

# 41391 High Performance Computing: FORTRAN, OpenMP and MPI

## Assignment 2: The Poisson Problem

Brandon Parks

DTU - Department of Solid Mechanics (TopOpt)

January 20, 2026

The work is solely created, executed and reported by the author in its entirety. In accordance with the Technical University of Denmark's guidelines on the use of Large Language Models (LLMs) in academic work, the following disclosure is made:

The author has utilized the Github Copilot [1] LLM to assist in the writing and editing of this thesis, specifically using the GPT-5.2 model. These LLMs are employed strictly to help navigate diction and accelerate the development of code in Fortran (and subsequently generated Matlab scripts), primarily through the correction of syntax. In addition, no code fragments are synthesized beyond the author's existing ability to create them independent of LLM assistance at the time of writing.

The primary interfaces for the use of LLMs are the built-in VScode integration with Github Copilot and the desktop application for ChatGPT.

# 1 Iterative Algorithm Implementations

The Poisson problem described in [2] is solved using both of these solvers on a three-dimensional grid with prescribed Dirichlet boundary conditions and a volumetric heating source. The boundaries at  $(x, y, z) = \pm 1$  are completely discluded from the analysis to enforce the Dirichlet boundary conditions. The finite difference method is utilized to solve this problem. The solution is shown below in Figure 1 at selected sections through the 3D domain. The solution is only shown for one implementation realization as the results are identical for all algorithms used to a reasonable tolerance.

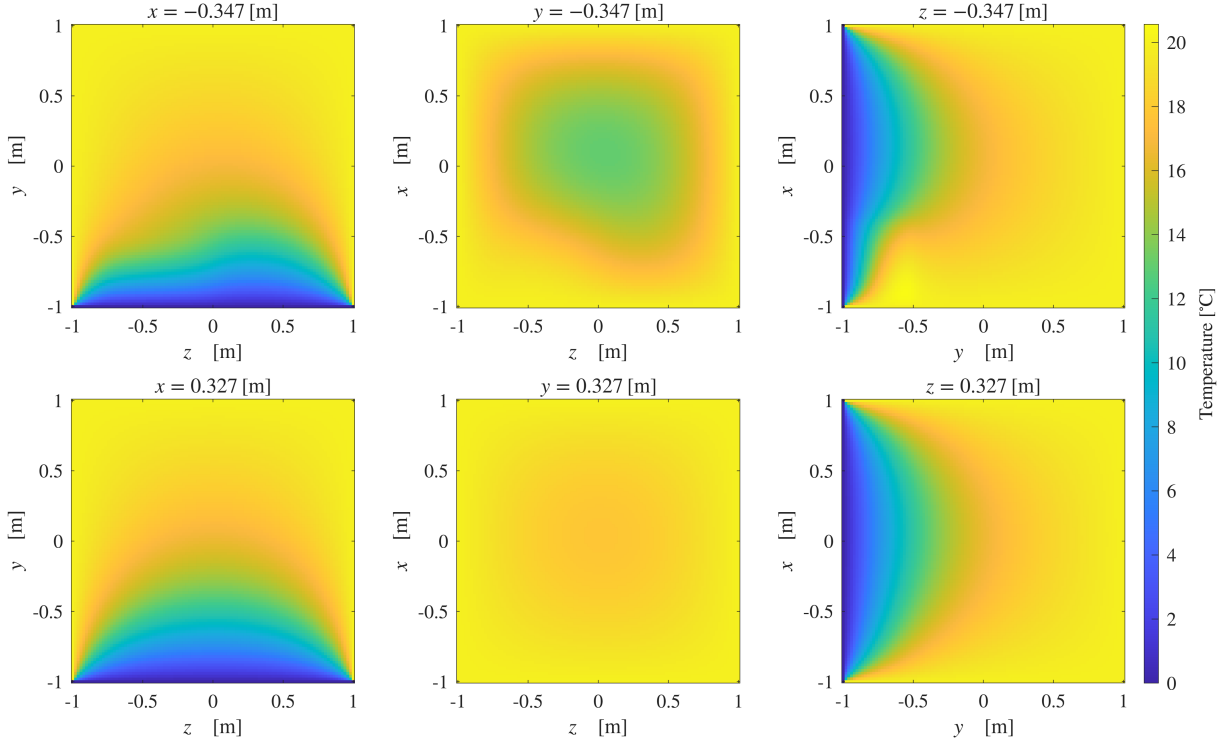


Figure 1: Select slices of the solution space  $\mathbf{u}$  for the 3D Poisson problem shown in [2] for 100 interior grid points per spatial dimension, or 1 million DOF.

The Jacobi and Gauss-Seidel iterative solvers are both implemented in Fortran for both sequential and parallel execution. For the parallel implementation, multiple versions are created to demonstrate the criticality of proper parallelization. The performance of the parallel portions of code (or the solver algorithm iterations for the sequential version) can be seen in Figure 3. Here both the initial and final implementations are compared with the theoretical maximum speed-up defined by Amdahl's Law. The benchmark for the speed-up multiplier is the sequential implementation for the specific iterative solver.

## 1.1 The Jacobi Iterative Solver

The Jacobi iterative solver uses solution information adjacent to each solution point to approach the true solution in an iterative manner. The important component of this solver is that it only uses solution information from previous iterations to improve upon the upcoming ones. This allows the

solver to be parallelized over each iteration. First, in a sequential setting the solver is fairly simple to implement by just looping over each solution location  $u_{i,j,k}^t$  and updating the next iteration's solution  $u_{i,j,k}^{t+1}$  where  $t$  represents the iteration number for the solver and  $i, j, k$  the spatial locations in the problem domain  $\Omega$ . This is implemented for a single iteration as:

Listing 1: Jacobi Primary Spatial/Interior Point Loop in Fortran

```

1  ... ! Beginning of iteration loop
2  ! Update interior points
3  do k = 2, n - 1
4      do j = 2, n - 1
5          do i = 2, n - 1
6              u_new_val = ( & ! compute next iteration solution value
7                  u_old(i - 1, j, k) + u_old(i + 1, j, k) + &
8                  u_old(i, j - 1, k) + u_old(i, j + 1, k) + &
9                  u_old(i, j, k - 1) + u_old(i, j, k + 1) + &
10                 delta2 * f(i, j, k) ) / 6._dp
11              diff = abs(u_new_val - u_old(i, j, k)) ! compute max solution difference
12              if (diff > diffmax) diffmax = diff
13              u_new(i, j, k) = u_new_val
14          end do
15      end do
16  end do
17  ... ! Convergence check & end of iteration loop

```

Again, it is clear by inspection of Listing 1 that each update of  $u_{i,j,k}^{t+1}$  only depends on values from the previous iteration  $u_{i,j,k}^t$  and is thus easily parallelizable. In accordance with the boundary conditions specific for this problem, only interior points are updated by excluding the first and last indices in each spatial dimension  $i, j, k$ .

### 1.1.1 Parallelization

As previously mentioned, the first and easiest way to parallelize this solver is to distribute the update of each solution point  $u_{i,j,k}^{t+1}$  equally across all recruited threads. A first-pass implementation is easily accomplished with OpenMP by adding a single compiler directive to the primary spatial loop as shown in Listing 2.

Listing 2: Parallelized Jacobi Primary Spatial/Interior Point Loop in Fortran (Version 1)

```

1  !$omp parallel default(shared) ! Double check number of worker threads
2  !$omp single
3  print '(a,i0)', 'OpenMP: jacobi threads used = ', omp_get_num_threads()
4  !$omp end single
5  !$omp end parallel
6  do iter = 1, itermax
7      t0_par = omp_get_wtime() ! time parallel portion only
8      !$omp parallel do default(shared) private(i, j, k, u_new_val, diff) &
9      !$omp collapse(3) schedule(static) reduction(max:diffmax)
10     do k = 2, n - 1

```

```

11     do j = 2, n - 1
12         do i = 2, n - 1
13             u_new_val = ( &
14                 u_old(i - 1, j, k) + u_old(i + 1, j, k) + &
15                 u_old(i, j - 1, k) + u_old(i, j + 1, k) + &
16                 u_old(i, j, k - 1) + u_old(i, j, k + 1) + &
17                 delta2 * f(i, j, k) ) / 6._dp
18             diff = abs(u_new_val - u_old(i, j, k))
19             if (diff > diffmax) diffmax = diff
20             u_new(i, j, k) = u_new_val
21         end do
22     end do
23 end do
24 !$omp end parallel do
25 t_par_total = t_par_total + (omp_get_wtime() - t0_par)
26 if (diffmax < tolerance) exit
27 ! Swap buffers
28 end do

```

Of key importance is the handling of the variable `diffmax` which is used to track the maximum change in solution value across all solution points for convergence checking. Since multiple threads will be updating this variable simultaneously, disclusion of the reduction operator seen on Line 9 in Listing 2 will generate a race condition leading to significantly decreased performance and incorrect results.

An obvious improvement to this approach is to now initialize the solution and forcing arrays,  $u_{i,j,k}^0$  and  $f_{i,j,k}$ , in parallel as well. At the same time, the current and next iteration solution arrays used in the Jacobi algorithm are distributed proportionally to the selected worker threads to help optimize for NUMA during the main parallel section. This places data that each thread will need to access in close proximity (i.e. in the local core's cache) rather than in another core's cache or main memory. These additions can be seen in Listing 4.

Listing 3: Solution and Forcing Arrays Initialized in Parallel in Fortran

```

1  ! ---- 'init' Subroutine ----
2  !$omp parallel default(shared) private(i, j, k, x, y, z)
3  !$omp do schedule(static)
4  do k = 1, n
5      do j = 1, n
6          do i = 1, n
7              ! Start values (interior defaults)
8              ! Radiator source term f(x,y,z)
9              ! Dirichlet boundary conditions:
10         end do
11     end do
12 end do
13 !$omp end do
14 !$omp end parallel
15

```

```

16  ! ---- 'jacobi' Subroutine ----
17  ! NUMA first-touch for the Jacobi buffers: initialize/copy in parallel
18  !$omp parallel do collapse(2) default(shared) private(i, j, k) schedule(static)
19      do k = 1, n
20          do j = 1, n
21              do i = 1, n
22                  u_old(i, j, k) = u(i, j, k)
23                  u_new(i, j, k) = u(i, j, k)
24              end do
25          end do
26      end do
27  !$omp end parallel do

```

Returning to the main parallel section shown in [Listing 2](#), further improvement may be gained by vectorizing the innermost loop. Doing so effectively assigns work chunks corresponding to complete sections (strips) of cache memory and assigns it to worker threads such that other threads will never need to access memory from another thread's cache. This avoids a severe performance penalty common in poorly optimized parallel code where different threads need access to data in the same memory strips. This parallel configuration also avoids a fork/join every Jacobi iteration which is costly.

Listing 4: Parallelized Jacobi Primary Spatial/Interior Point Loop in Fortran (Version 2)

```

1  !$omp parallel default(shared) private(i, j, k, u_new_val, diffmax_line, iter)
2  do iter = 1, itermax
3      !$omp single
4      diffmax = 0._dp
5      t0_par = omp_get_wtime()  ! Time only the parallelized update portion
6      !$omp end single
7      !$omp do collapse(2) schedule(static) reduction(max:diffmax)
8      do k = 2, n - 1
9          do j = 2, n - 1
10             diffmax_line = 0._dp
11             !$omp simd reduction(max:diffmax_line)
12             do i = 2, n - 1
13                 u_new_val = ( &
14                     u_old(i - 1, j, k) + u_old(i + 1, j, k) + &
15                     u_old(i, j - 1, k) + u_old(i, j + 1, k) + &
16                     u_old(i, j, k - 1) + u_old(i, j, k + 1) + &
17                     delta2 * f(i, j, k) ) / 6._dp
18                 diffmax_line = max(diffmax_line, abs(u_new_val - u_old(i, j, k)))
19                 u_new(i, j, k) = u_new_val
20             end do
21             diffmax = max(diffmax, diffmax_line)
22         end do
23     end do
24     !$omp end do
25     !$omp single

```

```

26     t_par_total = t_par_total + (omp_get_wtime() - t0_par)
27     ! Convergence check w/ diffmax
28     ! Swap buffers
29     !$omp end single
30     !$omp barrier
31 end do
32 !$omp end parallel
33 ... ! Convergence check & end of iteration loop

```

## 1.2 The Gauss-Seidel Iterative Solver

Very similar to the Jacobi solver, the Gauss-Seidel iterative solver uses adjacent solution information to approach the true solution in an iterative manner. The key difference comes from the Gauss-Seidel solver using the most up-to-date solution information available within the same iteration to improve upon the upcoming solution points. Parallelized in the most simple way, this would create a loop-carried dependency as approximately half of the solution terms at each solution point  $u_{i,j,k}^{t+1}$  would depend on other solution points from the same  $t + 1$  iteration, and this information is necessarily spread throughout the memory space. Thus, a more sophisticated approach is needed to parallelize this solver. The sequential implementation, however, is somehow simpler. By only working on a single solution variable  $\mathbf{u}$  rather than a new,  $\mathbf{u}_{new}$ , and old,  $\mathbf{u}_{old}$ , variable pair the stencil-update loop can reference the same array which will automatically use the most up-to-date information available as the algorithm prescribes. The sequential implementation is shown in [Listing 5](#).

Listing 5: Gauss-Seidel Primary Spatial/Interior Point Loop in Fortran

```

1  ... ! Beginning of iteration loop
2  ! Update interior points only
3  do k = 2, n - 1
4      do j = 2, n - 1
5          do i = 2, n - 1
6              u_old_val = u(i, j, k)
7              u_new_val = ( &
8                  u(i - 1, j, k) + u(i + 1, j, k) + &
9                  u(i, j - 1, k) + u(i, j + 1, k) + &
10                 u(i, j, k - 1) + u(i, j, k + 1) + &
11                 delta2 * f(i, j, k) ) / 6._dp
12              u(i, j, k) = u_new_val
13              diff = abs(u_new_val - u_old_val)
14              if (diff > diffmax) diffmax = diff
15          end do
16      end do
17  end do
18  ... ! Convergence check & end of iteration loop

```

Here the maximum change between iterations and subsequent convergence check is kept only for the sequential implementation as it comes almost for free. As will be shown, this is neglected for the parallel implementation and only the maximum iteration count is retained.

### 1.2.1 Parallelization

As discussed previously, the Gauss-Seidel solver uses the most up-to-date solution information at all update steps which creates a very large loop-carried dependency. Two methods to avoid this dependency are now implemented. The first parallel implementation is based on a red-black coloring scheme where adjacent solution points alternate being updated until convergence. This allows half the solution space to be parallelized at a time and then the other half to then be updated with the information from the first half. This implementation is shown in [Listing 6](#). Parallelized initialization as shown in [Listing 4](#) is also used for the parallelized Gauss-Seidel implementations.

Listing 6: Gauss-Seidel (Red-Black Method) Primary Spatial/Interior Point Loop in Fortran

```
1  !$omp parallel default(shared) private(i, j, k, u_new_val, iter)
2  do iter = 1, itermax
3      ! Color 1 sweep: points where (i+j+k) is even
4      !$omp do collapse(2) schedule(static)
5      do k = 2, n - 1
6          do j = 2, n - 1
7              do i = 2 + mod(j + k, 2), n - 1, 2
8                  u_new_val = ( &
9                      u(i - 1, j, k) + u(i + 1, j, k) + &
10                     u(i, j - 1, k) + u(i, j + 1, k) + &
11                     u(i, j, k - 1) + u(i, j, k + 1) + &
12                     delta2 * f(i, j, k) ) * inv6
13                  u(i, j, k) = u_new_val
14              end do
15          end do
16      end do
17      !$omp end do
18      ! Color 2 sweep: points where (i+j+k) is odd
19      !$omp do collapse(2) schedule(static)
20      do k = 2, n - 1
21          do j = 2, n - 1
22              do i = 2 + mod(j + k + 1, 2), n - 1, 2
23                  u_new_val = ( &
24                      u(i - 1, j, k) + u(i + 1, j, k) + &
25                      u(i, j - 1, k) + u(i, j + 1, k) + &
26                      u(i, j, k - 1) + u(i, j, k + 1) + &
27                      delta2 * f(i, j, k) ) * inv6
28                  u(i, j, k) = u_new_val
29              end do
30          end do
31      end do
32      !$omp end do
33  end do
34  !$omp end parallel
```

The second parallel implementation uses a wavefront approach where solution planes are updated

diagonally across the 3D domain. This allows for more concurrency than the red-black scheme as multiple planes can be updated simultaneously rather than just half the solution space at a time. Practically, this is implemented using doacross loops. This is shown in [Listing 7](#).

Listing 7: Gauss-Seidel (Wavefront Method) Primary Spatial/Interior Point Loop in Fortran

```

1  !$omp parallel default(shared) &
2  !$omp private(i,j,k,tj,tk,j0,j1,k0,k1,iter,u_new_val,u_old_val) &
3  !$omp shared(n,u,f,delta2,inv6,itermax,tolerance,jb,kb,ntj,ntk,threads_used_local)
4  do iter = 1, itermax
5      !$omp do ordered(2) collapse(2) schedule(static,1)
6      do tk = 1, ntk
7          do tj = 1, ntj
8              if (tk > 1) then
9                  !$omp ordered depend(sink: tk-1, tj)
10             end if
11             if (tj > 1) then
12                 !$omp ordered depend(sink: tk, tj-1)
13             end if
14             ! Compute physical bounds for this tile
15             k0 = 2 + (tk-1)*kb
16             k1 = min(n - 1, k0 + kb - 1)
17             j0 = 2 + (tj-1)*jb
18             j1 = min(n - 1, j0 + jb - 1)
19             ! Process the tile
20             do k = k0, k1
21                 do j = j0, j1
22                     do i = 2, n - 1
23                         u_old_val = u(i, j, k)
24                         u_new_val = ( &
25                             u(i-1, j, k) + u(i+1, j, k) + &
26                             u(i, j-1, k) + u(i, j+1, k) + &
27                             u(i, j, k-1) + u(i, j, k+1) + &
28                             delta2 * f(i, j, k) ) * inv6
29                         u(i, j, k) = u_new_val
30                     end do
31                 end do
32             end do
33             !$omp ordered depend(source)
34         end do
35     end do
36     !$omp end do
37 end do
38 !$omp end parallel

```

## 2 Hardware and computing environment

The system architecture used for benchmarking in [section 3](#) is shown in [Figure 2](#) and detailed below as an automated output from the cluster. As shown in [section 3](#) with the small speed-up drop, each CPU contains 12 cores for a total of 24 cores per node.

```
-- CPU --  
Architecture:                x86_64  
CPU op-mode(s):              32-bit, 64-bit  
Address sizes:               46 bits physical, 48 bits virtual  
Byte Order:                  Little Endian  
CPU(s):                      24  
On-line CPU(s) list:         0-23  
Vendor ID:                   GenuineIntel  
Model name:                  Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz  
CPU family:                   6  
Model:                        79  
Thread(s) per core:          1  
Core(s) per socket:          12  
Socket(s):                   2  
Stepping:                    1  
CPU(s) scaling MHz:          76%  
CPU max MHz:                  2900.0000  
CPU min MHz:                  1200.0000  
BogoMIPS:                     4389.79
```

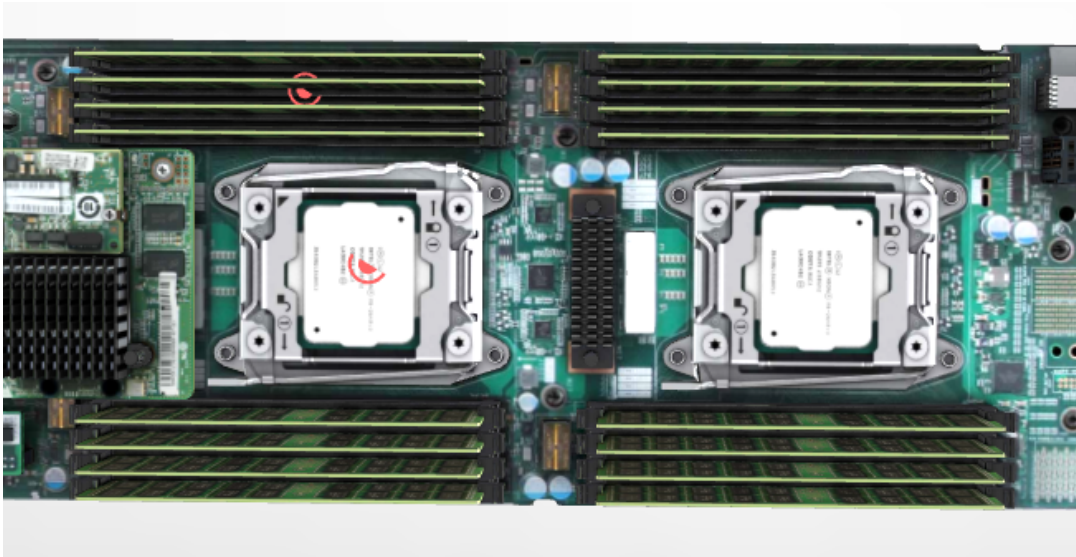


Figure 2: Physical hardware configuration used for the benchmarking shown in [section 3](#). Huawei XH620 V3 2-Socket server compute node shown with DIMMs fully populated and heat sink and outer chassis removed for clarity [3].

### 3 Performance comparisons

The following performance discussion is based on results of solving the Poisson problem as explained in [section 1](#) with problem parameters of the number of interior grid points,  $N = 100$ , a convergence tolerance of  $10^{-5}$ , a maximum of 5,000 iterations, and static chunk sizing for the OpenMP workshare.

For all results and implementations presented a checksum of a slice of the solution domain was considered to ensure numerical accuracy and check for race conditions or incorrect memory access. In all cases the checksums matched independent of the number of recruited worker threads or implementation strategy.

This problem is bound by the memory bandwidth of the underlying hardware. This is sensible as there is only really one operation being performed per iteration, but for each solution point which gives a very low FLOPs per byte of data transferred. The importance of optimizing a parallel implementation is clearly seen in [Figure 4](#) with the best-performing Jacobi and Gauss-Seidel implementations having nearly identical performance for thread counts above 4. As the Jacobi algorithm is well known to be embarrassingly parallel, it then follows that it almost perfectly follows the ideal scaling curve. However, for thread counts above 12 the influence of the dual-socket architecture becomes visible as the 13th thread is forced to be located on the second CPU further away from other threads, and the performance drop is visible in all implementations. The performance trend then continues the same as with lower thread counts but with this small penalty built-in.

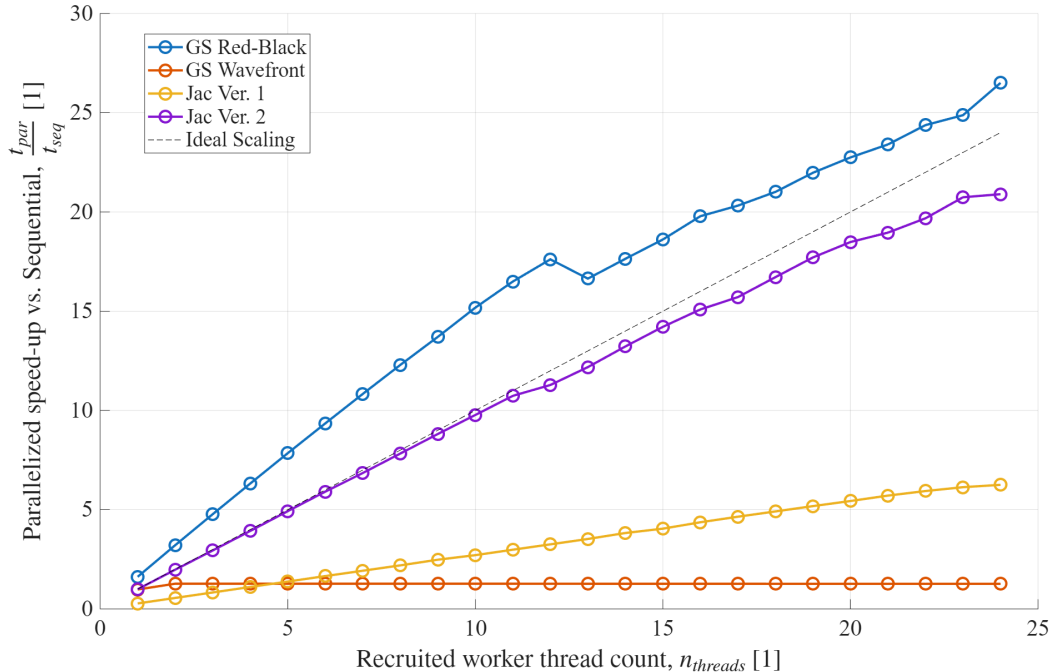


Figure 3: Speed-Ups achieved for all algorithm implementations.

The performance of the Gauss-Seidel Red-Black algorithm above the ideal scaling curve is likely due to the algorithmic difference between the parallel and the sequential implementation which was used to normalize the wall times for the speed-up calculations. However, it still performs the best

out of all implementations for all thread counts, barely beating the optimized Jacobi (Version 2) implementation. The Gauss-Seidel Wavefront implementation sees improvement from adding one additional worker thread, but then stagnates in performance from there. This is primarily from the memory bandwidth-limited nature of this algorithm together with a high parallel overhead compared to the problem size.

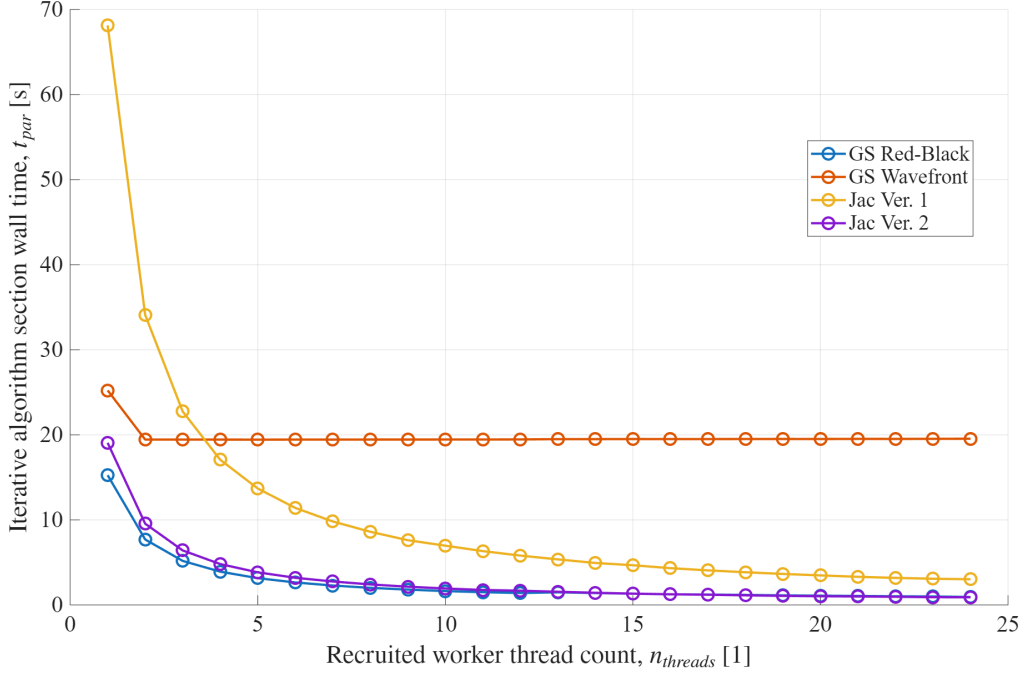


Figure 4: Total execution time for all algorithm implementations.

The FLOPs achieved by all four implementations can be seen in Table 1. As all implementations reached the max iteration count of 5000 for all cases, the FLOPs is a direct, linear scaling of the wall time results seen in Figure 4. As the operations per second only approaches and passes the processor’s clock speed of 2.2 GHz for thread counts of 8 and above, and only for the Jacobi Version 2 and Gauss-Seidel Red-Black implementations, it is again clear that the memory bandwidth is the limiting factor for this problem as the memory is not able to provide the processor with the data required to compute.

| Threads | Jacobi Ver. 1 | Jacobi Ver. 2 | GS Red-Black | GS Wavefront |
|---------|---------------|---------------|--------------|--------------|
| 1       | 0.0779        | 0.2785        | 0.3478       | 0.2103       |
| 2       | 0.1556        | 0.5554        | 0.6895       | 0.2729       |
| 4       | 0.3108        | 1.1039        | 1.3563       | 0.2729       |
| 8       | 0.6167        | 2.2001        | 2.6399       | 0.2729       |
| 12      | 0.9145        | 3.1706        | 3.7822       | 0.2727       |
| 16      | 1.2271        | 4.2403        | 4.2491       | 0.2721       |
| 24      | 1.7572        | 5.8724        | 5.6912       | 0.2715       |

Table 1: FLOPs achieved for all algorithm implementations in GFlops.

## References

- [1] Microsoft. *Visual Studio Code*. Version 1.102. 2025. URL: <https://code.visualstudio.com>.
- [2] Bernd Dammann. *02614 Assignemnt 2: The Poisson Problem*. Course handout for 02614 (41391) High Performance Computing. Jan. 2026.
- [3] Huawei Technologies Co., Ltd. *Huawei XH620 V3 Server*. 2026. URL: <https://info.support.huawei.com/computing/server3D/res/server/xh620v3/index.html?lang=en>.