EC527 Final: 3D Successive Over-Relaxation

Maiko Lum

May 1, 2025

Contents

1 Description of Algorithm

Successive Over-Relaxation (SOR) is a refinement of the Gauss-Seidel method designed to enhance convergence speed when solving large systems of linear equations. These systems typically arise from discretized versions of partial differential equations (PDEs) used in scientific simulations, such as modeling heat transfer, fluid dynamics, and electrostatics.

In three-dimensional SOR, the grid is represented as a cube with dimensions. Each grid point holds a scalar value that is iteratively updated based on the values of its six immediate neighbors: left/right (x-axis), front/back (y-axis), and top/bottom (z-axis). Ghost zones (extra layers) are added around the cube to simplify the boundary conditions and ensure valid neighbor access during updates.

The iterative formula used is:

$$new_{value} = old_{value} - \omega \cdot (old_{value} - \frac{1}{6}\sum(neighbors))$$

The value of the relaxation factor controls the degree of over-relaxation. Choosing an optimal value (usually between 1 and 1.99) accelerates convergence without compromising stability. The ω value of 1.83 was used throughout the project. The iterations continue until the average change across all points falls below a predefined tolerance of 10^{-5} (can vary).

2 Scalar CPU Version

The scalar version of the 3D SOR algorithm was implemented as a single-threaded baseline to establish correctness and provide a performance reference for evaluating the multithreaded implementations. This version operates sequentially over the entire 3D grid, updating one grid point at a time during each iteration.

The grid is stored as a 1D array in memory, with a helper function `idx3D(i,j,k,n)` used to compute the correct linear index for a 3D coordinate. This layout supports better memory locality compared to using a traditional 3D array structure in C. Ghost cells are included along each dimension to simplify boundary condition handling.

2.1 Where Time Is Spent

In each iteration, the scalar implementation performs the following for every interior grid point:

- 6 memory reads from the neighboring cells
- 1 memory read/write for the center point being updated
- 1 subtraction and 1 multiplication to compute the relaxation step
- 1 accumulation into a total-change variable for convergence testing

Given this, the majority of time is spent on:

- Memory accesses: 7 per grid point, which dominate the runtime due to relatively slow memory bandwidth
- Floating-point operations: Approximately 10 FLOPs per grid point

This heavy reliance on memory makes the SOR method memory-bound on most modern CPU architectures.

2.2 Arithmetic Intensity

Arithmetic intensity is defined as the ratio of floating-point operations to memory operations. For 3D SOR:

$$Arithmetic\ Intensity\ = \frac{10\ FLOPS}{7\ memory\ operations}\ \approx 1.43\ FLOPs/Byte$$

Compared to modern CPUs, which can handle 10–30 FLOP/byte, SOR utilizes only a small fraction of the available computational throughput. This further confirms that optimizing memory access – rather than computational logic – is the key to improving performance.

2.3 Code and Discussion

The core algorithm for the serial implementation, are shown as such:

```
void SOR3D_serial(cube_ptr c, int *iterations)
{
  int i, j, k;
  int n = c->n;
  data_t *data = c->data;
```

```
double change, total_change = 1.0e10;   /* start w/ something big */
int iters = 0;
int idx;

//Main iteration loop
while ((total_change / (double)(n*n*n)) > (double)TOL) {
  iters++;
  total_change = 0.0;

  //Process all interior 6 points of the cube
  for (k = 1; k < n-1; k++) {        //z-coordinate
    for (j = 1; j < n-1; j++) {      //y-coordinate
      for (i = 1; i < n-1; i++) {    //x-coordinate
        idx = idx3D(i, j, k, n);

        //Calculate change based on 6 neighbors
        change = data[idx] - (1.0/6.0) * (
          data[idx3D(i-1, j, k, n)] +  //left neighbor (x-1)
          data[idx3D(i+1, j, k, n)] +  //right neighbor (x+1)
          data[idx3D(i, j-1, k, n)] +  //down neighbor (y-1)
          data[idx3D(i, j+1, k, n)] +  //up neighbor (y+1)
          data[idx3D(i, j, k-1, n)] +  //back neighbor (z-1)
          data[idx3D(i, j, k+1, n)]    //front neighbor (z+1)
        );

        //Apply SOR update
        data[idx] -= change * OMEGA;

        //Track convergence
        if (change < 0) {
          change = -change;
        }
        total_change += change;
      }
    }
  }

  //Check for divergence
```

```
    int last_idx = idx3D(n-2, n-2, n-2, n);
    if (fabs(data[last_idx]) > 10.0 * (MAXVAL - MINVAL)) {
      printf("SOR3D: SUSPECT DIVERGENCE iter = %d\n", iters);
      break;
    }
  }

  *iterations = iters;
  printf("    SOR3D_serial() done after %d iters\n", iters);
}
```

Each iteration involves:

- Scanning the 3D domain from indices 1 to $n - 2$ in all directions, ignoring the ghost zones
- Reading the six neighboring values of each grid point
- Computing the new value and accumulating the magnitude of the change
- Checking if the average change per point has dropped below the tolerance threshold

This version uses helper functions like `new_cube()` for allocation, `init_cube_rand()` for random initialization of values, and `idx3D()` for efficient memory indexing.

In terms of complexity:

- Time: $O(k \cdot n^3)$, where k is the number of iterations required for convergence. Since $k \propto n$, the effective complexity is $O(n^4)$
- Space: $O(n^3)$, as the entire cube is stored in memory

As a result, while the scalar CPU version is essential for establishing functional correctness, it quickly becomes impractical for large-scale simulations due to its poor scalability and inefficient hardware utilization.

3 Parallel Implementation – Multithreading

Given the limitations of the serial implementation, particularly its inability to efficiently handle large datasets due to its sequential nature, we explored parallelizing the 3D SOR algorithm using multithreading. The goal was to divide the computational workload among multiple threads to reduce total runtime while preserving the correctness and convergence behavior of the algorithm.

The SOR algorithm has inherent data dependencies – each point in the grid depends on its immediate neighbors. Therefore, care must be taken to ensure that threads do not simultaneously read and write to shared memory locations, which could result in race conditions or inconsistent results. To address this, we employed domain decomposition – dividing the 3D cube along a specific axis so that each thread works on a distinct and independent subset of the data.

Three parallel versions were implemented, each slicing the domain along one of the principal axes (x, y, z). These slices were then assigned to threads. In each case, the threads execute the same update logic on their assigned subvolume of the grid. After each iteration, all threads synchronize using a pthread barrier to ensure that one iteration is fully completed across all slices before beginning the next. This synchronization guarantees that each thread has the latest values from all neighbors before proceeding.

Additional synchronization is required for convergence checking. Each thread calculates its own local change (the cumulative difference in grid values for its slice), and then a designated thread (in this case, thread 0) aggregates these local changes to compute the global average change. This value is then compared to the tolerance threshold to determine whether further iterations are needed.

Key Considerations in Multithreading:

- Thread Synchronization: implemented via barriers to ensure consistent state before and after each iteration
- Load Balancing: domain is split evenly among threads. Uneven domain sizes or poor alignment with the axis of decomposition can cause load imbalance
- Memory Access Patterns: The choice of decomposition axis has a significant impact on performance due to memory layout
- Scalability: Performance improvements are most pronounced when the decomposition aligns with the system's memory architecture and when the number of threads matches the available hardware threads (e.g., CPU cores)

Comparing these implementations aimed to determine not just how multithreading can improve performance, but how the direction of decomposition affects runtime due to memory access efficiency and cache behavior. Our results show that choosing the right slicing strategy is just as important as the decision to parallelize itself.

3.1 X-axis Partition

X-axis partitioning divides the cube into slabs perpendicular to the x-axis. Each thread is assigned a set of x-slices (planes where x varies and y, z are fixed). Due to how C stores arrays in memory (row-major order), accessing adjacent elements along the x-axis requires jumping across memory locations.

This partitioning results in high memory access latency and reduced spatial locality, as the threads access non-contiguous memory segments. Consequently, CPU cache lines are frequently invalidated, and performance degrades due to cache thrashing. Despite an even computational load across threads, the inefficiency in memory access overshadows potential benefits.

- Implementation Simplicity: straightforward index slicing
- Cache Performance: poor, frequent cache misses
- Use Case: rarely recommended unless memory access patterns can be optimized

3.2 Y-axis Partitioning

Y-axis partitioning assigns threads to work on y-slices of the cube. In this method, each thread updates all grid points in a subset of the y-dimension, meaning y varies while x and z loop within the thread's slice.

Memory stride improves compared to x-axis slicing because adjacent y-slices are spaced elements apart in memory, which is smaller than the stride of in the x-direction. As a result, cache reuse is more likely, although not as efficient as z-axis access.

- Memory Stride: moderate, accesses are more cache-coherent
- Scalability: reasonable for medium-sized grids.
- Thread Balance: fair, can degrade for uneven dimensions or thread counts.

3.3 Z-axis Partition

Z-axis partitioning is the most efficient method, where each thread processes a subset of z-planes. This configuration aligns perfectly with how C stores arrays in memory I – consecutively along the z-axis (assuming x and y are the innermost loops).

The threads access contiguous memory regions, maximizing spatial locality, minimizing cache misses, and benefiting from CPU hardware prefetching. Because each thread's slice forms a block in memory, there's little to no interference across threads, enabling fast and scalable parallelism.

- Memory Access: optimal, excellent spatial locality
- Performance: best of all strategies, scales well with increasing thread count
- Best Use Case: large grids with high memory bandwidth demands

4 Multithreading Results

All versions were tested using varying cube sizes calculated from the function $n = Ax^2 + Bx + C$. For each size, execution time and number of iterations were recorded.

While all partitioning strategies can be used to parallelize SOR, z-axis decomposition offers the most favorable trade-off between load balancing, synchronization simplicity, and memory performance. The contrast between slicing strategies clearly demonstrates the importance of matching parallel algorithms with underlying hardware memory models.

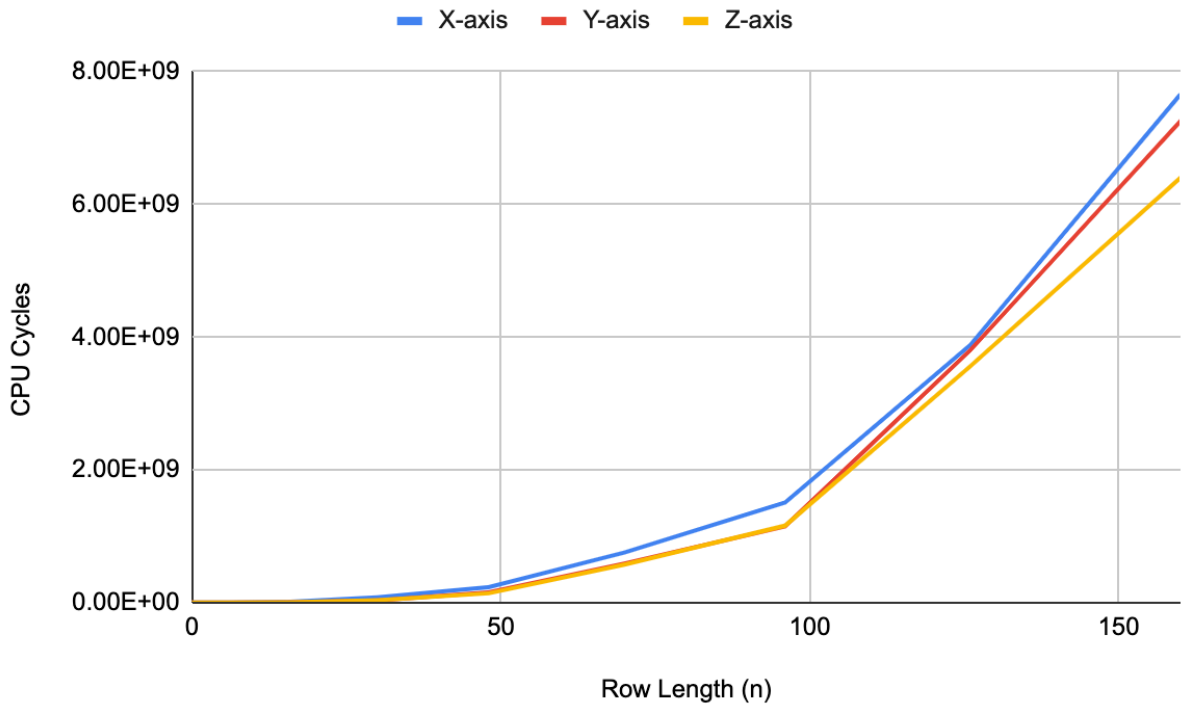

Figure 1: Results of different partitioning methods (x,y,z) at varying cube size with four threads.

Among the three slicing strategies, Z-partitioning showed the most consistent and scalable speedup. Because of its contiguous memory access, it places less strain on the memory subsystem, allowing more efficient use of multiple threads.

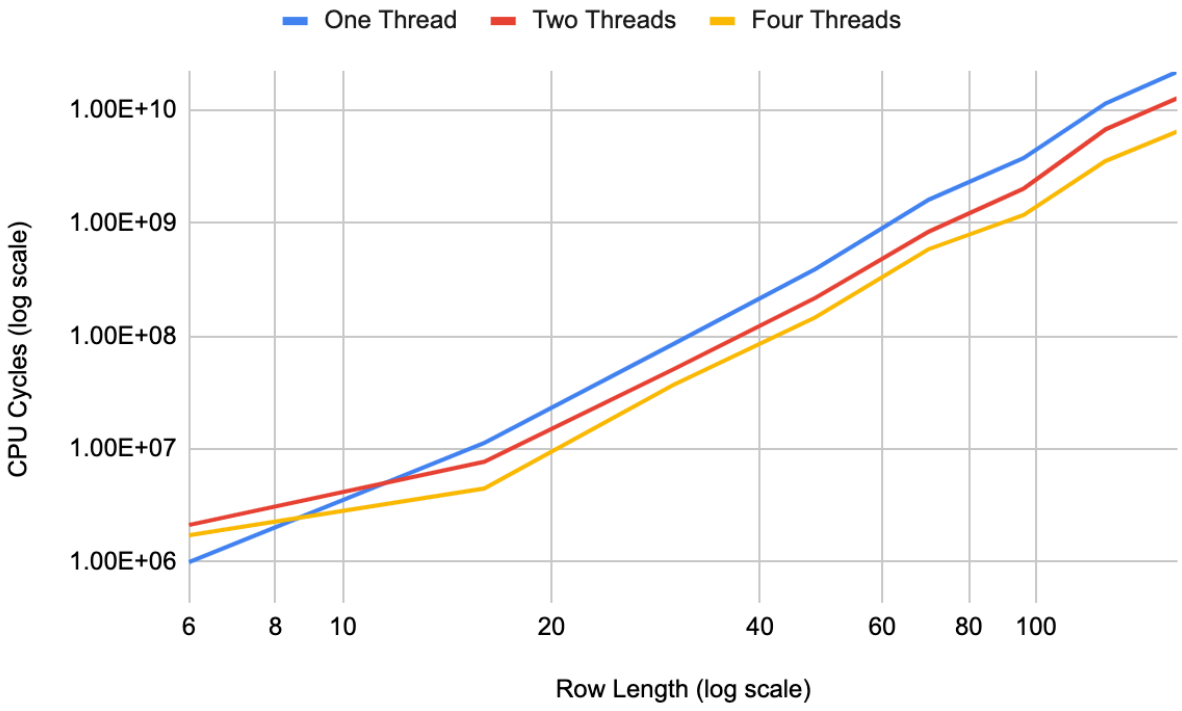

Figure 2: Performance of z partitioned multi-threaded code at varying thread count.

As more threads are introduced, the total computational workload is divided into smaller chunks, reducing the amount of work each individual thread must perform per iteration.

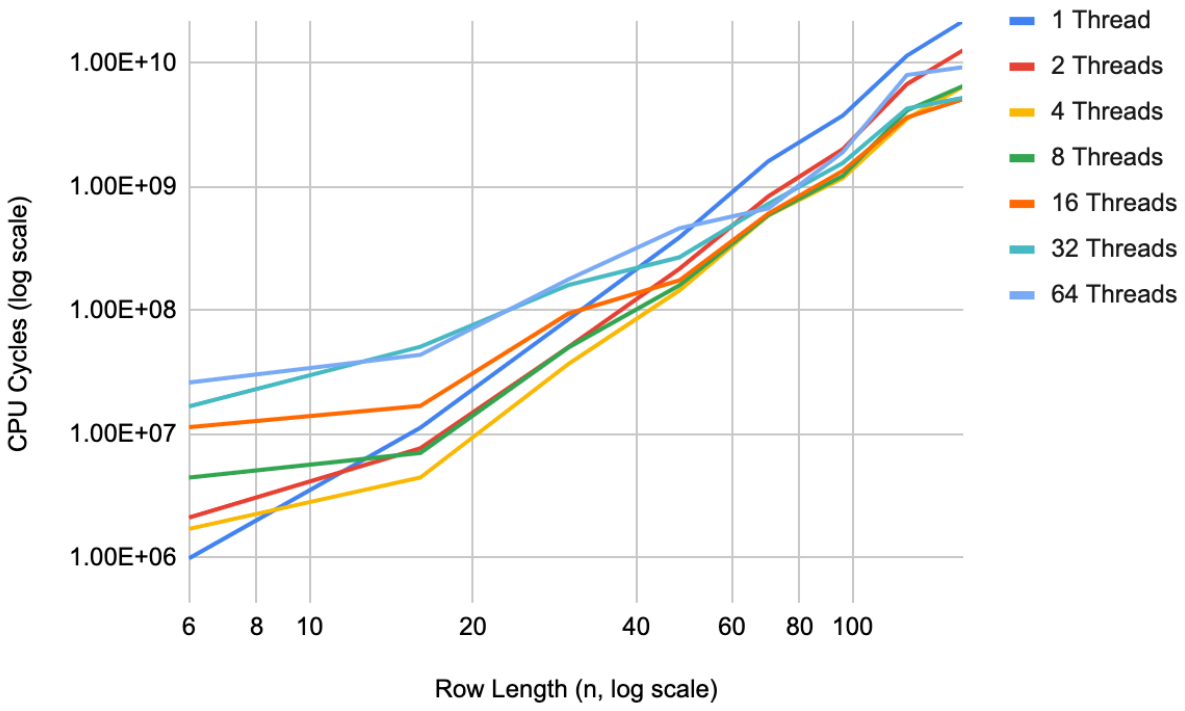

Figure 3: Extension of Fig. 2 with increased variance in thread count.

The most significant performance gains from multithreading were observed when the number of threads matched or was slightly below the number of available physical CPU cores. This configuration minimizes overhead and maximizes parallel efficiency. However, as the thread count exceeds the core count, performance improvements begin to diminish. This is primarily due to increased context switching and cache thrashing, which introduce latency and reduce effective throughput. Additionally, synchronization overhead becomes more noticeable with a higher number of threads. At the end of each iteration, all threads must reach a barrier before proceeding, which can lead to idle time if faster threads must wait for slower ones to catch up. This imbalance, while necessary for correctness, limits the benefits of adding more threads beyond a certain point.

5 Discussion

The results demonstrate that the performance of the 3D SOR algorithm is strongly influenced by both memory access patterns and thread synchronization overhead. While

all parallel implementations reduced runtime compared to the serial baseline, the extent of improvement varied significantly with the chosen partitioning strategy.

The Z-axis decomposition consistently delivered the best performance. This is primarily due to its alignment with the row-major layout of C arrays, which enables threads to operate on contiguous blocks of memory. As a result, cache utilization is maximized, and the memory subsystem is used efficiently. In contrast, X-axis partitioning suffered from poor spatial locality. Accesses in the x-direction incur large memory strides, leading to frequent cache misses and reduced throughput. Y-axis slicing provided intermediate performance, offering improved locality over the x-direction but not matching the contiguity of z-slices.

In terms of scalability, the performance improved with increasing thread count up to the number of available physical cores. Beyond this point, speedup gains diminished due to overhead from context switching and contention for shared resources such as caches and memory bandwidth. Notably, the barrier synchronization required at the end of each iteration introduced further delays when threads completed their assigned work at different rates. This imbalance became more pronounced as the thread count increased, especially in the x-partitioned version where memory access latency was higher.

The number of iterations required to converge remained approximately constant across all versions, indicating that the decomposition strategy did not affect numerical convergence. Therefore, the differences in runtime can be attributed almost entirely to differences in execution efficiency.

Overall, the experiment confirms that parallelism alone does not guarantee optimal performance. For memory-bound algorithms like SOR, the choice of decomposition axis has a significant impact. Z-axis partitioning emerges as the most effective strategy due to its alignment with data layout and resulting cache efficiency. These findings underscore the importance of memory-aware design when developing high-performance numerical applications.

6 Closing Thoughts

First of all, let me say this clearly: memory access is the real bottleneck for 3D SOR. I've seen it in every variant I've tried – x, y, z – it's not the math that slows you down, it's how you access memory.

We've shown again and again that z-axis slicing absolutely dominates for this kind of algorithm. It's not just a little faster, it's consistently better because of how memory is laid out. Row-major order loves z-plane partitioning. X-axis? It's the slowest by far. Even with perfect thread balance, you just can't overcome bad locality. Y-axis isn't terrible, but it's just okay.

Multithreading definitely helped. The improvement from 1 to 4 threads was solid. Even around 16 threads was doing well. Past that? You start to feel the pain of barrier syncs and memory contention. Threads waiting on others, contention for shared cache, and the unavoidable truth that there's only so much bandwidth to go around.

So what's the takeaway? Smart memory layout > more threads. If you don't respect the memory system, no amount of parallelism is going to bail you out. Going forward, some exciting things to look at would be:

1. Trying out OpenMP or hybrid models to reduce pthread boilerplate and scale cleanly
2. Pushing to the GPU to see if memory coalescing can unlock even more
3. Seeing what happens at other partitioning methods like partitioning as blocks rather than slabs across the axis

This project was a great sandbox to learn the trade-offs between algorithm design and system architecture. It's one thing to write a numerically stable solver. It's another thing to make it run fast on real hardware.

7 Compilation Instructions

7.1 Scalar

```
gcc -O1 -std=gnu11 serial_SOR3D.c -lm -o serial_SOR3D
```

7.2 Multithreading

7.2.1 X-axis

```
gcc -O1 -std=gnu11 3DSORx.c -lm -pthread -o parallel_SOR3D_x
```

7.2.2 Y-axis

```
gcc -O1 -std=gnu11 3DSORy.c -lm -pthread -o parallel_SOR3D_y
```

7.2.3 Z-axis

```
gcc -O1 -std=gnu11 3DSORz.c -lm -pthread -o parallel_SOR3D_z
```

8 List of Files

The files included in the *.zip file include:

- `final_report.pdf`: the report you're reading.
- `serial_SOR3D.c`: serial implementation of 3D SOR
- `3DSORx.c`: multithreaded 3D SOR partitioned at X-axis
- `3DSORy.c`: multithreaded 3D SOR partitioned at Y-axis
- `3DSORz.c`: multithreaded 3D SOR partitioned at Z-axis