High Performance Computing Proseminar 2024
Assignment 2

Stefan Wagner & Sebastian Bergner
Team: Planning Desaster

October 13, 2024

Exercise 1

This exercise consists in writing a parallel application to speed up the computation of π.

There are many ways of approximating π, one being a well-known Monte Carlo method: The ratio of the areas of a square and its incircle is $\pi/4$. Since the exact area of a circle cannot be computed (we don't know the value of π yet), one can instead sample random points, check their distance from the center and compute the ratio of points inside the circle to all sampled points.

- Write a sequential application 'pi_seq' in C or C++ that computes π for a given number of samples (command line argument). Test your application for various, large sample sizes to verify the correctness of your implementation.

  The main part of the monte carlo pi approximation is shown below. The code works as follows: generate S random points inside a squared area, if they lie inside the unit circle the length of the vector they build is ≤ 1 thus we can increase the counter. Otherwise we just ignore it. In the end we have to multipy by 4 to accomodate the fact that we only generated random numbers in the right upper quadrant (no negative numbers).

```
double mc_pi(unsigned int S) {
  int in_count = 0;
  for(unsigned i = 0; i < S; ++i) {
    const double x = rand() / (double)RAND_MAX;
    const double y = rand() / (double)RAND_MAX;
    if(x * x + y * y <= 1.f) {
      in_count++;
    }
  }
  return 4.f * in_count / S;
}
```

- Consider a parallelization strategy using MPI. Which communication pattern(s) would you choose and why?

  As we actually don't have to accommodate relevant data that has to be distributed we can utilize the mpi reduce functionality. This should allow for least communication overhead. Each process gets its own share to work on the problem and initializes a random seed based on the rank it has. And after computing the elements inside the quarter circle we sum it up using reduce and then do the last step of the above function only in the master node (rank = 0).

- Implement your chosen parallelization strategy as a second application 'pi_mpi'. Run it with varying numbers of ranks and sample sizes and verify its correctness by comparing the output to 'pi_seq'.

  In the below code section the main parts of the code are shown (boilerplate code has been omited).

Problem size 10^N	Sequential Time [s]	Parallel Time [s]
2	0.0000366	0.0954576
3	0.0000624	0.0954746
4	0.0003084	0.0944174
5	0.0025352	0.0947864
6	0.0257526	0.0957872
7	0.2516176	0.100753
8	2.4850934	0.1531918
9	24.8637024	0.5713218

```
int mc_pi(int S) {
    int in_count = 0;
    for(unsigned i = 0; i < S; ++i) {
        const double x = rand() / (double)RAND_MAX;
        const double y = rand() / (double)RAND_MAX;
        if(x * x + y * y <= 1.f) {
            in_count++;
        }
    }
    return in_count; // only return the number of elements inside
}

int main(int argc, char* argv[]) {
        // .. some boilerplate code left out
  MPI_Init(&argc, &argv); //start mpi
  // let every process work on their problem (we won't instruct them from the root node 0 only gather ↩
    the data)
  int myRank, numProcesses;
  int insideLocal = 0;
  MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
  MPI_Comm_size(MPI_COMM_WORLD, &numProcesses);

  srand(time(NULL)*myRank);

  int calculations_per_process = N/numProcesses;
  insideLocal = mc_pi(calculations_per_process);

  // here https://mpitutorial.com/tutorials/mpi-reduce-and-allreduce/ was quite helpful
  int insideGlobal = 0;
  MPI_Reduce(&insideLocal, &insideGlobal, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

  if (myRank == 0){ // let only one rank do the last step
    double result = 4.f * insideGlobal / (N-(N%numProcesses)); // correction if N is not fully divisible↩
     by numProcesses
    // ... some more boilerplate code
  }
  MPI_Finalize();
  return 0;
}
```

Problem size 10^N	serial result	parallel result
2	3.12	3.04
3	3.132	3.032
4	3.1712	3.1564
5	3.141520	3.1392
6	3.141664	3.143120
7	3.141130	3.142680
8	3.141698	3.141692
9	3.141603	3.141559

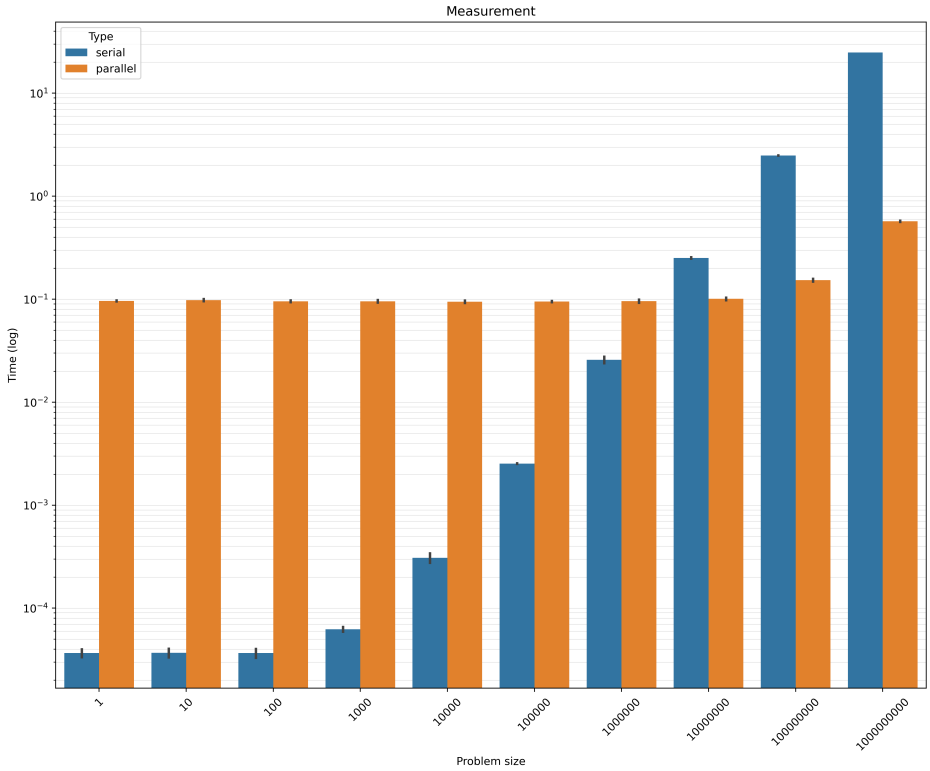

Figure 1: Monte Carlo Pi Approximation Measurements.

- Discuss the effects and implications of your parallelization.

  This rather simple parallelization is able to offer an speedup of ≈ 43.6 at the largest problem size we've measured. But looking at small problem sizes, the parallelized version has a rather high overhead of at least $\approx 95ms$ (compared to an execution time of $41\mu s$ at 100 points). This can be seen in Fig. 1. So if we would want to have the best overall performing version we could find the limit where the parallel version performs better and switch between these two.

- Insert the measured wall time for 10^9 samples for the sequential implementation and on 96 cores for MPI into the comparison spreadsheet

Exercise 2

This exercise consists in parallelizing an application simulating the propagation of heat. A large class of scientific applications are so-called stencil applications. These simulate time-dependent physical processes such as the propagation of heat or pressure in a given medium. The core of the simulation operates on a grid and updates each cell with information from its neighbor cells.

- A sequential implementation of a 1-D heat stencil is available in heat_stencil_1D_seq.c. Read the code and make sure you understand what happens. See the Wikipedia article on Stencil Codes for more information.
- Consider a parallelization strategy using MPI. Which communication pattern(s) would you choose and why? Are there additional changes required in the code beyond calling MPI functions? If so, elaborate!

a) Complex Parallelization Strategy
First, we considered a rather complex communication pattern, as shown in Figure 2.

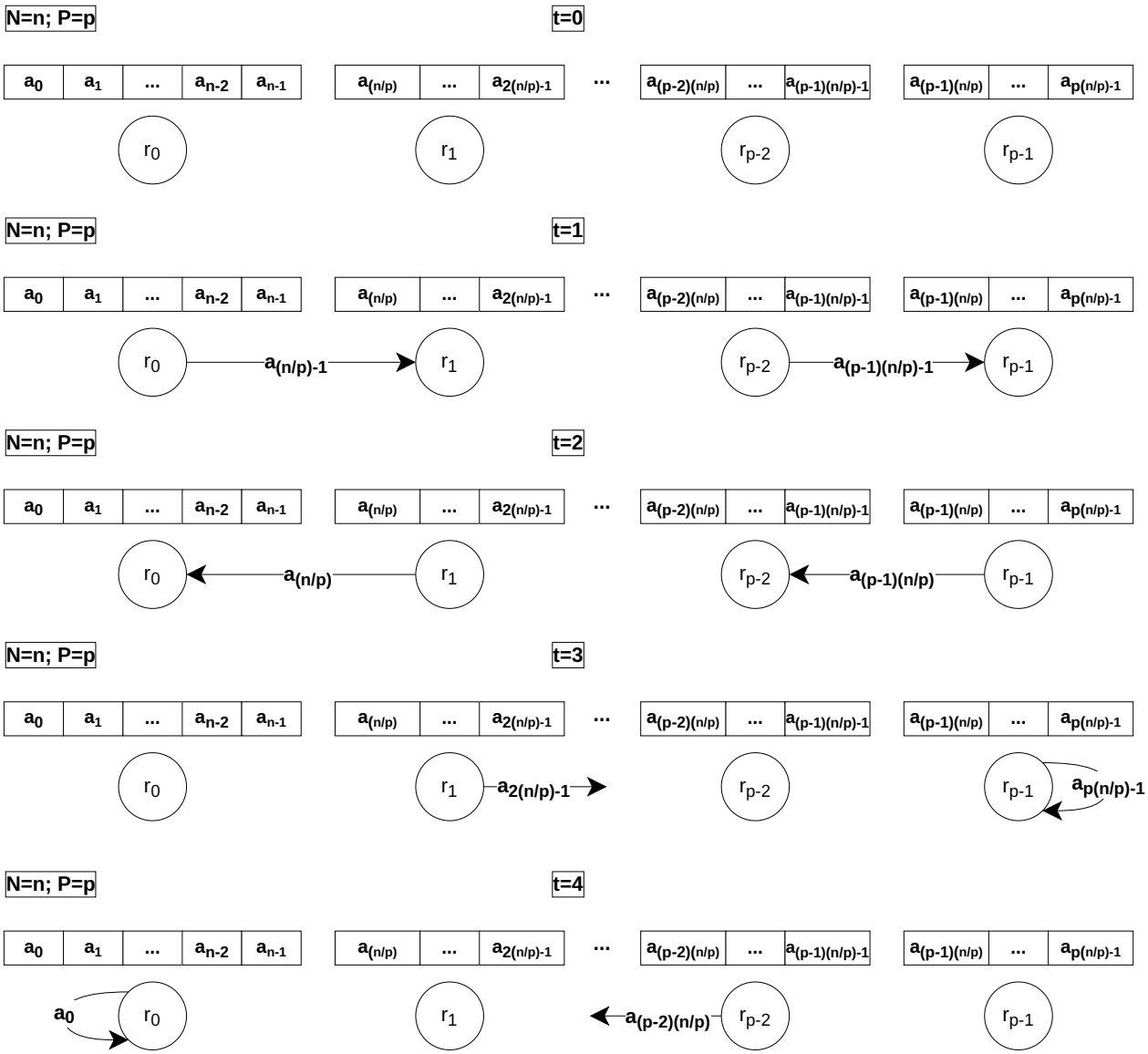

Figure 2: Complex Parallelization Strategy

The communication pattern is **blocking** and as follows:

1) The total number of elements N is distributed across P vectors V_p, one vector per rank/process. Only rank r_0 holds a vector of length N since it handles tasks like printing intermediate results. However, during computation, r_0 operates only on its own portion of the elements.

2) At $t = 1$, all even-numbered ranks send their **last** element (except for r_0, where it is not the last) to their **right** neighbor, if one exists, while all odd-numbered ranks receive the element from their **left** neighbors.

3) At $t = 2$, all odd-numbered ranks send their **first** element to their **left** neighbor, while all even-numbered ranks receive the element from their **right** neighbors, if one exists.

4) At $t = 3$, all odd-numbered ranks send their **last** element to their **right** neighbor, while all even-numbered ranks receive the element from their **left** neighbors, if one exists.

5) At $t = 4$, all even-numbered ranks send their **first** element to their **left** neighbor, while all odd-numbered ranks receive the element from their **right** neighbors, if one exists.

6) Now that all ranks have received the necessary information from their neighbors, they can begin their heat propagation computation.

7) If the **first** rank attempts to send to a non-existent **left** neighbor, or the **last** rank to a non-existent **right** neighbor, no element is sent. Instead, the receive buffer is populated with the first or last entry of vector V, respectively.

8) To print an intermediate or final result, all ranks send their portion of the elements to rank r_0, which holds a vector of size N, using `MPI_Gather`. Rank r_0 then prints the complete result.

– **Why did we choose this communication pattern?**
To parallelize the computation effectively, we need to distribute the problem size across all processes. However, due to the nature of the heat propagation calculation, each process requires the first and last elements from neighboring processes. To exchange this information while minimizing communication overhead (e.g. avoiding `MPI_Scatter`), we focused solely on handling these edge cases.

b) Simple Parallelization Strategy
After implementing the initial strategy and comparing the results with the data from the comparison spreadsheet, we decided to simplify the communication pattern, as illustrated in Figure 3.

The simpler communication pattern is now **non-blocking** and as follows:

1) The total number of elements N is distributed across P vectors V_p, one vector per rank/process. Only rank r_0 holds a vector of length N since it handles tasks like printing intermediate results. However, during computation, r_0 operates only on its own portion of the elements.

2) At $t = 1$, **all** ranks first send their **first** and **last** elements (except for r_0, where it is not the last) to their **left** and **right** neighbors, if they exist, using the non-blocking `Isend()` call. Then, **all** ranks use the non-blocking `Irecv()` function to receive the **first** and **last** elements from their **right** and **left** neighbors, respectively.

3) At $t = 2$, **all** ranks wait for the completion of their `Isend()` and `Irecv()` operations before starting the heat propagation computation.

4) If the **first** rank attempts to send to a non-existent **left** neighbor, or the **last** rank to a non-existent **right** neighbor, no element is sent. Instead, the receive buffer is populated with the first or last entry of vector V, respectively.

5) To print an intermediate or final result, all ranks send their portion of the elements to rank r_0, which holds a vector of size N, using `MPI_Gather`. Rank r_0 then prints the complete result.

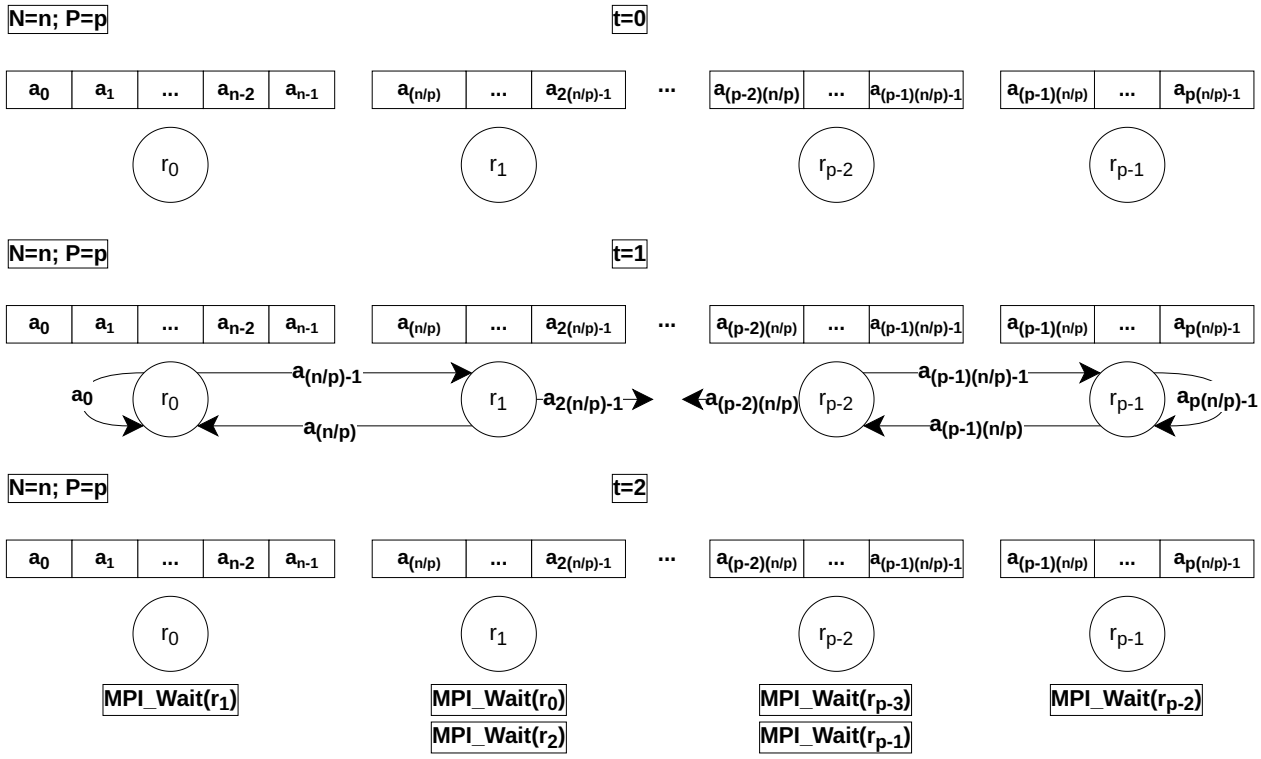

Figure 3: Simple Parallelization Strategy

- Implement your chosen parallelization strategy as a second application 'heat_stencil_1D_mpi'. Run it with varying numbers of ranks and problem sizes and verify its correctness by comparing the output to 'heat_stencil_1D_seq'.

1) heat_stencil_1D_par_complex.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <unistd.h>
#include <sys/stat.h>
#include <mpi.h>

typedef double value_t;
#define RESOLUTION 120

// -- vector utilities --
typedef value_t *Vector;
Vector createVector(int N);
void releaseVector(Vector m);
void printTemperature(Vector m, int N);

// -- measurment utilities --
#define FOLDER "output"
#define FILENAME "measurements.csv"
void timings_to_csv(unsigned problem_size, double time, int numRanks);

// -- simulation code ---
int main(int argc, char **argv) {
  clock_t start = clock();

  if(argc < 2){
    printf("Usage: %s <number of iterations>\n", argv[0]);
```

```
    return EXIT_FAILURE;
  }

  int N = atoi(argv[1]);
  int T = N * 500;

  MPI_Init(&argc, &argv);
  int myRank, numRanks;
  MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
  MPI_Comm_size(MPI_COMM_WORLD, &numRanks);

  if (N % numRanks) {
    printf("Configuration not possible: N=%d, ranks=%d\n", N, numRanks);
    MPI_Finalize();
    return EXIT_FAILURE;
  }

  if (myRank == 0) {
    printf("Computing heat-distribution for room size N=%d for T=%d timesteps\n", N, T);
  }

  // ---------- setup ----------

  // create a buffer for storing temperature fields per rank
  int vector_size_per_rank = N / numRanks;
  Vector A = NULL;
  Vector B = NULL; // create a second buffer for the computation
  if (myRank == 0) {
    A = createVector(N);
    B = createVector(N);
  } else {
    A = createVector(vector_size_per_rank);
    B = createVector(vector_size_per_rank);
  }

  // set up initial conditions in A
  for (int i = 0; i < vector_size_per_rank; i++) {
    A[i] = 273; // temperature is 0 C everywhere (273 K)
  }

  // and there is a heat source somewhere
  int source_x = N / 4;
  int source_y = 273 + 60;

  int rank_with_source = source_x / vector_size_per_rank;
  if (myRank == 0) {
    A[source_x] = source_y;
  }
  if (myRank == rank_with_source) {
    A[source_x % vector_size_per_rank] = source_y;
  }

  if (myRank == 0) {
    printf("Initial:\t");
    printTemperature(A, N);
    printf("\n");
  }

  // ---------- compute ----------
  value_t t_from_previous_rank = 0;
  value_t t_from_next_rank = 0;

  // for each time step ..
  for (int t = 0; t < T; t++) {
    // communication between ranks to get temperatures of adjacent cells
    if (myRank % 2 == 0) {
      if (myRank != numRanks-1) {
        // send last temperature to next rank
```

```
97          MPI_Send(&A[vector_size_per_rank-1], 1, MPI_DOUBLE, myRank+1, 0, MPI_COMM_WORLD);
98          // receive first temperature from next rank
99          MPI_Recv(&t_from_next_rank, 1, MPI_DOUBLE, myRank+1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
100       } else {
101         t_from_next_rank = A[vector_size_per_rank-1];
102       }
103
104       if (myRank != 0) {
105         // receive last temperature from previous rank
106         MPI_Recv(&t_from_previous_rank, 1, MPI_DOUBLE, myRank-1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
107         // send first temperature to previous rank
108         MPI_Send(&A[0], 1, MPI_DOUBLE, myRank-1, 0, MPI_COMM_WORLD);
109       } else {
110         t_from_previous_rank = A[0];
111       }
112     } else {
113       // receive last temperature from previous rank
114       MPI_Recv(&t_from_previous_rank, 1, MPI_DOUBLE, myRank-1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
115       // send first temperature to previous rank
116       MPI_Send(&A[0], 1, MPI_DOUBLE, myRank-1, 0, MPI_COMM_WORLD);
117
118       if (myRank != numRanks-1) {
119         // send last temperature to next rank
120         MPI_Send(&A[vector_size_per_rank-1], 1, MPI_DOUBLE, myRank+1, 0, MPI_COMM_WORLD);
121         // receive first temperature from next rank
122         MPI_Recv(&t_from_next_rank, 1, MPI_DOUBLE, myRank+1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
123       } else {
124         t_from_next_rank = A[vector_size_per_rank-1];
125       }
126     }
127
128     // .. we propagate the temperature
129     for (long long i = 0; i < vector_size_per_rank; i++) {
130       // center stays constant (the heat is still on)
131       if (myRank == rank_with_source && i == (source_x % vector_size_per_rank)) {
132         B[i] = A[i];
133         continue;
134       }
135
136       // get temperature at current position
137       value_t tc = A[i];
138
139       // get temperatures of adjacent cells
140       value_t tl = (i != 0) ? A[i - 1] : t_from_previous_rank;
141       value_t tr = (i != vector_size_per_rank - 1) ? A[i + 1] : t_from_next_rank;
142
143       // compute new temperature at current position
144       B[i] = tc + 0.2 * (tl + tr + (-2 * tc));
145     }
146
147     // swap matrices (just pointers, not content)
148     Vector H = A;
149     A = B;
150     B = H;
151
152     // show intermediate step
153     if (!(t % 10000)) {
154       // Gather all data from all ranks to rank 0
155       MPI_Gather(A, vector_size_per_rank, MPI_DOUBLE, A, vector_size_per_rank, MPI_DOUBLE, 0, ↩
    MPI_COMM_WORLD);
156       if (myRank == 0) {
157         printf("Step t=%d:\t", t);
158         printTemperature(A, N);
159         printf("\n");
160       }
161     }
162   }
163
```

```
164   releaseVector(B);
165
166   MPI_Gather(A, vector_size_per_rank, MPI_DOUBLE, A, vector_size_per_rank, MPI_DOUBLE, 0, ↩
        MPI_COMM_WORLD);
167   int success = 1;
168
169   if (myRank == 0) {
170     // measure time
171     clock_t end = clock();
172     double total_time = ((double)(end - start)) / CLOCKS_PER_SEC;
173     timings_to_csv(N, total_time, numRanks);
174
175     // ---------- check ----------
176     printf("Final:\t\t");
177     printTemperature(A, N);
178     printf("\n");
179
180     int success = 1;
181     for (long long i = 0; i < N; i++) {
182       value_t temp = A[i];
183       if (273 <= temp && temp <= 273 + 60)
184         continue;
185       success = 0;
186       break;
187     }
188
189     printf("Verification: %s\n", (success) ? "OK" : "FAILED");
190     printf("Wall Clock Time = %f seconds\n", total_time);
191   }
192
193   // ---------- cleanup ----------
194   releaseVector(A);
195   MPI_Finalize();
196
197   // done
198   return (success) ? EXIT_SUCCESS : EXIT_FAILURE;
199 }
200
201 Vector createVector(int N) {
202   // create data and index vector
203   return malloc(sizeof(value_t) * N);
204 }
205
206 void releaseVector(Vector m) { free(m); }
207
208 void printTemperature(Vector m, int N) {
209   const char *colors = " .-:=+*^X#%@";
210   const int numColors = 12;
211
212   // boundaries for temperature (for simplicity hard-coded)
213   const value_t max = 273 + 60;
214   const value_t min = 273 + 0;
215
216   // set the 'render' resolution
217   int W = RESOLUTION;
218
219   // step size in each dimension
220   int sW = N / W;
221
222   // room
223   // left wall
224   printf("X");
225   // actual room
226   for (int i = 0; i < W; i++) {
227     // get max temperature in this tile
228     value_t max_t = 0;
229     for (int x = sW * i; x < sW * i + sW; x++) {
230       max_t = (max_t < m[x]) ? m[x] : max_t;
```

```
    }
    value_t temp = max_t;

    // pick the 'color'
    int c = ((temp - min) / (max - min)) * numColors;
    c = (c >= numColors) ? numColors - 1 : ((c < 0) ? 0 : c);

    // print the average temperature
    printf("%c", colors[c]);
  }
  // right wall
  printf("X");
}

void timings_to_csv(unsigned problem_size, double time, int numRanks) {
  FILE* fpt;
  int set_header = 0;
  char full_filepath[1024];
  sprintf(full_filepath, "%s/%s", FOLDER, FILENAME);
  if(access(FOLDER, F_OK) != 0) mkdir(FOLDER, 0755);
  if(access(full_filepath, F_OK) != 0) set_header = 1;
  fpt = fopen(full_filepath, "a+");
  if(set_header) fprintf(fpt, "Impl/Ranks,Problem Size,Time\n");
  fprintf(fpt, "par_complex/%d,%u,%.9f\n", numRanks, problem_size, time);
  fclose(fpt);
}
```

2) heat_stencil_1D_par_simple.c

```
int main(int argc, char **argv) {
  clock_t start = clock();

  if(argc < 2){
    printf("Usage: %s <number of iterations>\n", argv[0]);
    return EXIT_FAILURE;
  }

  int N = atoi(argv[1]);
  int T = N * 500;

  MPI_Init(&argc, &argv);
  int myRank, numRanks;
  MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
  MPI_Comm_size(MPI_COMM_WORLD, &numRanks);

  if (N % numRanks) {
    if (myRank == 0) {
      printf("Configuration not possible: N=%d, ranks=%d\n", N, numRanks);
    }
    MPI_Finalize();
    return EXIT_FAILURE;
  }
  if (myRank == 0) {
    printf("Computing heat-distribution for room size N=%d for T=%d timesteps\n", N, T);
  }

  // ---------- setup ----------

  // create a buffer for storing temperature fields per rank
  int vector_size_per_rank = N / numRanks;
  Vector A = NULL;
  Vector B = NULL;
  if (myRank == 0) {
    A = createVector(N);
    B = createVector(N);
  } else {
    A = createVector(vector_size_per_rank);
    B = createVector(vector_size_per_rank);
  }
```

```
41
42  // set up initial conditions in A
43  for (int i = 0; i < vector_size_per_rank; i++) {
44    A[i] = 273; // temperature is 0 C everywhere (273 K)
45  }
46
47  // and there is a heat source somewhere
48  int source_x = N / 4;
49  int source_y = 273 + 60;
50
51  int rank_with_source = source_x / vector_size_per_rank;
52  if (myRank == 0) {
53    A[source_x] = source_y;
54  }
55  if (myRank == rank_with_source) {
56    A[source_x % vector_size_per_rank] = source_y;
57  }
58
59  if (myRank == 0) {
60    printf("Initial:\t");
61    printTemperature(A, N);
62    printf("\n");
63  }
64  MPI_Request requests[4];
65  // ---------- compute ----------
66
67  // for each time step ..
68  for (int t = 0; t < T; t++) {
69    // sync with other ranks
70    value_t t_left, t_right;
71    if (myRank != 0 && myRank != numRanks-1){ //consider edge cases separatelly
72      MPI_Isend(&A[vector_size_per_rank-1], 1, MPI_DOUBLE, myRank+1, 0, MPI_COMM_WORLD, &requests[0]);
73      MPI_Isend(&A[0], 1, MPI_DOUBLE, myRank-1, 0, MPI_COMM_WORLD, &requests[1]);
74
75      MPI_Irecv(&t_left, 1, MPI_DOUBLE, myRank-1, 0, MPI_COMM_WORLD, &requests[2]);
76      MPI_Irecv(&t_right, 1, MPI_DOUBLE, myRank+1, 0, MPI_COMM_WORLD, &requests[3]);
77
78      MPI_Wait(&requests[2], MPI_STATUS_IGNORE);
79      MPI_Wait(&requests[3], MPI_STATUS_IGNORE);
80    }
81    else if(myRank == 0){ // edge case 1 first rank
82      MPI_Isend(&A[vector_size_per_rank-1], 1, MPI_DOUBLE, myRank+1, 0, MPI_COMM_WORLD, &requests[0]);
83      MPI_Irecv(&t_right, 1, MPI_DOUBLE, myRank+1, 0, MPI_COMM_WORLD, &requests[1]);
84      t_left = A[0];
85    }
86    else{ // edge case 1 last rank
87      MPI_Isend(&A[0], 1, MPI_DOUBLE, myRank-1, 0, MPI_COMM_WORLD, &requests[0]);
88      MPI_Irecv(&t_left, 1, MPI_DOUBLE, myRank-1, 0, MPI_COMM_WORLD, &requests[1]);
89      t_right = A[vector_size_per_rank-1];
90    }
91    // pre-calc syncing done
92    MPI_Wait(&requests[0], MPI_STATUS_IGNORE);
93    MPI_Wait(&requests[1], MPI_STATUS_IGNORE);
94
95
96    // .. we propagate the temperature
97    for (long long i = 0; i < vector_size_per_rank; i++) {
98      // center stays constant (the heat is still on)
99      if (myRank == rank_with_source && i == (source_x%vector_size_per_rank)) {
100       B[i] = A[i];
101       continue;
102     }
103
104     // get temperature at current position
105     value_t tc = A[i];
106
107     // get temperatures of adjacent cells
108     value_t tl = (i != 0) ? A[i - 1] : t_left;
```

```
109       value_t tr = (i != vector_size_per_rank - 1) ? A[i + 1] : t_right;
110
111       // compute new temperature at current position
112       B[i] = tc + 0.2 * (tl + tr + (-2 * tc));
113     }
114
115     // swap matrices (just pointers, not content)
116     Vector H = A;
117     A = B;
118     B = H;
119
120     // show intermediate step
121     if (!(t % 10000)) {
122       // now we have to gather all data in rank 0
123       MPI_Gather(A, vector_size_per_rank, MPI_DOUBLE, A, vector_size_per_rank, MPI_DOUBLE, 0, ↩
      MPI_COMM_WORLD);
124
125       if (myRank == 0){
126         printf("Step t=%d:\t", t);
127         printTemperature(A, N);
128         printf("\n");
129       }
130     }
131   }
132
133   releaseVector(B);
134   int success = 1;
135   // last sync
136   MPI_Gather(A, vector_size_per_rank, MPI_DOUBLE, A, vector_size_per_rank, MPI_DOUBLE, 0, ↩
        MPI_COMM_WORLD);
137   if (myRank == 0){
138     // measure time
139     clock_t end = clock();
140     double total_time = ((double)(end - start)) / CLOCKS_PER_SEC;
141     timings_to_csv(N, total_time, numRanks);
142
143     // ---------- check ----------
144
145     printf("Final:\t\t");
146     printTemperature(A, N);
147     printf("\n");
148
149
150     for (long long i = 0; i < N; i++) {
151       value_t temp = A[i];
152       if (273 <= temp && temp <= 273 + 60)
153         continue;
154       success = 0;
155       break;
156     }
157
158     printf("Verification: %s\n", (success) ? "OK" : "FAILED");
159     printf("Wall Clock Time = %f seconds\n", total_time);
160   }
161
162   // ---------- cleanup ----------
163
164   releaseVector(A);
165
166   MPI_Finalize();
167   // done
168   return (success) ? EXIT_SUCCESS : EXIT_FAILURE;
```

- Discuss the effects and implications of your parallelization.

Results of 1D Heat Stencil Execution (5 Repetitions)								
Impl/Ranks	**Problem Size**							
	768		1536		3072		6144	
	μ	σ	μ	σ	μ	σ	μ	σ
seq/1	0.56	0.0	2.38	0.01	9.65	0.02	38.47	0.1
par_complex/2	0.8	0.0	2.82	0.02	10.83	0.29	42.78	2.08
par_simple/2	0.75	0.0	2.74	0.01	10.79	0.49	41.47	0.05
par_complex/6	0.66	0.0	1.75	0.04	5.17	0.02	17.28	0.03
par_simple/6	0.53	0.0	1.46	0.01	4.62	0.01	16.19	0.04
par_complex/12	0.58	0.03	1.32	0.01	3.46	0.01	10.38	0.02
par_simple/12	**0.44**	0.01	**1.07**	0.01	**2.95**	0.03	9.34	0.05
par_complex/24	1.48	0.01	2.93	0.02	6.02	0.02	12.82	0.04
par_simple/24	0.82	0.01	1.61	0.01	3.44	0.02	7.64	0.03
par_complex/48	1.49	0.0	2.93	0.02	5.86	0.03	12.18	0.02
par_simple/48	0.85	0.01	1.63	0.01	3.31	0.02	7.22	0.05
par_complex/96	1.55	0.02	2.98	0.01	5.94	0.03	11.93	0.08
par_simple/96	0.83	0.01	1.6	0.03	3.14	0.02	**6.5**	0.02

Table 1: 1D Stencil Measurements

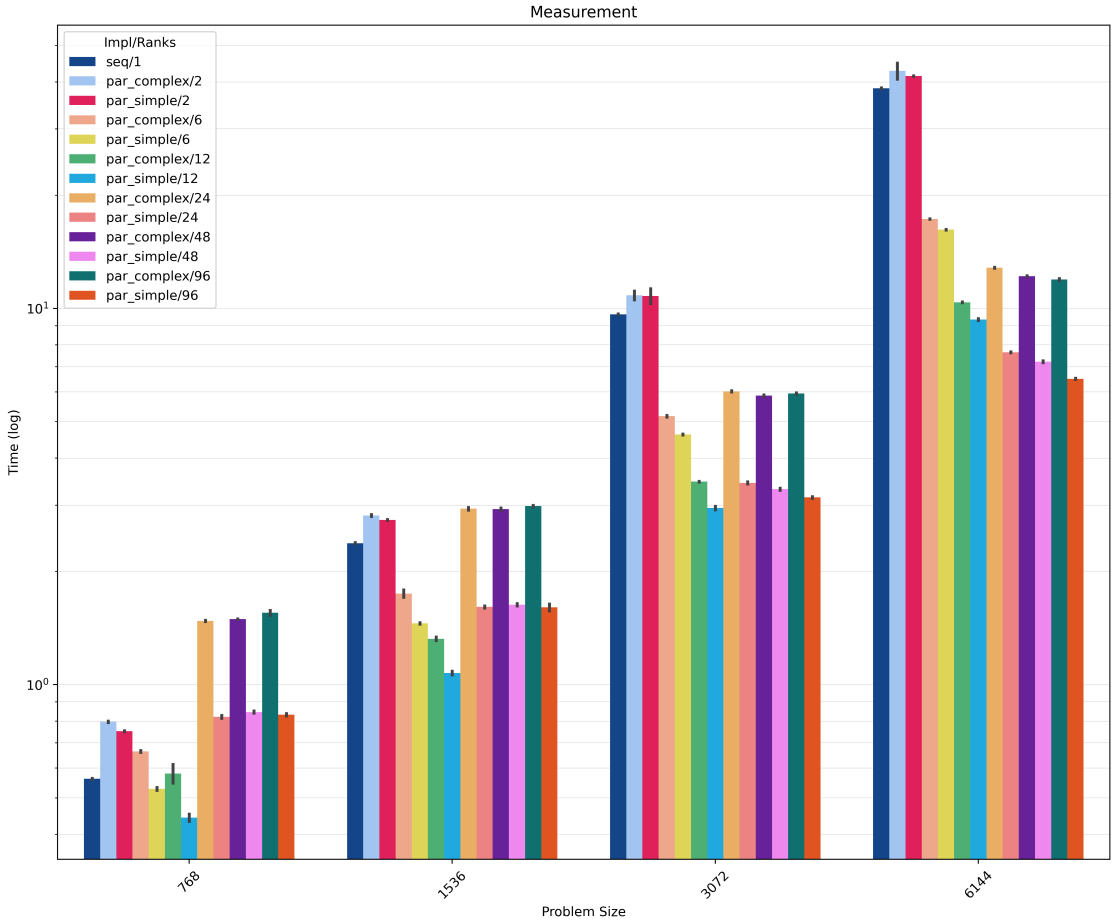

Figure 4: 1D Stencil Measurements.

The measurements of the 1D stencil heat propagation algorithms, as discussed earlier, are displayed in Table 1 and illustrated in Figure 4.

  - **Sequential vs. Parallel Execution**:
    The sequential version `seq/1` has the longest execution times, especially at larger problem sizes (e.g. 6144). The parallel implementations `par_complex` and `par_simple` significantly reduce execution time compared to the sequential version across all problem sizes. This shows that parallelization improves performance as the number of ranks increases.
  - **Performance at Different Ranks**:
    The execution time decreases with an increase in the number of ranks for both the `par_complex` and `par_simple` versions. This suggests that the workload is being effectively distributed across more resources. For higher ranks, especially between 48 and 96, performance gains diminish, with little difference in execution time.
  - **Complex vs. Simple Implementations**:
    The `par_simple` version consistently outperforms `par_complex` due to its use of non-blocking MPI calls, which allow ranks to continue processing while communication occurs. This approach is faster because MPI efficiently manages the communication order, ensuring all ranks receive the correct information with minimal delays. In contrast, the `par_complex` version uses a blocking communication pattern, where ranks must wait to send and receive messages sequentially, leading to slower performance.
  - **Standard Deviation and Stability**: The standard deviation σ values are very low across all implementations and ranks, which suggests that the measurements are stable and there is little variation between repetitions.

- Insert the measured wall time for N=6144 and 96 cores into the comparison spreadsheet