

Proseminar: High Performance Computing  
Assignment 02

## 1 Exercise 1

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

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4
5 double calculate_pi(int num_points) {
6     int points_in_circle = 0;
7
8     for (int count = 0; count < num_points; count++) {
9         double random_x = (double)rand() / RAND_MAX;
10        double random_y = (double)rand() / RAND_MAX;
11
12        if (random_x * random_x + random_y * random_y <= 1.0) {
13            points_in_circle++;
14        }
15    }
16
17    return 4.0 * points_in_circle / num_points;
18 }
19
20 int main(int argc, char *argv[]) {
21     int total_points = atoi(argv[1]);
22
23     double pi_approx = calculate_pi(total_points);
24
25     printf("Approximation of Pi: %f\n", pi_approx);
26
27     return 0;
28 }
```

Listing 1: Sequential Pi Approximation

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

As there are no dependencies while generating random points and checking if they lie in the circle, we can do that individually per rank. So we can just split the problem size  $N$  into  $N \div n_{ranks}$  chunks. At the end one core have to retrieve the number of points in circle from all other cores, build a sum out of it and calculate  $\pi$  with it.

For this we would use `MPI_Reduce(...)` as it can collect a variable from all ranks and directly use operations as addition by giving reduce the operation type `MPI_SUM`.

1.3 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`.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4 #include <mpi.h>
5
```

```

6 void calculate_pi(int num_points, int rank, int numProcs) {
7     // WTime Initialization
8     double starttime, endtime;
9     starttime = MPI_Wtime();
10    // Define how many points each rank generates and generate them
11    int points_for_rank = round(num_points / numProcs);
12    int points_in_circle_in_rank = 0;
13    int points_in_circle_global = 0;
14
15    for (int count = 0; count < points_for_rank; count++) {
16        double random_x = (double)rand() / RAND_MAX;
17        double random_y = (double)rand() / RAND_MAX;
18
19        if (random_x * random_x + random_y * random_y <= 1.0) {
20            points_in_circle_in_rank++;
21        }
22    }
23    // Gather results from all ranks
24    MPI_Reduce(&points_in_circle_in_rank, &points_in_circle_global, 1, MPI_INT, MPI_SUM, 0,
25    MPI_COMM_WORLD);
26
27    if (rank == 0) {
28        double pi_approx = 4.0 * points_in_circle_global / num_points;
29        endtime = MPI_Wtime();
30        printf("WTime: %f seconds\n", endtime - starttime);
31        printf("Approximation of Pi: %f\n", pi_approx);
32    }
33
34    int main(int argc, char *argv[]) {
35        // MPI Initialization
36        MPI_Init(&argc, &argv);
37        int rank, numProcs;
38        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
39        MPI_Comm_size(MPI_COMM_WORLD, &numProcs);
40
41        int total_points = atoi(argv[1]);
42
43        if (rank == 0) {
44            printf("Total Points used for Approximation: %i \n", total_points);
45        }
46
47        calculate_pi(total_points, rank, numProcs);
48
49        MPI_Finalize();
50        return 0;
51    }

```

Listing 2: Parallel Pi Approximation

## 1.4 Discuss the effects and implications of your parallelization.

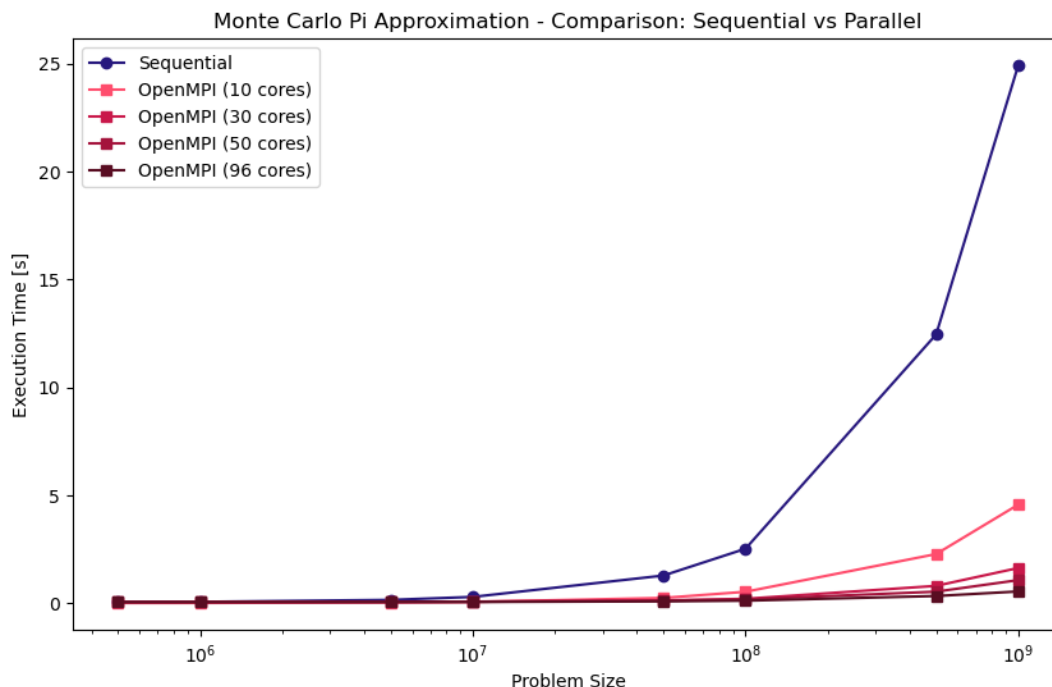


Figure 1: Monte Carlo Pi Approximation - Runtime Comparison

When running the experiment with a problem size below  $10^7$  the sequential code and the parallelized code have around the same runtimes. If more than  $10^7$  points are used the execution time of the sequential program grows rapidly. By adding more cores the highest problem size ( $10^9$ ) gets faster computed. The execution time for  $10^9$  points is 5x faster with 96 cores instead of just 10.

## 2 Exercise 2

**2.1 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.**

Iterative Stencil Loops

**2.2 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!**

- **Boundary-Exchange-Pattern** (using `MPI_Sendrecv`), because exchanging boundary values between processes is essential for correct calculations in each section.
- **Broadcast-Pattern** (with `MPI_Bcast`), to efficiently distribute initial data to all processes.
- **Gather-Pattern** (with `MPI_Gather`), to collect results from individual processes at the end of the computation.

Additional changes that need to be made include implementing functionality that splits the problem into chunks that can be distributed via MPI. This also includes additional error handling to ensure that the input fits our strategy.

## 2.3 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 #include <stdio.h>
2 #include <stdlib.h>
3 #include <mpi.h>
4
5 typedef double value_t;
6
7 #define RESOLUTION 120
8
9 // -- vector utilities --
10
11 typedef value_t *Vector;
12
13 Vector createVector(int N);
14
15 void releaseVector(Vector m);
16
17 void printTemperature(Vector m, int N);
18
19 // -- simulation code ---
20
21 int main(int argc, char **argv) {
22     // 'parsing' optional input parameter = problem size
23     int N = 2000;
24     if (argc > 1) {
25         N = atoi(argv[1]);
26     }
27     int T = N * 500;
28
29     // MPI Initialization
30     MPI_Init(&argc, &argv);
31     int rank, numProcs;
32     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
33     MPI_Comm_size(MPI_COMM_WORLD, &numProcs);
34
35     // Time measurement variables
36     double starttime, endtime;
37
38     // Calculate chunk size, start and end point for each process
39     int chunkSize = N / numProcs;
40     int remainder = N % numProcs;
41
42     if (rank == 0) {
43         if (remainder > 0) {
44             printf("The problem size can't be evenly distributed to the number of processes. Please
45             redefine it.");
46             return 0;
47         }
48     }
49
50     int startIdx = rank * chunkSize;
51     int endIdx = startIdx + chunkSize;
52     printf("[DEBUG] Rank: %i, ChunkSize: %i, startIdx: %i, endIdx: %i \n", rank, chunkSize, startIdx,
53           endIdx);
54
55     // create a buffer for storing temperature fields
56     Vector A = createVector(N);
57
58     int source_x = N / 4;
59
60     if (rank == 0) {
61         printf("Computing heat-distribution for room size N=%d for T=%d timesteps\n", N, T);
```

```

62 // ----- setup -----
63 // set up initial conditions in A
64 for (int i = 0; i < N; i++) {
65     A[i] = 273; // temperature is 0    C everywhere (273 K)
66 }
67
68 // and there is a heat source in one corner
69 A[source_x] = 273 + 60;
70
71 printf("Initial:\t");
72 printTemperature(A, N);
73 printf("\n");
74 }
75
76 // Distribute A to all processes
77 MPI_Bcast(A, N, MPI_DOUBLE, 0, MPI_COMM_WORLD);
78 printf("[DEBUG] Broadcasted A to every rank for computation. \n");
79
80 // Start time measurement before the computation begins
81 starttime = MPI_Wtime();
82
83 // ----- compute -----
84
85 // create a second buffer for the computation and initialize with A
86 Vector B = createVector(chunkSize);
87 Vector C = createVector(chunkSize);
88 for (int i = startIdx; i < endIdx; i++) {
89     B[i - startIdx] = A[i];
90 }
91 // create variables for the neighbours
92 value_t rightNeighbour = 0;
93 value_t leftNeighbour = 0;
94
95
96 // for each time step ..
97 for (int t = 0; t < T; t++) {
98     // Exchange boundary values with neighboring processes
99     // printf("[DEBUG] Rank %i started with timestep %i. \n", rank, t);
100     if ((rank > 0) & (rank < numProcs - 1)) {
101         // printf("[DEBUG] Rank %i tries to send and receive right boundary element ... \n", rank);
102         MPI_Sendrecv(&B[0], 1, MPI_DOUBLE, rank - 1, 0, &rightNeighbour, 1, MPI_DOUBLE, rank + 1, 0,
103             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
104         // printf("[DEBUG] Rank %i send and received right boundary element. \n", rank);
105
106         //printf("[DEBUG] Rank %i tries to send and receive left boundary element ... \n", rank);
107         MPI_Sendrecv(&B[chunkSize - 1], 1, MPI_DOUBLE, rank + 1, 1, &leftNeighbour, 1, MPI_DOUBLE,
108             rank - 1, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
109         //printf("[DEBUG] Rank %i send and received left boundary element. \n", rank);
110     }
111     if (rank == 0) {
112         //printf("[DEBUG] Rank %i tries to receive right boundary element ... \n", rank);
113         MPI_Sendrecv(&B[chunkSize - 1], 1, MPI_DOUBLE, rank + 1, 1, &rightNeighbour, 1, MPI_DOUBLE,
114             rank + 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
115         // printf("[DEBUG] Rank %i sent received right boundary element and sent left one. \n", rank)
116     };
117     if (rank == numProcs - 1) {
118         //printf("[DEBUG] Rank %i tries to receive left boundary element ... \n", rank);
119         MPI_Sendrecv(&B[0], 1, MPI_DOUBLE, rank - 1, 0, &leftNeighbour, 1, MPI_DOUBLE, rank - 1, 1,
120             MPI_COMM_WORLD, MPI_STATUS_IGNORE);
121         //printf("[DEBUG] Rank %i received left boundary element and sent right one. \n", rank);
122     }
123
124     // .. we propagate the temperature
125     for (long long i = startIdx; i < endIdx; i++) {
126         // center stays constant (the heat is still on)
127         if (i == source_x) {
128             C[i - startIdx] = B[i - startIdx];

```

```

125     continue;
126 }
127
128 // get temperature at current position
129 value_t tc = B[i - startIdx];
130
131 // get temperatures of adjacent cells
132 value_t tl;
133 if (i == startIdx) {
134     tl = (rank > 0) ? leftNeighbour : tc;
135 } else {
136     tl = B[i - 1 - startIdx];
137 }
138 value_t tr;
139 if (i == endIdx - 1) {
140     tr = (rank < numProcs - 1) ? rightNeighbour : tc;
141 } else {
142     tr = B[i - startIdx + 1];
143 }
144
145 // compute new temperature at current position
146 C[i - startIdx] = tc + 0.2 * (tl + tr + (-2 * tc));
147 }
148
149 Vector H = B;
150 B = C;
151 C = H;
152
153 if (t % 10000 == 0) {
154     MPI_Gather(B, chunkSize, MPI_DOUBLE, A, chunkSize, MPI_DOUBLE, 0, MPI_COMM_WORLD);
155
156     if (rank == 0) {
157         printf("Step t=%d:\t", t);
158         printTemperature(A, N);
159         printf("\n");
160     }
161 }
162 }
163
164 MPI_Gather(B, chunkSize, MPI_DOUBLE, A, chunkSize, MPI_DOUBLE, 0, MPI_COMM_WORLD);
165
166 releaseVector(B);
167
168 // End time measurement after computation
169 endtime = MPI_Wtime();
170
171 // ----- check -----
172 int success = 1;
173
174 if (rank == 0) {
175     printf("Final:\t\t");
176     printTemperature(A, N);
177     printf("\n");
178
179     for (long long i = 0; i < N; i++) {
180         value_t temp = A[i];
181         if (273 <= temp && temp <= 273 + 60)
182             continue;
183         success = 0;
184         break;
185     }
186
187     printf("Verification: %s\n", (success) ? "OK" : "FAILED");
188
189     // Print the time
190     printf("WTime: %f seconds\n", endtime - starttime);
191 }
192

```

```

193 // ----- cleanup -----
194
195 releaseVector(A);
196
197 // done
198 MPI_Finalize();
199 return (success) ? EXIT_SUCCESS : EXIT_FAILURE;
200 }
201
202 Vector createVector(int N) {
203     // create data and index vector
204     return malloc(sizeof(value_t) * N);
205 }
206
207 void releaseVector(Vector m) { free(m); }
208
209 void printTemperature(Vector m, int N) {
210     const char *colors = " .-:=+*^X#%@";
211     const int numColors = 12;
212
213     // boundaries for temperature (for simplicity hard-coded)
214     const value_t max = 273 + 30;
215     const value_t min = 273 + 0;
216
217     // set the 'render' resolution
218     int W = RESOLUTION;
219
220     // step size in each dimension
221     int sW = N / W;
222
223     // room
224     // left wall
225     printf("X");
226     // actual room
227     for (int i = 0; i < W; i++) {
228         // get max temperature in this tile
229         value_t max_t = 0;
230         for (int x = sW * i; x < sW * i + sW; x++) {
231             max_t = (max_t < m[x]) ? m[x] : max_t;
232         }
233         value_t temp = max_t;
234
235         // pick the 'color'
236         int c = ((temp - min) / (max - min)) * numColors;
237         c = (c >= numColors) ? numColors - 1 : ((c < 0) ? 0 : c);
238
239         // print the average temperature
240         printf("%c", colors[c]);
241     }
242     // right wall
243     printf("X");
244 }

```

Listing 3: Parallel propagation

## 2.4 Discuss the effects and implications of your parallelization.

### 1. Performance Improvements

With MPI, the problem is split into parts, and each process (rank) works on one part. This way, the work is shared between cores or nodes, which makes the simulation faster compared to the sequential version.

### 2. Data Communication Overhead

The parallel version needs processes to exchange boundary information using `MPI_Sendrecv`. This adds communication overhead, especially as the number of processes increases. Also, collective functions like `MPI_Bcast` and `MPI_Gather` make all processes wait for each other, which can slow down the program if not managed well.

### 3. Scalability Considerations

Each process handles the same amount of work. But if the problem size does not divide evenly between processes (e.g.,  $N \% \text{numProcs} \neq 0$ ), the program stops. More processes can also increase communication overhead, especially if there is too much communication compared to the actual computation.

### 4. Potential Bottlenecks

Global synchronization using `MPI_Bcast` and `MPI_Gather` can cause delays because all processes must wait for the slowest one to finish its work. This may lead to idle time, reducing efficiency.