Karl Grossmann, Lukas Mayr
14.10.2024

Proseminar: High Performance Computing
Assignment 02

1 Exercise 1

1.1 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.

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

double calculate_pi(int num_points) {
    int points_in_circle = 0;

    for (int count = 0; count < num_points; count++) {
        double random_x = (double)rand() / RAND_MAX;
        double random_y = (double)rand() / RAND_MAX;

        if (random_x * random_x + random_y * random_y <= 1.0) {
            points_in_circle++;
        }
    }

    return 4.0 * points_in_circle / num_points;
}

int main(int argc, char *argv[]) {
    int total_points = atoi(argv[1]);

    double pi_approx = calculate_pi(total_points);

    printf("Approximation of Pi: %f\n", pi_approx);

    return 0;
}
```

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 π 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.

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <mpi.h>

```

```c
void calculate_pi(int num_points, int rank, int numProcs) {
    // WTime Initialization
    double starttime, endtime;
    starttime = MPI_Wtime();
    // Define how many points each rank generates and generate them
    int points_for_rank = round(num_points / numProcs);
    int points_in_circle_in_rank = 0;
    int points_in_circle_global = 0;

    for (int count = 0; count < points_for_rank; count++) {
        double random_x = (double)rand() / RAND_MAX;
        double random_y = (double)rand() / RAND_MAX;

        if (random_x * random_x + random_y * random_y <= 1.0) {
            points_in_circle_in_rank++;
        }
    }
    // Gather results from all ranks
    MPI_Reduce(&points_in_circle_in_rank, &points_in_circle_global, 1, MPI_INT, MPI_SUM, 0,
    MPI_COMM_WORLD);

    if (rank == 0) {
        double pi_approx = 4.0 * points_in_circle_global / num_points;
        endtime = MPI_Wtime();
        printf("WTime: %f seconds\n", endtime - starttime);
        printf("Approximation of Pi: %f\n", pi_approx);
    }
}

int main(int argc, char *argv[]) {
    // MPI Initialization
    MPI_Init(&argc, &argv);
    int rank, numProcs;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &numProcs);

    int total_points = atoi(argv[1]);

    if (rank == 0) {
        printf("Total Points used for Approximation: %i \n", total_points);
    }

    calculate_pi(total_points, rank, numProcs);

    MPI_Finalize();
    return 0;
}
```

Listing 2: Parallel Pi Approximation

1.4 Discuss the effects and implications of your parallelization.

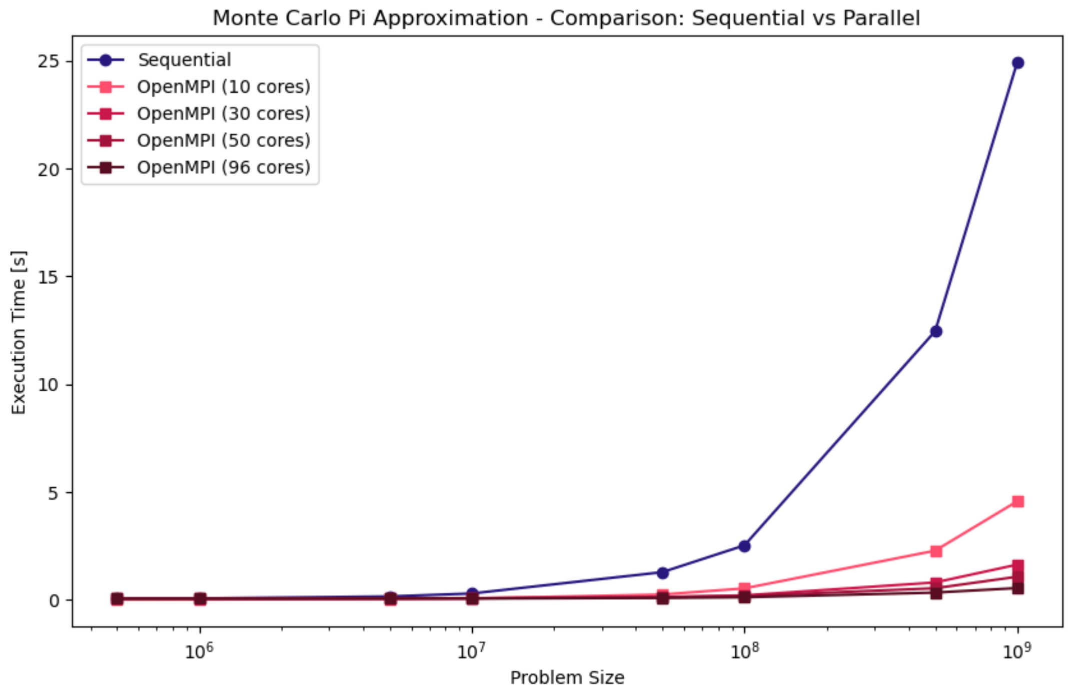


Figure 1: Monte Carlo Pi Approximation - Runtime Comparison

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 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.

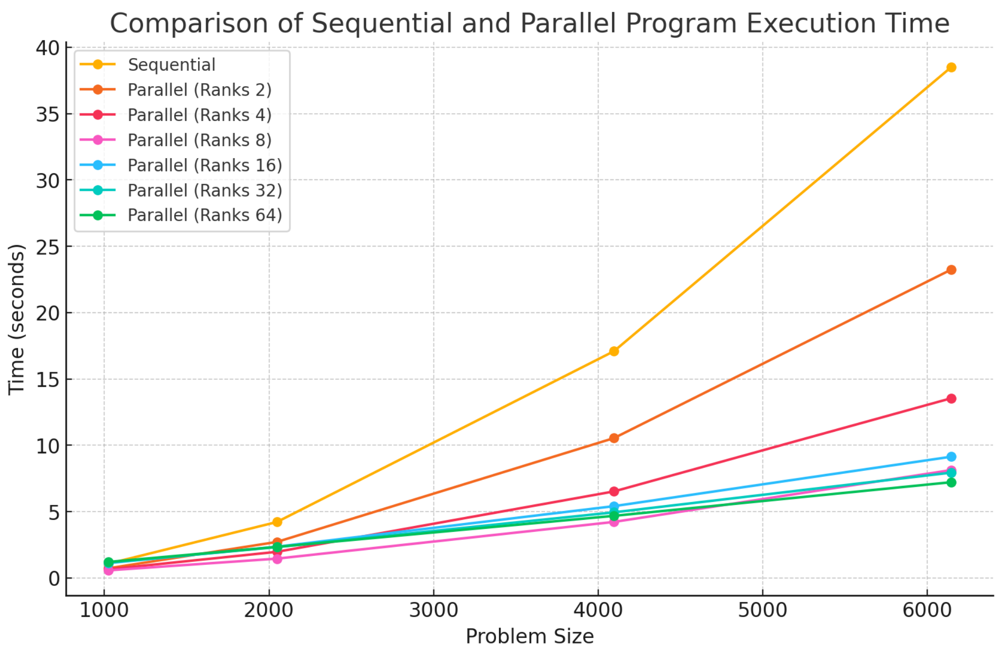
Additional changes that need to be made include implementing functionality that splits the problem into chunks that can be distributed via MPI. This also includes additional error handling to ensure that the input fits our strategy.

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.

```c
#include <stdio.h>
#include <stdlib.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);

// -- simulation code ---

int main(int argc, char **argv) {
  // 'parsing' optional input parameter = problem size
  int N = 2000;
  if (argc > 1) {
    N = atoi(argv[1]);
  }
  int T = N * 500;

  // MPI Initialization
  MPI_Init(&argc, &argv);
  int rank, numProcs;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &numProcs);

  // Time measurement variables
  double starttime, endtime;

  // Calculate chunk size, start and end point for each process
  int chunkSize = N / numProcs;
  int remainder = N % numProcs;

  if (rank == 0) {
    if (remainder > 0) {
      printf("The problem size can't be evenly distributed to the number of processes. Please
    redefine it.");
      return 0;
    }
  }

  int startIdx = rank * chunkSize;
  int endIdx = startIdx + chunkSize;
  printf("[DEBUG] Rank: %i, ChunkSize: %i, startIdx: %i, endIdx: %i \n", rank, chunkSize, startIdx,
     endIdx);

  // create a buffer for storing temperature fields
  Vector A = createVector(N);

  int source_x = N / 4;

  if (rank == 0) {

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

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

    // and there is a heat source in one corner
    A[source_x] = 273 + 60;

    printf("Initial:\t");
    printTemperature(A, N);
    printf("\n");
  }

  // Distribute A to all processes
  MPI_Bcast(A, N, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  printf("[DEBUG] Broadcasted A to every rank for computation. \n");

  // Start time measurement before the computation begins
  starttime = MPI_Wtime();

  // ---------- compute ----------

  // create a second buffer for the computation and initialize with A
  Vector B = createVector(chunkSize);
  Vector C = createVector(chunkSize);
  for (int i = startIdx; i < endIdx; i++) {
      B[i - startIdx] = A[i];
  }
  // create variables for the neighbours
  value_t rightNeighbour = 0;
  value_t leftNeighbour = 0;


  // for each time step ..
  for (int t = 0; t < T; t++) {
    // Exchange boundary values with neighboring processes
    // printf("[DEBUG] Rank %i started with timestep %i. \n", rank, t);
    if ((rank > 0) & (rank < numProcs - 1)) {
      // printf("[DEBUG] Rank %i tries to send and receive right boundary element ... \n", rank);
      MPI_Sendrecv(&B[0], 1, MPI_DOUBLE, rank - 1, 0, &rightNeighbour, 1, MPI_DOUBLE, rank + 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      // printf("[DEBUG] Rank %i send and received right boundary element. \n", rank);

      //printf("[DEBUG] Rank %i tries to send and receive left boundary element ... \n", rank);
      MPI_Sendrecv(&B[chunkSize - 1], 1, MPI_DOUBLE, rank + 1, 1, &leftNeighbour, 1, MPI_DOUBLE, rank - 1, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      //printf("[DEBUG] Rank %i send and received left boundary element. \n", rank);
    }
    if (rank == 0) {
      //printf("[DEBUG] Rank %i tries to receive right boundary element ... \n", rank);
      MPI_Sendrecv(&B[chunkSize - 1], 1, MPI_DOUBLE, rank + 1, 1, &rightNeighbour, 1, MPI_DOUBLE, rank + 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      // printf("[DEBUG] Rank %i sent received right boundary element and sent left one. \n", rank);
    }
    if (rank == numProcs - 1) {
      //printf("[DEBUG] Rank %i tries  to receive left boundary element ... \n", rank);
      MPI_Sendrecv(&B[0], 1, MPI_DOUBLE, rank - 1, 0, &leftNeighbour, 1, MPI_DOUBLE, rank - 1, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      //printf("[DEBUG] Rank %i received left boundary element and sent right one. \n", rank);
    }

    // .. we propagate the temperature
    for (long long i = startIdx; i < endIdx; i++) {
      // center stays constant (the heat is still on)
      if (i == source_x) {
        C[i - startIdx] = B[i - startIdx];
        continue;
      }

      // get temperature at current position
      value_t tc = B[i - startIdx];

      // get temperatures of adjacent cells
      value_t tl;
      if (i == startIdx) {
        tl = (rank > 0) ? leftNeighbour : tc;
      } else {
          tl = B[i - 1- startIdx];
      }
      value_t tr;
      if (i == endIdx - 1) {
          tr = (rank < numProcs - 1) ? rightNeighbour : tc;
      } else {
          tr = B[i - startIdx + 1];
      }

      // compute new temperature at current position
      C[i - startIdx] = tc + 0.2 * (tl + tr + (-2 * tc));
    }

    Vector H = B;
    B = C;
    C = H;

    if (t % 10000 == 0) {
      MPI_Gather(B, chunkSize, MPI_DOUBLE, A, chunkSize, MPI_DOUBLE, 0, MPI_COMM_WORLD);

      if (rank == 0) {
        printf("Step t=%d:\t", t);
        printTemperature(A, N);
        printf("\n");
      }
    }
  }

  MPI_Gather(B, chunkSize, MPI_DOUBLE, A, chunkSize, MPI_DOUBLE, 0, MPI_COMM_WORLD);

  releaseVector(B);

  // End time measurement after computation
  endtime = MPI_Wtime();

  // ---------- check ----------
  int success = 1;

  if (rank == 0) {
    printf("Final:\t\t");
    printTemperature(A, N);
    printf("\n");

    for (long long i = 0; i < N; i++) {
      value_t temp = A[i];
      if (273 <= temp && temp <= 273 + 60)
        continue;
      success = 0;
      break;
    }

    printf("Verification: %s\n", (success) ? "OK" : "FAILED");

    // Print the time
    printf("WTime: %f seconds\n", endtime - starttime);
  }

```

```c
  // ---------- cleanup ----------

  releaseVector(A);

  // done
  MPI_Finalize();
  return (success) ? EXIT_SUCCESS : EXIT_FAILURE;
}

Vector createVector(int N) {
  // create data and index vector
  return malloc(sizeof(value_t) * N);
}

void releaseVector(Vector m) { free(m); }

void printTemperature(Vector m, int N) {
  const char *colors = " .-:=+*^X#%@";
  const int numColors = 12;

  // boundaries for temperature (for simplicity hard-coded)
  const value_t max = 273 + 30;
  const value_t min = 273 + 0;

  // set the 'render' resolution
  int W = RESOLUTION;

  // step size in each dimension
  int sW = N / W;

  // room
  // left wall
  printf("X");
  // actual room
  for (int i = 0; i < W; i++) {
    // get max temperature in this tile
    value_t max_t = 0;
    for (int x = sW * i; x < sW * i + sW; x++) {
      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");
}
```

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.