# Parallel Computing Project
# Five-Point-Stencil in MPI

Jordy Aaldering, s1004292

July 22, 2020

## 1 Sequential

Before evaluating the performance of the sequential implementation I decided to optimise it to give a more realistic result when comparing it to the MPI implementation. There are two major changes; the matrix allocation and initialisation have been combined using a `calloc`[1] call, and the relaxation function keeps track of whether it is stable or not to avoid an extra loop through the matrix. These changes, as well as a few minor others, significantly sped up the sequential version.

For the evaluation a number of around 500 iterations per run seemed reasonable, this resulted in a heat of 400.0 and an epsilon of 0.05. Making the total number of iterations 445 per run for sufficiently large matrices. Matrices started out with a size of 100 by 100 and increased in steps of 100 until a size of 5000 by 5000 was reached. Each test has been repeated 10 times and then averaged to reduce the error. The performance of the sequential implementation will be described after the explanation of the MPI implementation in section 2, so that the two can be evaluated in parallel.

## 2 MPI

### 2.1 Description

The main idea of the MPI implementation is that every process gets its own rectangular block of the matrix. These blocks span the entire width of the matrix and are distributed vertically over the processes. An example of what this looks like can be seen in figure 1. To be able to relax the heat of the top and bottom rows, the process needs to have knowledge about its neighbouring processes. Therefor a row of overlap is introduced between neighbouring processes, figure 1 shows an example of this overlap for process 1. The coloured area shows the values process 1 will diffuse, and the area between the dotted lines is everything process 1 knows about, including the overlap. This overlap is possible because these values are only read from and never written to.

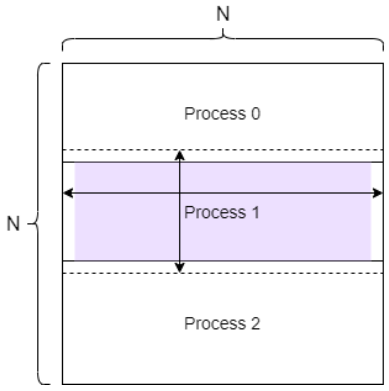



Figure 1: An example layout of processes and their blocks for a matrix of width N with three processes.

Because these blocks have to be rectangular, the matrix cannot be perfectly split up into equally sized blocks. This is solved by giving the last process the remaining part of the matrix. This is usually not a lot and therefore does not impact the performance in a meaningful way.

Whilst relaxing the matrix, each process also keeps track of whether it is stable or not. After relaxation this information is shared between all processes and as long as there is any process that is not yet stable, every process keeps relaxing and checking for stability.

[1] http://www.cplusplus.com/reference/cstdlib/calloc/

### 2.2 Implementation

In this section all important parts of the MPI implementation will be explained, along with the corresponding code snippets. This means that not the entire implementation will be shown here. The entire MPI code, as well as the optimised sequential version and evaluation processing, can be found in the attachments[2]. All C++ code can also be found in the appendices.

Before entering the main loop, every process needs to know the size of its part of the matrix. First a process gets the basic size of its block by getting its number of rows and multiplying it with the width of those rows. Then padding is added for shared rows at the top, bottom, or both. Finally, the remainder is added to the last process. The implementation of this can be seen below.

```
size_t arraySize = (n / worldSize) * n; // note the implicit flooring of integer division
if (rank != 0) {                        // there is a process above us, share top row
    arraySize += n;
}
if (rank != worldSize - 1) {            // there is a process below us, share bottom row
    arraySize += n;
} else {                                // we are the last process, add the remaining rows
    arraySize += (n % worldSize) * n;
}

return arraySize;
```

Relaxation is implemented slightly differently in the MPI implementation. Because the matrix is no longer square we first need to find out how many rows we have by dividing this block's total size by the width of the rows. Those point are then diffused. The check for stability is also done at this point. This check is relatively expensive so it is only done if the matrix is still stable.

```
bool stable = true;
for (size_t y = 1; y < arraySize / n - 1; y++) { // exclude the first and last row
    for (size_t x = 1; x < n - 1; x++) {         // exclude the first and last column
        size_t index = x + y * n;                // cast to a one-dimensional array
        Diffuse(in, out, n, index);
        if (stable && fabs(in[index] - out[index]) > eps) {
            stable = false;
        }
    }
}

return stable;
```

After relaxation all processes need to know if the matrix is stable. Every process broadcasts whether it is stable or not to every other process using an `MPI_Allreduce`[3] call, these values are then reduced using a logical AND. If this reduces to true break from the main loop, otherwise keep relaxing.

```
bool local_stable, global_stable;
while (true) {
    local_stable = Relax(in, out, n, arraySize, eps);
    MPI_Allreduce(&local_stable, &global_stable, 1, MPI_C_BOOL, MPI_LAND, MPI_COMM_WORLD);
    if (global_stable) { // only when every process is stable we can stop
        break;           // exit the main loop
    }

    ...
}
```

[2] RelaxMPI.cpp, Relax.cpp, and Evaluation.ipynb respectively, as well as Shared.h.
[3] https://www.mpich.org/static/docs/latest/www3/MPI_Allreduce.html

Now comes the most difficult part; updating the matrices with the data from a process' neighbours. This is a performance critical part of the program, so walking through all processes and updating them one by one would not suffice. Instead, all sending and receiving is done in parallel for optimal performance. Every process that has an even rank first sends and then receives, and processes with an odd rank first receive and then send. The first process never sends or receives the top row, and the last process never sends or receives the bottom row. The top and bottom row communications both have a unique tag for added safety (0 and 1 respectively). Now the benefit of the added restriction of allowing only rectangular shaped becomes clear; we know that all matrices are rectangular and thus that a row always has length N, so we do not need the status of the receive to know how many values will be received and where they should go. This both reduces the complexity of the code, and reduces the amount of processing that needs to be done in order to receive these values.

```
if (rank % 2 == 0) {
    if (rank < worldSize - 1) { // there is a process below us, share bottom row
        MPI_Send(&out[arraySize - n - 1], n, MPI_DOUBLE, rank + 1, 0, MPI_COMM_WORLD);
        MPI_Recv(&out[arraySize - n - 1], n, MPI_DOUBLE, rank + 1, 0, MPI_COMM_WORLD,
                 MPI_STATUS_IGNORE);
    }

    if (rank > 0) {             // there is a process above us, share top row
        MPI_Send(&out[0], n, MPI_DOUBLE, rank - 1, 1, MPI_COMM_WORLD);
        MPI_Recv(&out[0], n, MPI_DOUBLE, rank - 1, 1, MPI_COMM_WORLD,
                 MPI_STATUS_IGNORE);
    }
} else {
    if (rank > 0) {             // there is a process above us, share top row
        MPI_Recv(&out[0], n, MPI_DOUBLE, rank - 1, 0, MPI_COMM_WORLD,
                 MPI_STATUS_IGNORE);
        MPI_Send(&out[0], n, MPI_DOUBLE, rank - 1, 0, MPI_COMM_WORLD);
    }

    if (rank < worldSize - 1) { // there is a process below us, share bottom row
        MPI_Recv(&out[arraySize - n - 1], n, MPI_DOUBLE, rank + 1, 1, MPI_COMM_WORLD,
                 MPI_STATUS_IGNORE);
        MPI_Send(&out[arraySize - n - 1], n, MPI_DOUBLE, rank + 1, 1, MPI_COMM_WORLD);
    }
}
```

# 3 Performance

## 3.1 Hardware

All tests were run on an up to date Windows laptop with an i7-7700HQ processor[4] with 4 cores and 8 threads, running at 2.8GHz base and 3.8GHz boost. The device has 8GB of dual channel memory with a speed of 2400MHz. All code was written and compiled in C++ instead of C. The latest versions of MPI MS[5] and the MinGW[6] CPP compiler were used. All programs were compiled with the `-O2` flag set and no background tasks were running whilst evaluating the performance of the programs. The testing results of the sequential and MPI implementations can also both be found in the attachments[7].

[4] https://ark.intel.com/content/.../intel-core-i7-7700hq-processor-6m-cache-up-to-3-80-ghz.html
[5] https://docs.microsoft.com/en-us/message-passing-interface/microsoft-mpi
[6] http://www.mingw.org/
[7] relax.csv and mpi.csv respectively.

### 3.2 Speed

Figure 2 shows the speed of the different implementations. Denoted as the time it took in seconds to complete the heat diffusion with the given N. We see that the sequential and single core versions follow roughly the same line. The single core MPI implementation is expected to not be much slower due to the fact that there is not a lot of extra processing and communication introduced by this MPI implementation, especially when there is only a single core. As expected we see improved speeds when adding more cores, with diminishing returns as the number of cores increases.

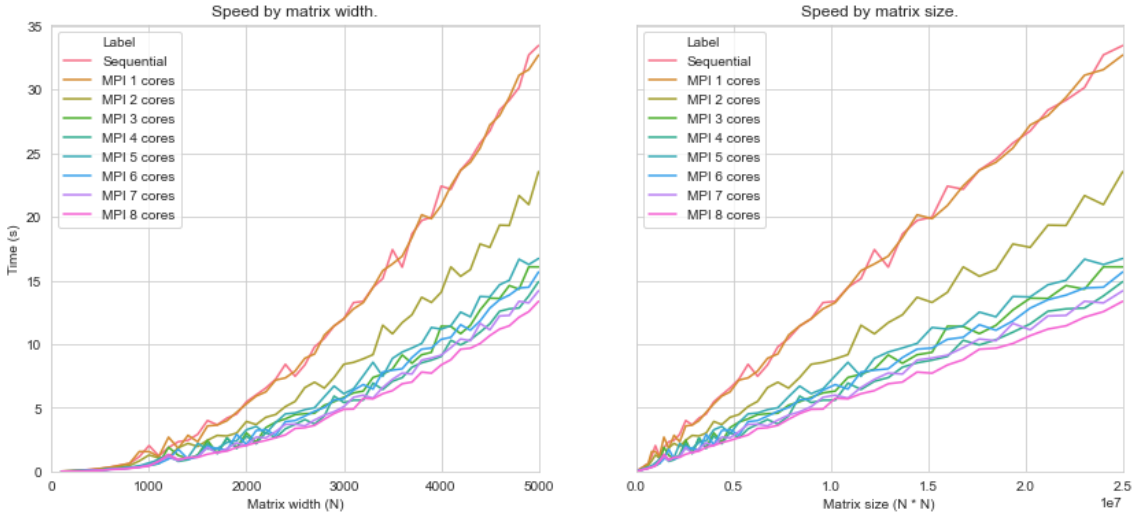



Figure 2: Speed per number of cores by matrix width and size.

### 3.3 FLOPS

An approximation of the floating point operations per second for varying matrix sizes and numbers of cores can be seen in figure 3. The sequential and single core version having roughly the same value, and the number of FLOPS increases with diminishing returns as the number of cores increases.

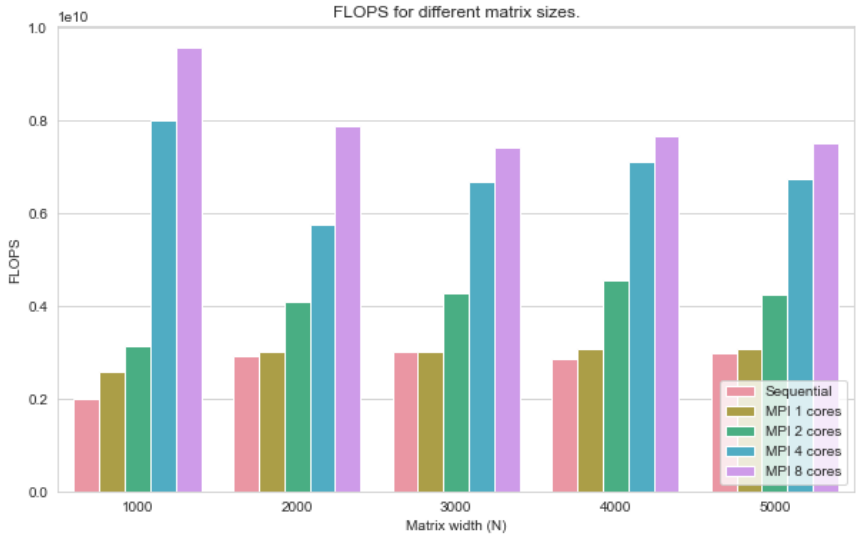



Figure 3: Floating point operations per second by matrix width for different core counts.

### 3.4 Speedup and Efficiency

The speedup and efficiency will be discussed together, as the efficiency is dependant on the speedup. Figure 4 below shows the speedup as a line plot for each number of cores with respect to the matrix width. Except for a few spikes for low matrix widths, the speedup stays roughly the same. Therefor the average can be taken, allowing this data to be plotted as a bar chart showing the average speedup for each number of cores. The same holds for the efficiency. These bar charts can be seen in figure 5a and 5b respectively. As expected the speedup for the single core version is close to 1. The speedup then steadily increases up until four cores. After that the speedup stays quite similar. Efficiency keeps decreasing up until 5 cores and then seems to stabilise around an efficiency of 0.35.

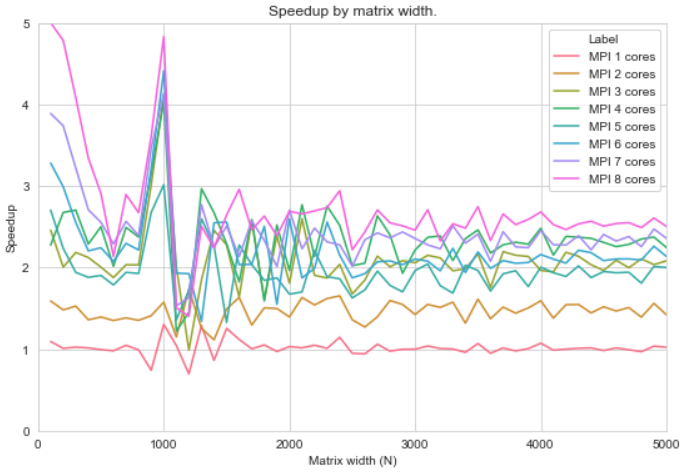
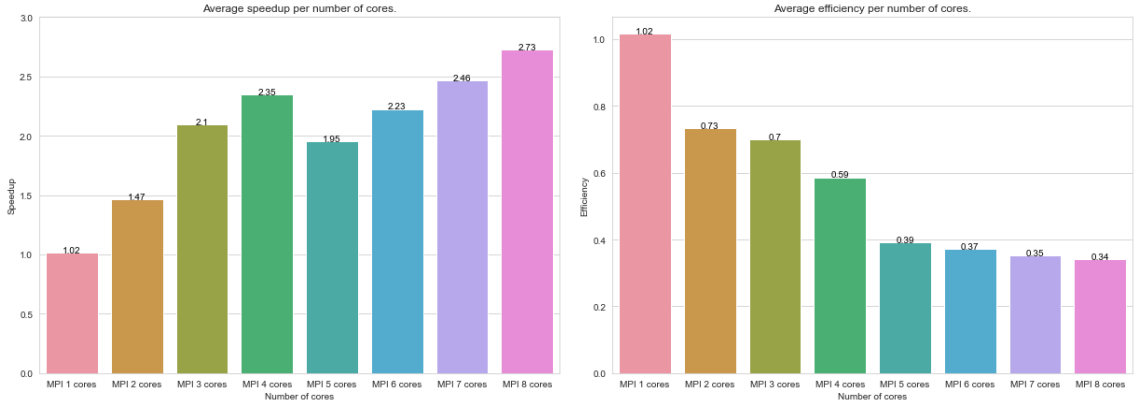



Figure 4: Speedup for each number of cores by matrix width.




(a) Average speedup per number of cores.

(b) Average efficiency per number of cores.

Figure 5: Average speedup and efficiency per number of cores.

### 3.5 Memory

One concern that might have popped up is the increased memory usage introduced by sharing parts of the matrix. With the MPI implementation the top and bottom processes both have to share one row, and all other processes have to share both their top and bottom row. This increases the total size of all matrices from $n^2$ doubles in the sequential implementation to $n^2 + 2n(\#Cores - 1)$ doubles in the MPI implementation. Since matrices are stored twice this brings the total to $2n^2 + 4n(\#Cores - 1)$ doubles. This means the space-complexity is $\Theta((n^2 + 4n(\#Cores - 1)) * 8)$, which is still $\Theta(n^2)$. Just like in the sequential implementation. This can also be seen in practise in figure 6. Therefor the increased memory usage of the MPI implementation is not a problem.

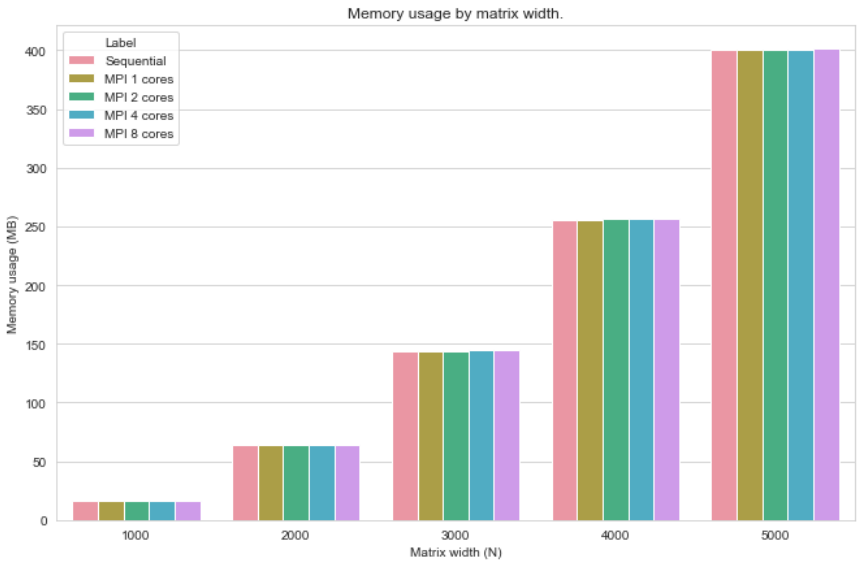



Figure 6: Memory usage by matrix width for different core counts.

## 4 Conclusion

This parallel implementation of the Five-Point-Stencil algorithm in MPI is very effective as it provides good performance improvements without sacrificing on single core performance or memory usage. The MPI code is concise and does not add a lot of overhead compared to the sequential implementation. A lot of time has gone into keeping the error low for the evaluation metrics with a total testing time of 16.32 hours, creating clean and well readable graphs.

# Appendices

## A RelaxMPI.cpp

```cpp
#include "Shared.h"
#include <mpi.h>

/// <summary>Prints information about the state of the program.</summary>
static void PrintBlock(int rank, int worldSize, int arraySize, int n, double heat, double eps, int iterations, double start, double end) {
    printf("Rank      : %d\n", rank);
    printf("World     : %d\n", worldSize);
    printf("N         : %d\n", n);
    printf("Block     : %d\n", arraySize);
    printf("Size      : %dMB\n", (int)(arraySize * arraySize * sizeof(double) / (1024 * 1024)));
    printf("Heat      : %f\n", heat);
    printf("Epsilon   : %f\n", eps);
    printf("Iterations: %d\n", iterations);
    printf("Time      : %dms\n", (int)((end - start) * 1000.0));
    printf("\n");
}

/// <summary>Calculates the size of this process' matrix.</summary>
/// <param name="rank">The rank of the current process.</param>
/// <param name="worldSize">The total number of processes.</param>
/// <param name="n">The width of the matrix.</param>
/// <returns>The size of this process' matrix.</returns>
static size_t GetArraySize(int rank, int worldSize, int n) {
    size_t arraySize = (n / worldSize) * n;
    if (rank != 0) { // share top row
        arraySize += n;
    }
    if (rank != worldSize - 1) { // share bottom row
        arraySize += n;
    } else { // add remainder
        arraySize += (n % worldSize) * n;
    }

    return arraySize;
}

/// <summary>Individual step of the 5-point stencil.</summary>
/// <param name="in">The original matrix.</param>
/// <param name="out">The resulting matrix.</param>
/// <param name="n">The width of the matrix.</param>
/// <param name="arraySize">The size of this process' matrix.</param>
/// <param name="eps">The epsilon value.</param>
/// <returns>Whether the resulting matrix is stable.</returns>
static bool Relax(double* in, double* out, size_t n, size_t arraySize, double eps) {
    bool stable = true;
    for (size_t y = 1; y < arraySize / n - 1; y++) {
        for (size_t x = 1; x < n - 1; x++) {
            size_t index = x + y * n;
            Shared::Diffuse(in, out, n, index);
            if (stable && fabs(in[index] - out[index]) > eps) {
                stable = false;
```

```
52              }
53          }
54      }
55
56      return stable;
57  }
58
59  /// <summary>Updates neighbouring processes by telling them about overlapping values with
    ↪  this process' matrix.
60  /// Then this process updates its matrix with the data received from its
    ↪  neighbours.</summary>
61  /// <param name="rank">The rank of the current process.</param>
62  /// <param name="worldSize">The total number of processes.</param>
63  /// <param name="n">The width of the matrix.</param>
64  /// <param name="arraySize">The size of this process' matrix.</param>
65  /// <param name="out">The resulting matrix.</param>
66  static void UpdateNeighbours(int rank, int worldSize, size_t n, size_t arraySize, double*
    ↪  out) {
67      if (rank % 2 == 0) {
68          if (rank < worldSize - 1) {
69              MPI_Send(&out[arraySize - n - 1], n, MPI_DOUBLE, rank + 1, 0, MPI_COMM_WORLD);
70              MPI_Recv(&out[arraySize - n - 1], n, MPI_DOUBLE, rank + 1, 0, MPI_COMM_WORLD,
                ↪  MPI_STATUS_IGNORE);
71          }
72          if (rank > 0) {
73              MPI_Send(&out[0], n, MPI_DOUBLE, rank - 1, 1, MPI_COMM_WORLD);
74              MPI_Recv(&out[0], n, MPI_DOUBLE, rank - 1, 1, MPI_COMM_WORLD,
                ↪  MPI_STATUS_IGNORE);
75          }
76      } else {
77          if (rank > 0) {
78              MPI_Recv(&out[0], n, MPI_DOUBLE, rank - 1, 0, MPI_COMM_WORLD,
                ↪  MPI_STATUS_IGNORE);
79              MPI_Send(&out[0], n, MPI_DOUBLE, rank - 1, 0, MPI_COMM_WORLD);
80          }
81          if (rank < worldSize - 1) {
82              MPI_Recv(&out[arraySize - n - 1], n, MPI_DOUBLE, rank + 1, 1, MPI_COMM_WORLD,
                ↪  MPI_STATUS_IGNORE);
83              MPI_Send(&out[arraySize - n - 1], n, MPI_DOUBLE, rank + 1, 1, MPI_COMM_WORLD);
84          }
85      }
86  }
87
88  static void Run(std::ofstream& file, size_t n, double heat, double eps) {
89      double start = MPI_Wtime();
90
91      int rank, worldSize;
92      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
93      MPI_Comm_size(MPI_COMM_WORLD, &worldSize);
94      size_t arraySize = GetArraySize(rank, worldSize, n);
95
96      int iterations = 1;
97      double* in = Shared::CreateMatrix(arraySize, rank == 0 ? n / 2 : -1, heat);
98      double* out = Shared::CreateMatrix(arraySize, rank == 0 ? n / 2 : -1, heat);
99      double* tmp;
100
101     bool local_stable, global_stable;
102     while (true) {
```

```
103         local_stable = Relax(in, out, n, arraySize, eps);
104         MPI_Allreduce(&local_stable, &global_stable, 1, MPI_C_BOOL, MPI_LAND,
            ↪  MPI_COMM_WORLD);
105         if (global_stable) { // only when every process is stable we can stop
106             break;
107         }
108 
109         UpdateNeighbours(rank, worldSize, n, arraySize, out);
110 
111         tmp = in;
112         in = out;
113         out = tmp;
114         iterations++;
115     }
116 
117     double end = MPI_Wtime();
118     free(in);
119     free(out);
120 
121     Shared::WriteInfo(file, n, iterations, (int)((end - start) * 1000.0), worldSize);
122     PrintBlock(rank, worldSize, arraySize, n, heat, eps, iterations, start, end);
123 }
124 
125 int main(int argc, char** argv) {
126     std::ofstream file = Shared::OpenFile("mpi");
127     MPI_Init(&argc, &argv);
128 
129     for (int i = 1; i <= STEPS; i++) {
130         for (int r = 0; r < REPEATS; r++) {
131             Run(file, i * N, HEAT, EPS);
132             MPI_Barrier(MPI_COMM_WORLD);
133         }
134     }
135 
136     MPI_Finalize();
137     file.close();
138     return 0;
139 }
```

## B Relax.cpp

```cpp
#include "Shared.h"
#include <time.h>

/// <summary>Prints information about the state of the program.</summary>
static void PrintMatrix(int n, double heat, double eps, int iterations, clock_t start, clock_t end) {
    printf("N          : %d\n", n);
    printf("Size       : %dMB\n", (int)(n * n * sizeof(double) / (1024 * 1024)));
    printf("Heat       : %f\n", heat);
    printf("Epsilon    : %f\n", eps);
    printf("Iterations: %d\n", iterations);
    printf("Time       : %dms\n", (int)((end - start) / (CLOCKS_PER_SEC / 1000.0)));
    printf("\n");
}

/// <summary>Individual step of the 5-point stencil.</summary>
/// <param name="in">The original matrix.</param>
/// <param name="out">The resulting matrix.</param>
/// <param name="n">The width of the matrix.</param>
/// <param name="eps">The epsilon value.</param>
/// <returns>Whether the resulting matrix is stable.</returns>
static bool Relax(double* in, double* out, size_t n, double eps) {
    bool stable = true;
    for (size_t y = 1; y < n - 1; y++) {
        for (size_t x = 1; x < n - 1; x++) {
            size_t index = x + y * n;
            Shared::Diffuse(in, out, n, index);
            if (stable && fabs(in[index] - out[index]) > eps) {
                stable = false;
            }
        }
    }

    return stable;
}

static void Run(std::ofstream& file, size_t n, double heat, double eps) {
    clock_t start = clock();

    int iterations = 1;
    double* in = Shared::CreateMatrix(n * n, n / 2, heat);
    double* out = Shared::CreateMatrix(n * n, n / 2, heat);
    double* tmp;

    while (!Relax(in, out, n, eps)) {
        tmp = in;
        in = out;
        out = tmp;
        iterations++;
    }

    clock_t end = clock();
    free(in);
    free(out);

```

```
55      Shared::WriteInfo(file, n, iterations, (int)((end - start) / (CLOCKS_PER_SEC /
        ↪  1000.0)));
56      PrintMatrix(n, heat, eps, iterations, start, end);
57  }
58
59  int main() {
60      std::ofstream file = Shared::OpenFile("relax");
61
62      for (int i = 1; i <= STEPS; i++) {
63          for (int r = 0; r < REPEATS; r++) {
64              Run(file, i * N, HEAT, EPS);
65          }
66      }
67
68      file.close();
69      return 0;
70  }
```

## C Shared.h

```
#pragma once

#include <stdio.h>
#include <fstream>
#include <math.h>
#include <memory>

#define N 100
#define HEAT 400.0
#define EPS 0.05

#define STEPS 50
#define REPEATS 10

class Shared {
public:
    /// <summary>Allocate a flattened matrix and initialise the values.</summary>
    /// <param name="size">The total size of the matrix.</param>
    /// <param name="heatIndex">At what index to place the heat, -1 if no heat should be added.</param>
    /// <param name="heat">The heat value to place.</param>
    /// <returns>A new matrix with the given size and potentially a heat value.</returns>
    inline static double* CreateMatrix(size_t size, int heatIndex = -1, double heat = HEAT) {
        double* m = (double*)calloc(size, sizeof(double));
        if (heatIndex >= 0) {
            m[heatIndex] = heat;
        }
        return m;
    }

    /// <summary>Diffuses a point using neighbouring values.</summary>
    /// <param name="in">The original matrix.</param>
    /// <param name="out">The resulting matrix.</param>
    /// <param name="n">The width of the matrix.</param>
    /// <param name="i">The point to diffuse.</param>
    inline static void Diffuse(double* in, double* out, size_t n, size_t i) {
        out[i] = 0.25 * in[i]     // center
            + 0.250 * in[i - n]  // upper
            + 0.125 * in[i + n]  // lower
            + 0.175 * in[i - 1]  // left
            + 0.200 * in[i + 1]; // right
    }

    /// <summary>Opens a csv file and returns it.</summary>
    /// <param name="filename">The name of the file to open.</param>
    /// <returns>The opened file.</returns>
    inline static std::ofstream OpenFile(const std::string filename) {
        std::ofstream file;
        std::string path = "Evaluation/" + filename + ".csv";
        file.open(path, std::ios_base::app);

        if (!file.is_open()) {
            printf("Could not open file '%s.csv'.", filename);
            exit(1);
        }
```

```
55
56          return file;
57      }
58
59      /// <summary>Write information about the state of the program.</summary>
60      inline static void WriteInfo(std::ofstream& file, int n, int iterations, int ms, int
        ↪  cores = -1) {
61          if (cores > 0) {
62              file << cores << ",";
63          }
64
65          file << n << ","
66               << (int)(n * n * sizeof(double) / (1024 * 1024)) << ","
67               << iterations << ","
68               << ms << std::endl;
69      }
70  };
```