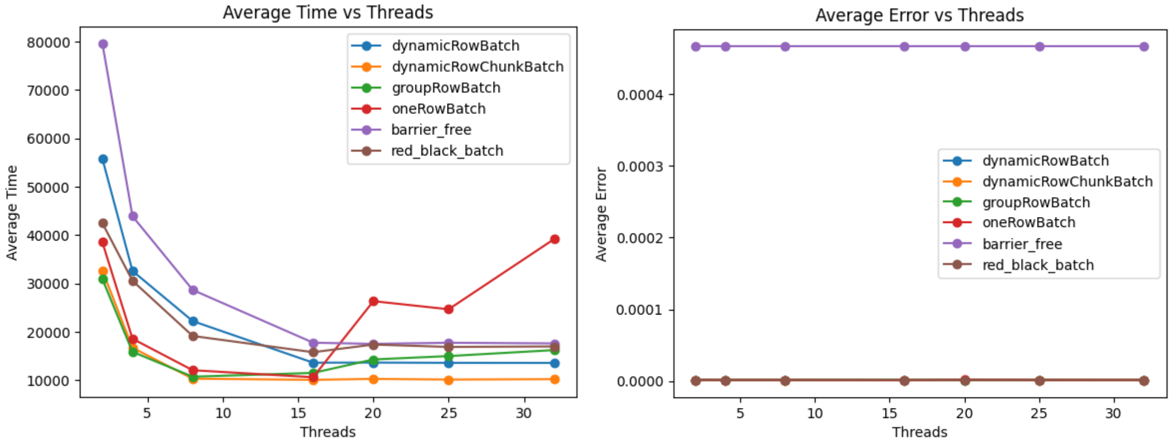
Figure 8: Barrier vs BarrierWithBatch using dynamic allocation (GridSize: 300 x 150)



(a) Time vs Grid Size

(b) Error vs Grid Size

Figure 9: Sequential vs RedBlack vs BarrierWithBatch vs BarrierFree (# threads = 16)



(a) Time vs Threads

(b) Error vs Threads

Figure 10: Sequential vs RedBlack vs BarrierWithBatch vs BarrierFree (GridSize: 300 x 150)

6 Observations and Conclusions

Following are the observations obtained from various executions of all the methods:

- When grid size is increased, the error decreases because the values of gradients calculated are more accurate and a finer grid allows for a more detailed representation of the solution that might be missed on a coarser grid. This can be seen in Figure 4, 5 and 6.
- With the increase in threads, the error remains constant as grid size is kept constant. The error is calculated with respect to the exact solution, and the error changes with a change in grid size. This can be seen in Figure 7, 8 and 10.
- While using barriers, allocating chunks of rows to each thread is more beneficial than allocating single rows (for both static as well as dynamic allocation).
- Performing iterations in batches proves to be really useful as can be seen in Figure 7 and 8.
- The speedup obtained by methods using barriers reaches convergence after employing a limited number of threads.
- The barrier free algorithm scales really well in the sense that when the number of threads are by a factor of f, then the time taken is decreased by a factor of approx. f. However, because the threads are not synchronizing anymore, the average error increases by a significant factor. Although the error obtained using the barrier-free algorithm ($\approx 1e-4$) is significantly higher than the sequential algorithm ($\approx 1e-6$), it remains well within an acceptable range. Users, prioritizing speedup and a low overall error may find this solution satisfactory.

7 System Specification, Input File and Naming

- All performance tests are conducted on Intel(R) Core(TM) i7-10700 CPU @ 2.90GHz, having 8 cores 16 thread socket.
- Domain size is kept 0.0 to 1.0 in x-direction and 0.0 to 0.5 in y-direction.
- For a grid of dimensions (nx, ny), grid size is changed by changing nx and keeping $ny = nx/2$.

- Time is measured in milliseconds in all the experiments and plots.
- oneRow refers to static row allocation, while groupRow refers to static chunk allocation.
- dynamicRow refers to dynamic row allocation, while dynamicRowChunk refers to dynamic chunk allocation.
- If Batch is used, then BarrierWithBatch method is used, else Barrier method unless BarrierFree is not specified.

8 References

- `https://www.sciencedirect.com/science/article/pii/S089812210900042X`
- `https://github.com/rpandya1990/Gauss-seidel-Parallel-Implementation`
- `https://gist.github.com/shelterz/4b13459668eec743f15be6c200aa91b2`

Note: Link for Github repository.