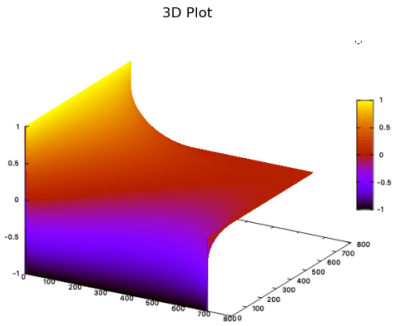
HPC Software Assignment 1

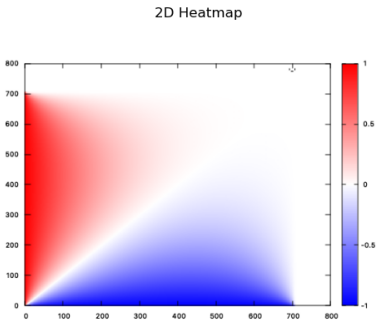
MAP-55611

Maurice Kiely - 23350601

14/03/2024

The plots are as follows:





Q1

I implemented the checkerboard pattern in 2 ways. Firstly, I simply iterated through all x and y with conditions $(x + y)\%2 == 0$. However, this proved inefficient as the large number of conditionals slowed the program.

```
double gauss_seidel_checkerboard(int n, double phi[n+1][n+1]){
    double phi_xy_old, diff=0;
    int x,y;
    // Update even points
    for (x=1;x<n;x++) {
        for (y=1;y<n;y++) {
            if ((x + y) % 2 == 0) {
                phi_xy_old = phi[x][y];
                phi[x][y] = 0.25 * (phi[x+1][y]  +  phi[x-1][y]
                   +  phi[x][y+1] +  phi[x][y-1]);
                diff += (phi_xy_old - phi[x][y]) * (phi_xy_old -
                   phi[x][y]);
            }
        }
    }
    // Update odd points
    for (x=1;x<n;x++) {
        for (y=1;y<n;y++) {
            if ((x + y) % 2 == 1) {
                phi_xy_old = phi[x][y];
                phi[x][y] = 0.25 * (phi[x+1][y]  +  phi[x-1][y]
                   +  phi[x][y+1] +  phi[x][y-1]);
                diff += (phi_xy_old - phi[x][y]) * (phi_xy_old -
                   phi[x][y]);
            }
```

```
        }
    }
    diff=sqrt(diff / (double) ((n-1)*(n-1)));
    return diff;
}
```

The other implementation of this that I did was to alter the nested loop. Instead of the condition $((x+y)\%2 == 0)$, I used the loop $(y = 1 + (x\%2); y < n; y+ = 2)$. This halves the size of loops in each iteration and dramatically speeds up the process.

```
double gauss_seidel_checkerboard(int n, double phi[n+1][n+1]){
    double phi_xy_old, diff=0;
    int x,y,p;

    // Update even points
   for (x = 1; x < n; x++) {
        for (y = 1 + (x % 2); y < n; y += 2) {
            phi_xy_old = phi[x][y];
            phi[x][y] = 0.25 * (phi[x+1][y] + phi[x-1][y] +
               phi[x][y+1] + phi[x][y-1]);
            diff += sqr(phi_xy_old - phi[x][y]);
        }
    }

    // Update odd points
    for (x = 1; x < n; x++) {
        for (y = 1 + ((x + 1) % 2); y < n; y += 2) {
            phi_xy_old = phi[x][y];
            phi[x][y] = 0.25 * (phi[x+1][y] + phi[x-1][y] +
               phi[x][y+1] + phi[x][y-1]);
            diff += sqr(phi_xy_old - phi[x][y]);
        }
    }
    diff=sqrt(diff / (double) ((n-1)*(n-1)) );
    return diff;
}
```

The code was compiled with

gcc-13 -std=c++20 -o gs gs.c -lm -fopenmp -O3

The speed of the processes were as follows:

1. Non-Checkerboard: 6.906328s
2. Inefficient Checkerboard: 6.717380s
3. Efficient Checkerboard: 4.405016s

Q2

To parallelise the code, I changed it slightly to take a command line argument specifying the number of threads and the value of n (num rows/cols). I added this argument as a parameter to the gauss-seidel function which was as follows:

```
double gauss_seidel_checkerboard(int n, double phi[n+1][n+1],
   int num_threads){
    double diff = 0.0;
    if (num_threads > n) {
        fprintf(stderr, "ERROR: Num_threads should be <= n\n");
        exit(EXIT_FAILURE);
    }
    omp_set_num_threads(num_threads);
    int block_size = n / num_threads;

    #pragma omp parallel reduction(+:diff)
    {
        int thread_id = omp_get_thread_num();
        int start_x = thread_id * block_size + 1;
        int end_x = (thread_id + 1) * block_size;
        if (thread_id == num_threads - 1) {
            end_x = n - 1; // Correct boundary for final thread
        }
        // Update even points in each block
        for (int x = start_x; x < end_x + 1; x++) {
            for (int y = 1 + (x % 2); y < n; y += 2) {
                double phi_xy_old = phi[x][y];
                phi[x][y] = 0.25 * (phi[x+1][y] + phi[x-1][y] +
                   phi[x][y+1] + phi[x][y-1]);
                diff += (phi_xy_old - phi[x][y]) * (phi_xy_old -
                   phi[x][y]);
            }
        }
        // Update odd points in each block
        for (int x = start_x; x < end_x + 1; x++) {
```

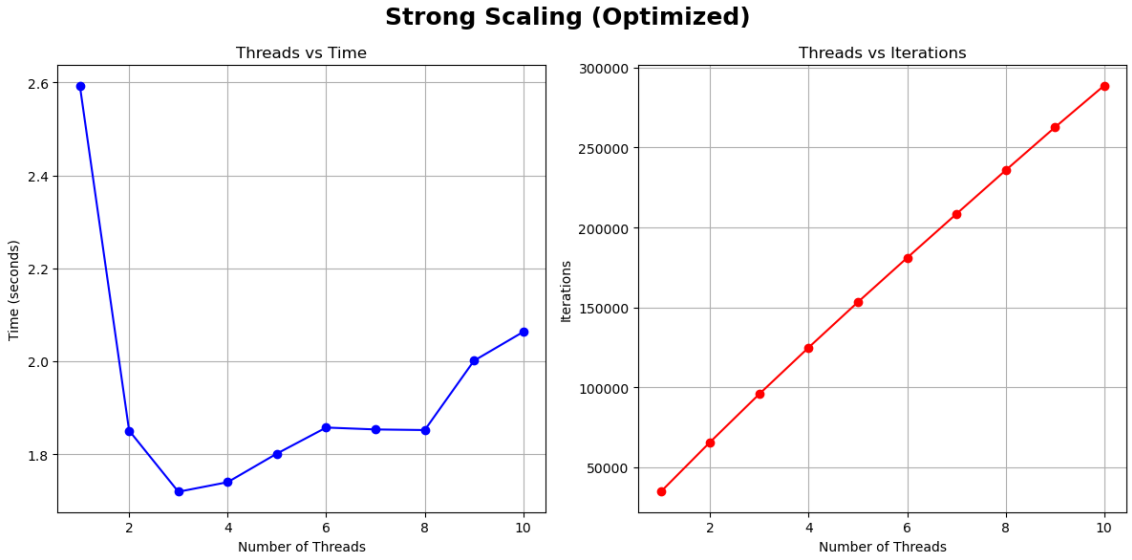
```
28              for (int y = 2 - (x % 2); y < n; y += 2) {
29                  double phi_xy_old = phi[x][y];
30                  phi[x][y] = 0.25 * (phi[x+1][y] + phi[x-1][y] +
                        phi[x][y+1] + phi[x][y-1]);
31                  diff += (phi_xy_old - phi[x][y]) * (phi_xy_old -
                        phi[x][y]);
32              }
33          }
34      }
35      diff = sqrt(diff / (double) ((n-1)*(n-1)));
36      return diff;
37  }
```

Q3

When testing all scaling, I tested on both optimised and non-optimised compiling.

Strong Scaling

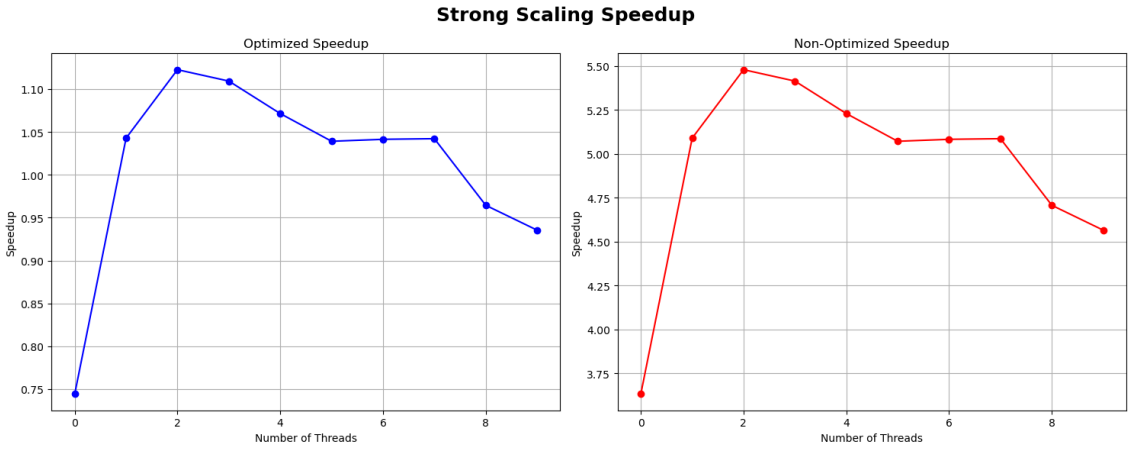
To test strong scaling, I wrote a bash script to run the code for 1 to 10 threads. This was with and without -O3 in the compiler. I also added a variable to count the number of iterations to reach the tolerance.
When plotting these we get the following:



We can see that once the number of threads goes over 4, the speed of the computation begins to slow. This is due to overheads involved in the parallelization. This is likely due to the computation time involved with creating and destroying so many threads. Another

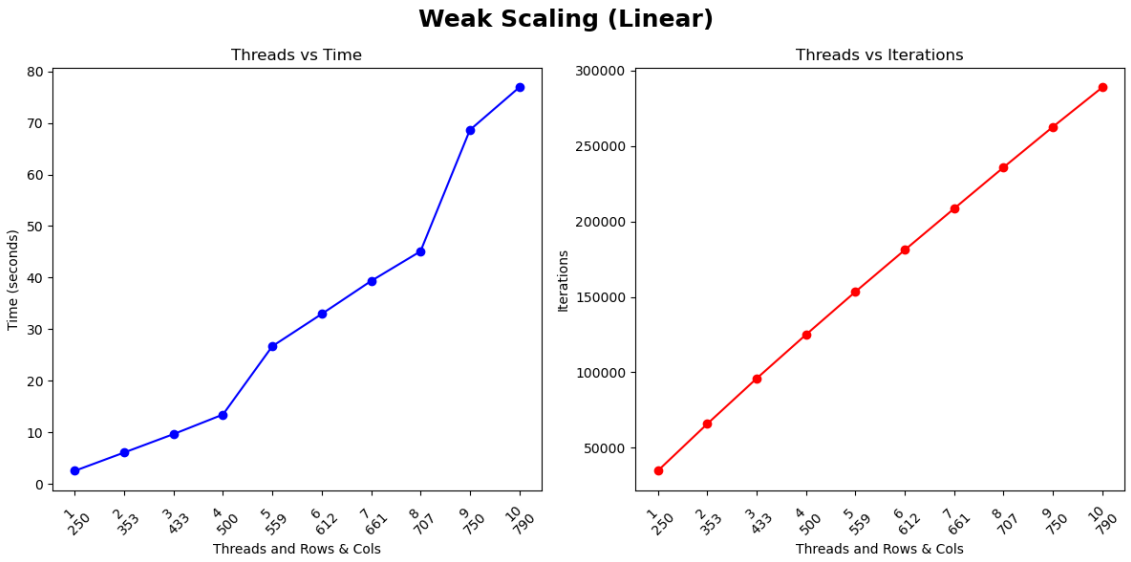
reason for the decrease in speed is due to the cache. An increase in the number of threads will increase cache misses.

The speedup can be seen a follows:



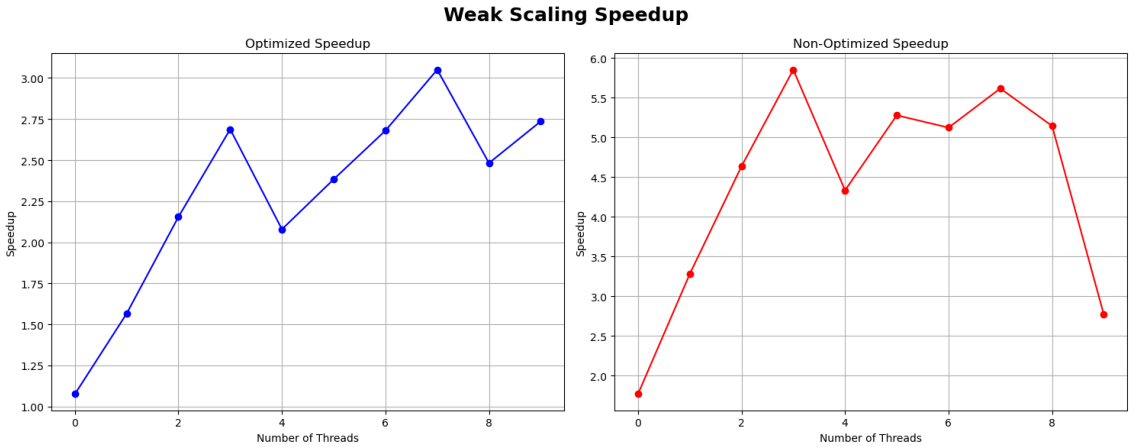
Weak Scaling

I also plotted weak scaling. Due to the size of the problem scaling with n^2, I chose $n = \sqrt{\#threads * size^2}$. This gave result as follows:



We can see from the scaling of the number of iterations, that the increase in required function calls increases linearly as expected. If we had no overheads, we would expect the speed to remain constant, however, we can see the time required increases as the number of threads increases. This indicates that as we add more threads, each additional thread becomes less effective than the last. Again, this is due to overheads and cache misses.

Again we can see the speedup below:


Weak Scaling Speedup
Optimized Speedup
Speedup
Number of Threads
Non-Optimized Speedup
Speedup
Number of Threads