Question 1: Parallel Scaling

The task of question 1 was to plot the strong scaling and weak scaling behaviours of a code run on data sets of different sizes. The runtime achieved with different numbers of processors on varying data set sizes is displayed in the figure below.

Note: All the relevant conclusions and computations are found in Figure 2 and Figure 3.

Strong scaling.

Using strong scaling as performance metric the amount of work remains the same, no matter how many CPUs are used to work on the problem. This allows to draw a conclusion on the scalability of the code implementation itself. In order to plot strong scaling from the provided data set one column out of the four given columns needs to be used. This can be any column as the amount of work, i.e. the number of points remains the same within a column.

Weak scaling.

Weak scaling is a performance metric that allows one to draw a conclusion on the impact of communication on the code implementation. The amount of work per worker, i.e. the amount of work per CPU is kept constant. The reference point for the plot is given by P = 1 and N = 500. As the complexity of the algorithm is $O(n^2)$, an increase in the number of points by a factor of x requires an increase in the number of CPUs by a factor of x^2 to obtain constant work for each CPU. Therefore, points of the diagonal of the table below need to be used for plotting. These values are underlined in blue in Figure 2.

P\N	500	1000	1500	2000
1	6.00	30.00	72.00	120.00
4	1.50	7.50	18.00	30.00
9	0.75	3.50	9.00	20.00
16	0.50	2.15	6.00	12.00
24	0.40	1.50	4.50	10.00

Figure 1. Provided data set for plotting the strong scaling and weak scaling behaviour of the code.

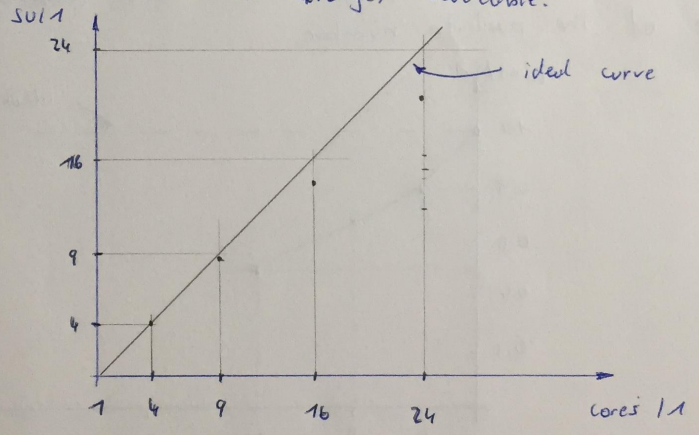
Question 1:

	Number of particles			
p	500	1000	1500	2000
1	6.00	30	72.00	120.00
4	1.50 1.50 (1x)	7.5 7.5 (1x)	18.00 18.00 (1x)	30.00 30.00 (1x)
9	0.75 0.67 (1.12x)	3.5 3.3 (1.05x)	9.00 8.00 (1.13x)	20.00 13.33 (1.50x)
16	0.50 0.38 (1.32x)	2.15 1.88 (1.14x)	6.00 4.50 (1.33x)	12.00 7.50 (1.6x)
24	0.40 0.25 (1.6x)	1.50 1.25 (1.2x)	4.50 3.00 (1.5x)	10.00 5.00 (2x)

$SU = \frac{T(1)}{T(n)}$ .... Speed-up 1/1

Relativ errors (m.l.expected)

Theory: A doubling of the number of CPUs should reduce the run-time by a factor of 2. If this works code can be described as strongly scaleable.



The best strong scaling behaviour is shown by N = 1000 points. The table shows the real measured run-times and the expected run-times in case of strong-scaling.

Strong scaling : work = const.

- 1 -

Figure 2. Plot of the strong scaling behaviour of the given data set. Both the ideal curve and the actual data points used from the table in this figure are displayed. As the same code was run on the different data sizes any set could be used, but 1000 points showed the nicest strong scaling so that these were used for plotting.

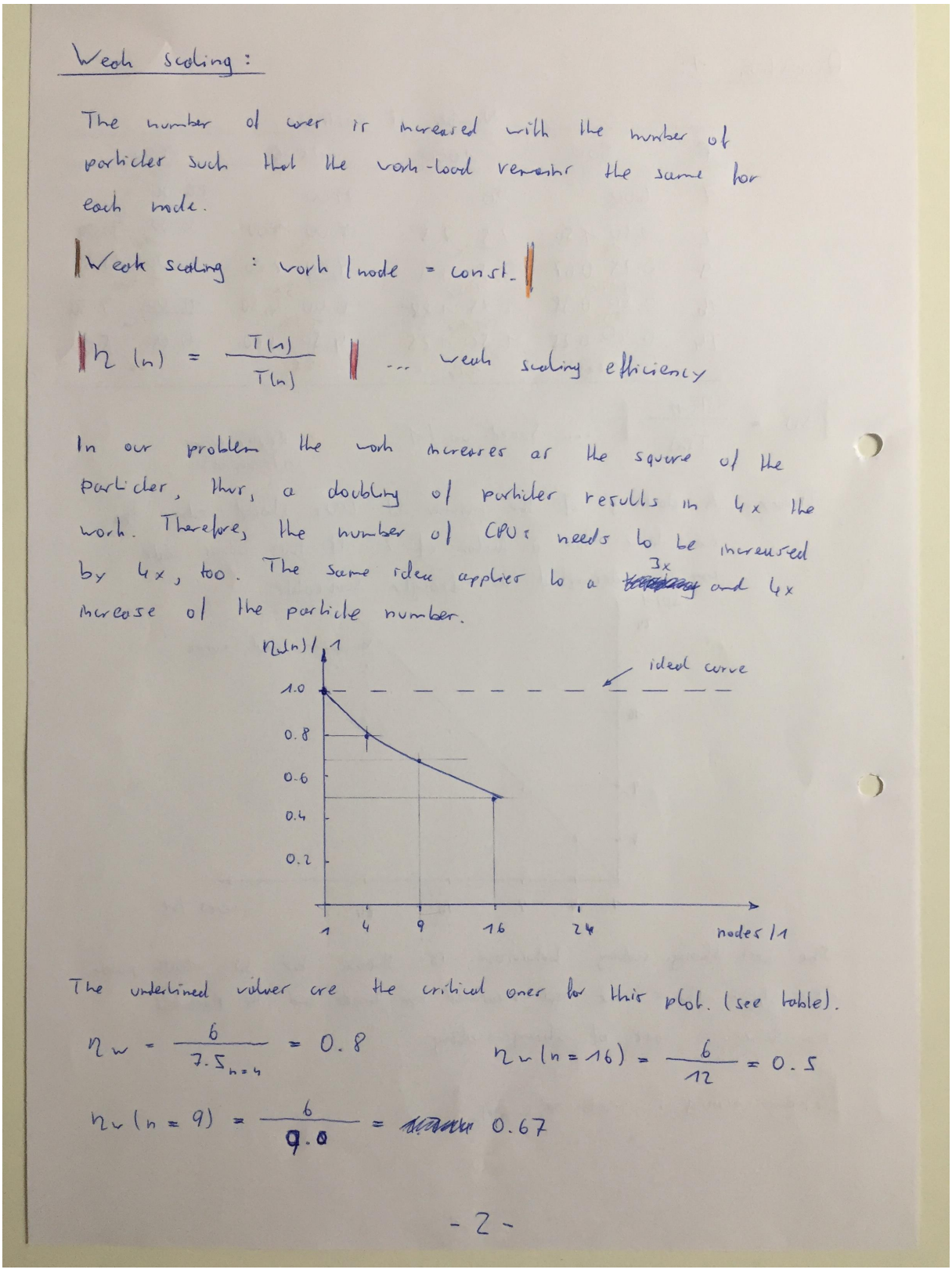


Figure 3. Plot of the weak scaling behaviour of the given data set. In order to obtain the weak scaling the work among the different workers (ranks) must remain constant. The reference point is P=1 and N=500. The important point to consider plotting weak scaling is that the big O notation of the code needs to be taken into consideration. As our complexity is O(2) a doubling of the data points means using 4 times the number of workers.

Question 2: Diffusion

This question deals with the discretized solution of the diffusion equation applying Dirichlet boundary conditions, i.e. no periodic boundaries are used. A square of a given size LxL features another square in its centre of size 0.5Lx0.5L with a concentration of 1. With every time step of the solution of the PDE the liquid flows into the surrounding area of the bigger square, and eventually some particles leave the total square (Dirichlet) resulting in an overall drop of the concentration in the LxL area.

The task is to parallelize the code using MPI. The overall grid is distributed into local rectangles in a row-wise manner. Therefore, a communication of the ranks is necessary to exchange the upper and lower bounds of the local grid area. This communication works as follows. Each rank is assigned two additional rows, which are so-called “ghost-cells”. One of these additional rows is the first row of the local rectangle (index 0) and the second one is the last row (index local_N+1). A rank sends its information stored in the second row (index 1) to the lower rank, which stores it in its upper ghost cell row (index local_N+1). The information of the second to last row (index local_N) is sent to the rank above, which stores the data in its ghost cell at the lowest row (index 0). By filling these ghost cell rows it is possible to use the discretized form of the diffusion equation, because the local information around a point of interest in the grid can be computed even at the boundaries. The first rank (lowest row chunk) and last rank (top row chunk) are treated differently. The lowest rank only sends data to the above rank, whereas the top rank only sends it data to the rank below it.

(a) (b)

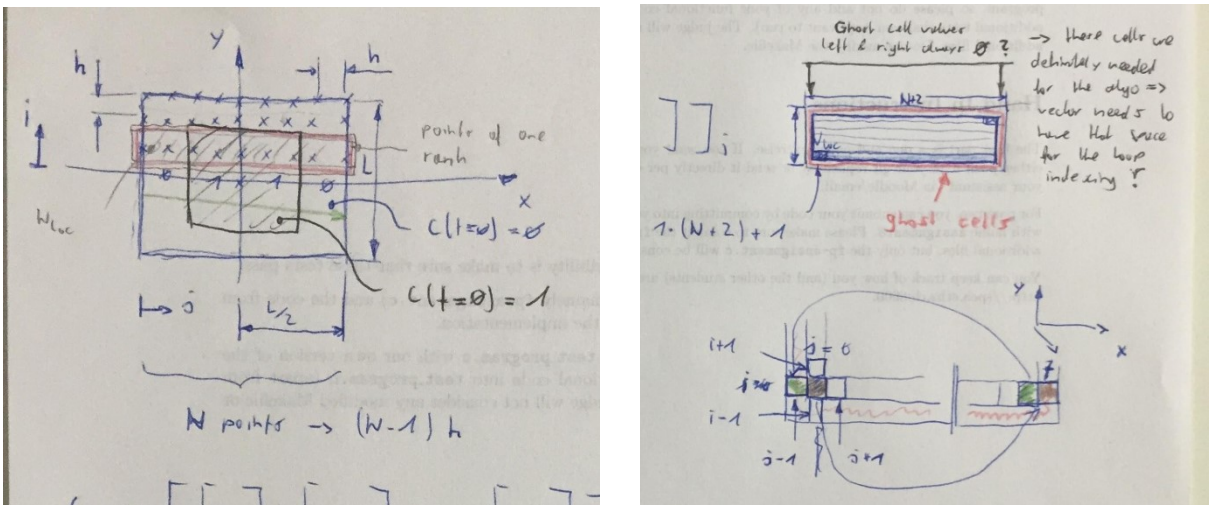


Figure 4. (a) Visualization of the grid and an exemplary chunk of a rank (red). **(b)** Ghost cell representation around the local grid of a rank.

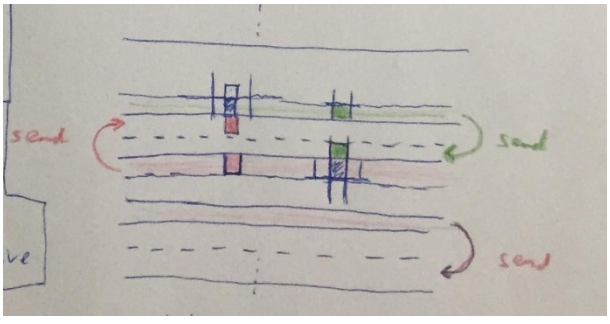


Figure 5. Schematic of how the ranks send their information to the above/below rank.

```
47      void advance()
48      {
49
50          // TODO: Implement Blocking MPI communication to exchange the ghost
51          // cells on a periodic domain required to compute the central finite
52          // differences below.
53
54          // *** start MPI part ***
55
56          // In case only one rank is used, the send receive must not be
57          // executed, it is going to cause an error.
58          if (size > 1) {
59
60              // Define number of request according to the rank number.
61              int nr;
62              if (rank > 0 && rank < (size-1))
63                  nr = 4;
64              else
65                  // Rank 0 and rank (size-1) are treated differently.
66                  // These two ranks do only send one message.
67                  // Rank 0 sends it to rank 1 and rank (size-1) sends it to rank (size-2).
68                  nr = 2;
69
70                  // Define number of send and receive requests.
71              std::vector<MPI_Request> requests(nr, MPI_REQUEST_NULL);
72              // Define the statuses.
73              std::vector<MPI_Status> stats(nr);
74              // Define a tag for this ranke.
75              int tag = rank;
76
77              // Send messages.
78              int suIdx = (N+2) * local_N; // Send up index.
79              int sdIdx = N+2;             // Send down index.
80
81              // Send down.
82              if (rank > 0)
83                  MPI_Isend(&c[sdIdx], N+2, MPI_DOUBLE, rank-1, tag, MPI_COMM_WORLD, &requests[0]);
84              // Send up.
85              if (rank == 0)
86                  MPI_Isend(&c[suIdx], N+2, MPI_DOUBLE, rank+1, tag, MPI_COMM_WORLD, &requests[0]);
87              else if (rank < (size-1))
88                  MPI_Isend(&c[suIdx], N+2, MPI_DOUBLE, rank+1, tag, MPI_COMM_WORLD, &requests[1]);
89              // Receive messages.
90              int raIdx = (N+2) * (local_N + 1); // Receive from above index.
91              int rbIdx = 0;                     // Receive from below index.
92
93              // Receive from below.
94              if (rank == (size-1))
```

Figure 6. The advance() function computes the concentration at each small square of the discretized grid for the next timestep (Part 1/2).

```
76
77              // Send messages.
78              int suIdx = (N+2) * local_N; // Send up index.
79              int sdIdx = N+2;             // Send down index.
80
81              // Send down.
82              if (rank > 0)
83                  MPI_Isend(&c[sdIdx], N+2, MPI_DOUBLE, rank-1, tag, MPI_COMM_WORLD, &requests[0]);
84              // Send up.
85              if (rank == 0)
86                  MPI_Isend(&c[suIdx], N+2, MPI_DOUBLE, rank+1, tag, MPI_COMM_WORLD, &requests[0]);
87              else if (rank < (size-1))
88                  MPI_Isend(&c[suIdx], N+2, MPI_DOUBLE, rank+1, tag, MPI_COMM_WORLD, &requests[1]);
89              // Receive messages.
90              int raIdx = (N+2) * (local_N + 1); // Receive from above index.
91              int rbIdx = 0;                     // Receive from below index.
92
93              // Receive from below.
94              if (rank == (size-1))
95                  MPI_Irecv(&c[rbIdx], N+2, MPI_DOUBLE, rank-1, tag-1, MPI_COMM_WORLD, &requests[1]);
96              else if (rank > 0)
97                  MPI_Irecv(&c[rbIdx], N+2, MPI_DOUBLE, rank-1, tag-1, MPI_COMM_WORLD, &requests[2]);
98              // Receive from above
99              if (rank == 0)
100                 MPI_Irecv(&c[raIdx], N+2, MPI_DOUBLE, rank+1, tag+1, MPI_COMM_WORLD, &requests[1]);
101             else if (rank < (size-1))
102                 MPI_Irecv(&c[raIdx], N+2, MPI_DOUBLE, rank+1, tag+1, MPI_COMM_WORLD, &requests[3]);
103
104             // Wait until all requests have completed.
105             MPI_Waitall(nr, requests.data(), stats.data());
106         }
107
108         /* Central differences in space, forward Euler in time, Dirichlet BCs */
109         for (int i = 1; i <= local_N; ++i)
110             for (int j = 1; j <= N; ++j)
111                 c_tmp[i * (N + 2) + j] =
112                     c[i * (N + 2) + j] +
113                     aux * (c[i * (N + 2) + (j + 1)] + c[i * (N + 2) + (j - 1)] +
114                            c[(i + 1) * (N + 2) + j] + c[(i - 1) * (N + 2) + j] -
115                            4 * c[i * (N + 2) + j]);
116
117         // Use swap instead of rho_ = rho_tmp__. This is much more efficient,
118         // because it does not copy element by element, just replaces storage
119         // pointers.
120         using std::swap;
121         swap(c_tmp, c);
122     }
```

Figure 7. The advance() function computes the concentration at each small square of the discretized grid for the next timestep (Part 2/2).

```
123
124     void compute_diagnostics(const double t)
125     {
126         /*
127          * Sum the concentration of all local grids of the different
128          * ranks and sum them up.
129          * Write the results to the diagnostics data structure.
130          */
131
132         // Storage variable for the concentration amount of this iteration.
133         double ammount = 0.0;
134
135         /* Integration to compute total concentration */
136         for (int i = 1; i <= local_N; ++i)
137             for (int j = 1; j <= N; ++j)
138                 ammount += c[i * (N + 2) + j] * h * h;
139
140         // TODO: Sum total ammount from all ranks
141         // *** start MPI part ***
142         MPI_Reduce(rank ? &ammount : MPI_IN_PLACE, &ammount, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
143         // *** end MPI part ***
144
145         if (rank == 0) {
146             std::cout << "t = " << t << " ammount = " << ammount << '\n';
147             diag.push_back(Diagnostics(t, ammount));
148         }
149     }
150
```

Figure 8. The function compute_diagnostics() computes the overall concentration of the discretized grid at each time step. Due to the use of Dirichlet boundary conditions a decrease of the concentration over time is expected as molecules/particles of the fluid will flow out of bounds of the grid.

```
164
165     void compute_histogram()
166     {
167         /*
168         * Distributes the concentration of all the small grid
169         * squares into a histogram of M bins.
170         */
171
172         /* Number of histogram bins */
173         const int M = 10;
174         std::vector<int> hist(M, 0);
175
176         /* Find the local max and min density values */
177         double max_c, min_c, c0;
178         max_c = c[1 * (N + 2) + 1];
179         min_c = c[1 * (N + 2) + 1];
180
181         for (int i = 1; i <= local_N; ++i)
182             for (int j = 1; j <= N; ++j) {
183             c0 = c[i * (N + 2) + j];
184                 if (c0 > max_c)
185                     max_c = c0;
186                 if (c0 < min_c)
187                     min_c = c0;
188             }
189
190         // TODO: Compute the global min and max concentration values on this myRank and
191         // store the result in min_c and max_c, respectively.
192
193         // *** start MPI part ***
194         MPI_Allreduce(rank ? &min_c : MPI_IN_PLACE, &min_c, 1, MPI_DOUBLE, MPI_MIN, MPI_COMM_WORLD);
195         MPI_Allreduce(rank ? &max_c : MPI_IN_PLACE, &max_c, 1, MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD);
196         // *** end MPI part ***
197
198         double epsilon = 1e-8;
199         double dh = (max_c - min_c + epsilon) / M;
200
201         for (int i = 1; i <= local_N; ++i)
202             for (int j = 1; j <= N; ++j) {
203                 // Compute histrogram bin index.
204                 int bin = (c[i * (N + 2) + j] - min_c) / dh;
205                 // Update histogram at a particular bin.
206                 hist[bin]++;
207             }
208
209         // TODO: Compute the sum of the histogram bins over all ranks and store
210         // the result in the array g_hist.  Only myRank 0 must print the result.
211         std::vector<int> g_hist(M, 0);
```

Figure 9. The function compute_histogram() computes the global min and max concentrations of the liquid inside the square. The min. global concentration is then subtracted from the concentrations at every point of the grid. This value is stored in a bin of the histogram to make the distribution of concentrations in the grid visible (Part 1/2).

```
208
209         // TODO: Compute the sum of the histogram bins over all ranks and store
210         // the result in the array g_hist.  Only myRank 0 must print the result.
211         std::vector<int> g_hist(M, 0);
212         // *** start MPI part ***
213         MPI_Reduce(&hist[0], &g_hist[0], M, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
214
215         // Print results of individual ranks.
216         for (int i = 0; i < size; i++) {
217             if (rank == i) {
218             printf("\nRank %d\n", rank);
219                 for (int l = 0; l < M; l++)
220                     printf("bin[%d] = %d\n", l, hist[l]);
221             }
222         }
223         // *** end MPI part ***
224
225         // Print results (histogram of all the ranks combined).
226         if (rank == 0) {
227             printf("====================================\n");
228             printf("Output of compute_histogram():\n");
229             int gl = 0;
230             // Print results to console.
231             for (int i = 0; i < M; i++) {
232                 printf("g_bin[%d] = %d\n", i, g_hist[i]);
233                 gl += g_hist[i];
234             }
235             // Print the total number of elements as a check.
236             printf("Total elements = %d\n", gl);
237         }
238
239     } // end public
```

Figure 10. The function compute_histogram() computes the global min and max concentrations of the liquid inside the square. The min. global concentration is then subtracted from the concentrations at every point of the grid. This value is stored in a bin of the histogram to make the distribution of concentrations in the grid visible (Part 2/2).

```cpp
int main(int argc, char* argv[])
{
    if (argc < 4) {
        std::cerr << "Usage: " << argv[0] << " D L N \n";
        return 1;
    }

    // TODO: Start-up the MPI environment and determine this process' myRank ID
    // as well as the total number of processes (=ranks) involved in the
    // communicator

    int rank, size;
    // *** start MPI part ***
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    // *** end MPI part ***

    // Variable to check timing t = 0.5 s.
    int check = 1;

    // Diffusion constant.
    const double D = std::stod(argv[1]);
    // Length of the square in which the substance diffuses.
    const double L = std::stod(argv[2]);
    // Number of grid points (square grid).
    const int N = std::stoul(argv[3]);

    if (rank == 0)
        printf("Running Diffusion 2D on a %d x %d grid with %d ranks.\n",N,N,size);

    Diffusion system(D, L, N, rank, size);
    system.compute_diagnostics(0);
    for (int step = 0; step < 10000; ++step) {
    system.advance();
        system.compute_diagnostics(system.dt * step);
        // Print results of time t = 0.5 s to the console.
        if (check == 1 && (system.dt * step) > 0.5) {
            check = 0;
            system.compute_histogram();
        }
    }
    system.compute_histogram();
    if (rank == 0)
        system.write_diagnostics("diagnostics.dat");

    // TODO: Shutdown the MPI environment
    // *** start MPI part ***
    MPI_Finalize();
    // *** end MPI part ***
    return 0;
}
```

Figure 11. The main() function with the MPI initialization and finalization.

The following figure displays the overall results of the computed histogram. The numbers for each histogram bin match the numbers of the serial code implemenatation and can therefore be described as correct.

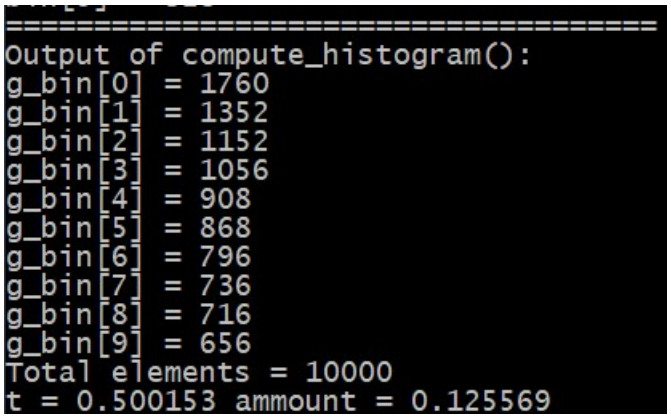
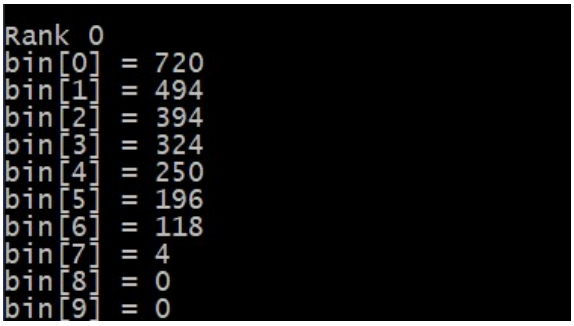


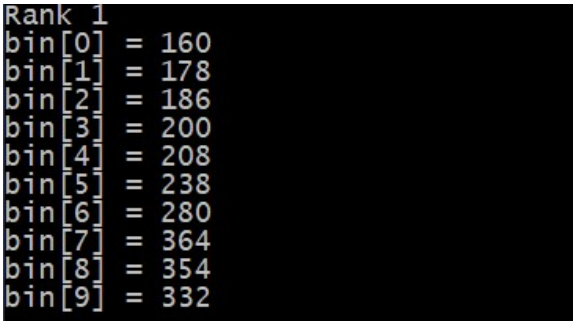
Figure 12. Distribution of the concentrations in the discretized grid at timestep t = 0.5 s.

The figures (a) to (d) below represent the histogram results of each rank (4 ranks used for the computation). As the global discretized grid is distributed into an even number of row chunks that have each the same number of grid points, two histograms are expected to have the same concentration distribution among the histograms.

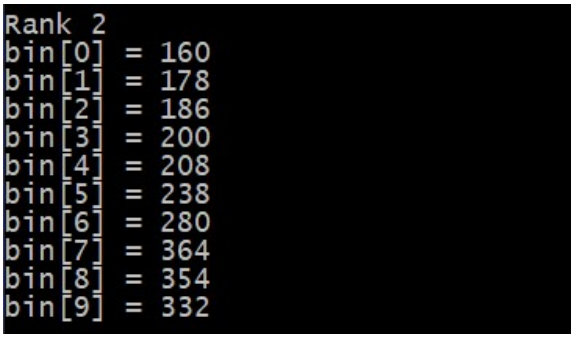
(a)



(b)



(c)



(d)

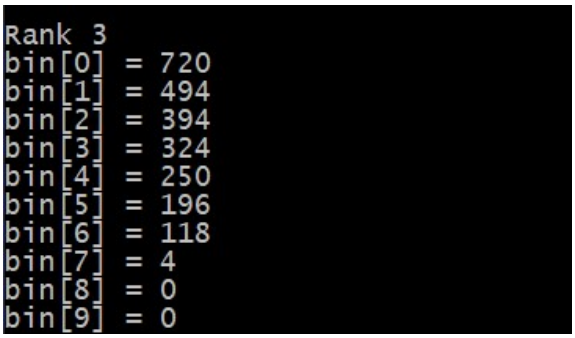


Figure 13. Distribution of the concentrations inside of the local grid areas of each rank. Four ranks were used for the computation. As the global grid is divided into 4 squares row-wise it is expected that rank 0 and 3 as well as rank 1 and 2 show the same results (symmetric problem). The results observed are as expected.

The following figures display the concentration development inside the grid over time and the concentration distribution of the grid at time step t = 0.5 s.

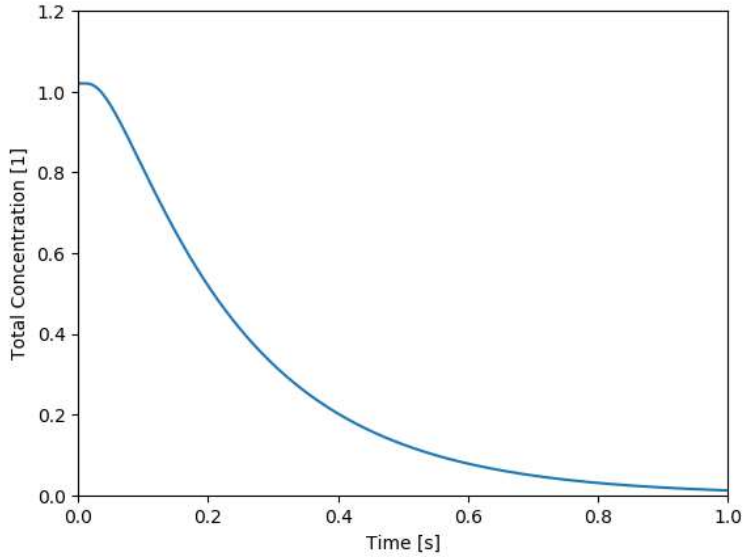


Figure 14. Plot of the total concentration of the square over time. Parameters are D = 1 (diffusion coefficient), L = 2 (side length of the square), and N = 100 (number of grid points).

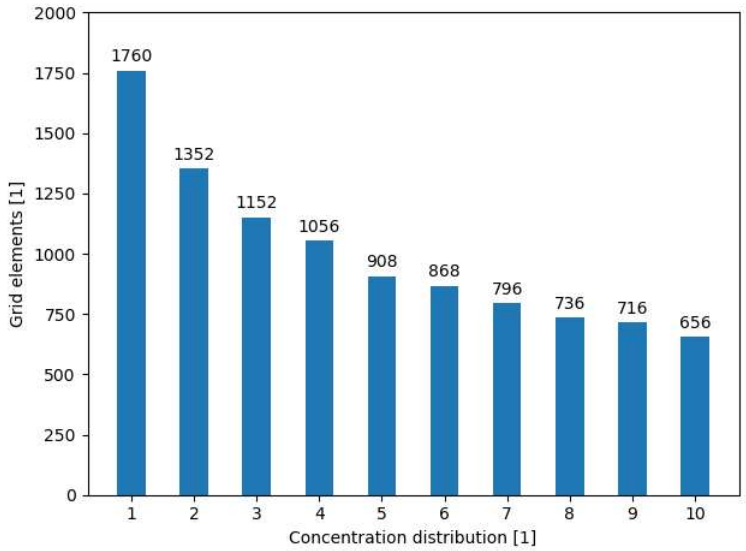


Figure 15. Histogram of the concentration distribution of the whole grid at time step t = 0.5 s. Parameters are D = 1 (diffusion coefficient), L = 2 (side length of the square), and N = 100 (number of grid points).

Question 3: Roll-up of a vortex line

The task of this question was to implement the roll up of a vortex line. For this purpose a set of particles (representing atmospheric particles) was initialized in one dimension. The code then simulates the change of position of the particles in 2D space over time given an initial circulation gamma. The descritized form of the PDE for diffusion was used for the simulation.

Task 1. Serial implementation.

The first task was the implementation of a serial code for the simulation to have a reference for the correct output for the then parallelized code using MPI. The serial implementation can be observed in the following figure.

```
36
37  static void computeVelocities(double epsSq, const std::vector<double>& x,
38                                const std::vector<double>& y,
39                                const std::vector<double>& gamma,
40                                std::vector<double>& u, std::vector<double>& v)
41  {
42      // TODO compute the interactions between the particles and write the result
43      // in the velocity vectors u and v.
44
45      // Every particle must interact with all other particles.
46
47      // Declar and define variables.
48      double xsum, ysum, xi, yi, r;
49      int lim = y.size();
50
51      for (int i = 0; i < lim; i++) {
52          // Storage variables for the updated coordinates of every particle interaction.
53          xsum = 0.0;
54          ysum = 0.0;
55          // Set variables that interact with all other particles.
56          xi = x[i];
57          yi = y[i];
58          // Loop over all particles for the interaction.
59          for (int k = 0; k < lim; k++) {
60              // Descritized form of the PDE for diffusion.
61              r = gamma[k] / (2 * M_PI * (epsSq + (xi-x[k])*(xi-x[k]) + (yi-y[k])*(yi-y[k])));
62              xsum += - (yi-y[k]) * r;
63              ysum += (xi-x[k]) * r;
64          }
65          // Update the velocity values.
66          u[i] = xsum;
67          v[i] = ysum;
68      }
69  }
70
```

Figure 16. Serial implementation of the PDE for diffusion for the roll-up of a vortex line.

Task 2. Parallel implementation using MPI (no communication-computation overlap).

The second task was to implement the simulation using a parallelized code with MPI. The idea was to implement the simulation without a communication-computation overlap. The key point for the MPI implementation is that the different MPI ranks that work on different particles need to obtain the information of all other ranks when the let their "own" particles interact with the ones of other ranks. In order to pass this data around the MPI Allgather function was used, as it is made for exactly this task. A cyclic implementation using for loops is done in task 3 anyway. Due to the fact that MPI Allgather is an MPI built in function the code is certainly more efficient using this function than making use of for loops to pass the data between the different ranks.

Furthermore, it was necessary to use MPI to gather all the information of all ranks in one rank to dump the values of x,y, and gamma to the csv files.

Inside the main() function the data needed to be distributed according to the number of ranks. See code implementation under task 3.

```
119  static void dumpToCsv(MPI_Comm comm, int step, std::vector<double>& x,
120                        std::vector<double>& y,
121                        std::vector<double>& gamma)
122  {
123      int rank, nranks;
124      MPI_Comm_rank(comm, &rank);
125      MPI_Comm_size(comm, &nranks);
126
127      int numParticles = x.size();               // Particles of one rank.
128      int numgParticles = numParticles * nranks;  // Gathered particles.
129
130      std::vector<double> xAll(numgParticles), yAll(numgParticles), gammaAll(numgParticles);
131
132      // TODO Gather the data on rank zero before dumping to the csv files.
133      MPI_Gather(&x[0], numParticles, MPI_DOUBLE, &xAll[0], numParticles, MPI_DOUBLE, 0, comm);
134      MPI_Gather(&y[0], numParticles, MPI_DOUBLE, &yAll[0], numParticles, MPI_DOUBLE, 0, comm);
135      MPI_Gather(&gamma[0], numParticles, MPI_DOUBLE, &gammaAll[0], numParticles, MPI_DOUBLE, 0, comm);
136
137      if (rank == 0)
138          dumpToCsv(step, xAll, yAll, gammaAll);
139  }
140
```

Figure 17. Implementation of the gather of the rank data in order to write the results to a CSV file.

```cpp

static void computeVelocities(MPI_Comm comm, double epsSq,
                              std::vector<double>& x,
                              std::vector<double>& y,
                              std::vector<double>& gamma,
                              std::vector<double>& u, std::vector<double>& v)
{
    // TODO: perform multi pass to compute interactions and update the local
    // velocities.
    // Test if MPI vars are available.
    int rank, size;
    MPI_Comm_rank(comm, &rank);
    MPI_Comm_size(comm, &size);

    // Every rank needs all the data of all other ranks.
    // Use all gather so that the data of every rank is available at every rank.
    // Number of particles worked with.
    int numParticles = x.size();
    int numgParticles = numParticles*size;
    // Data receive storage for gathered values.
    std::vector<double> xg(numgParticles);
    std::vector<double> yg(numgParticles);
    std::vector<double> gg(numgParticles);

    // MPI_Allgather to pass data to every rank.
    MPI_Allgather(&x[0], numParticles, MPI_DOUBLE, &xg[0], numParticles, MPI_DOUBLE, comm);
    MPI_Allgather(&y[0], numParticles, MPI_DOUBLE, &yg[0], numParticles, MPI_DOUBLE, comm);
    MPI_Allgather(&gamma[0], numParticles, MPI_DOUBLE, &gg[0], numParticles, MPI_DOUBLE, comm);

    // Compute velocities of this rank.
    // Declare variables.
    double xsum, ysum, xi, yi, r;
    // Loop over particle values of this rank.
    for (int i = 0; i < numParticles; i++) {
        // Let the particles of this rank interact with all other particles.
        xsum = 0.0;
        ysum = 0.0;
        // Define particles that interact with all other particles.
        xi = x[i];
        yi = y[i];

        for (int k = 0; k < numgParticles; k++) {
            r = gg[k] / (2 * M_PI * (epsSq + (xi-xg[k])*(xi-xg[k]) + (yi-yg[k])*(yi-yg[k])));
            xsum += - (yi-yg[k]) * r;
            ysum += (xi-xg[k]) * r;
        }
        u[i] = xsum;
        v[i] = ysum;
    }
}
```

Figure 18. MPI implementation of the computeVelocities function. The MPI_Allgather() function was used to pass the data of each rank to all other ranks.

Task 3. Parallel implementation using MPI (communication-computation overlap).

Task 3 of question 3 was concerned with a parallel implementation of the simulation using cyclic communication in a non-blocking way. I.e., a rank should be doing useful work while it is receiving information of other ranks.

The code in the figure below shows the use of the MPI function MPI_Isend(), which sends the message and returns immediately from the function so that other wok can be done. A receive buffer on the receiving end takes care of the rest. On the receiving side the MPI function MPI_Irecv() was used as it works in a similar way to the MPI_Isend() function and allows to go ahead in the code while waiting for the data.

In order to achieve the asked communication-communication overlap, each ranks starts computing the interaction of all the particles in its own set of particles while waiting for the data to arrive in the receive buffers. This data is already present in each rank and can therefore be exploited immediately. Additionally, it provides some slack for other data to arrive in the meantime.

After that a for loop iterates over the number of ranks-1 and checks if data has already arrived. In case data is present in the data receive buffer the code execution proceeds and lets all the particles of the current set interact with all particles of the data set received from another rank. This ensures that only data of one rank needs to be available to go ahead with the computation of the discretized form of the PDE for diffusion.

```
1
37
38  static void computeVelocities(MPI_Comm comm, double epsSq,
39                                const std::vector<double>& x,
40                                const std::vector<double>& y,
41                                const std::vector<double>& gamma,
42                                std::vector<double>& u, std::vector<double>& v)
43  {
44      // TODO: perform multi pass to compute interactions and update the local
45      // velocities with non blocking communication
46
47      // Obtain MPI information.
48      int rank, size;
49      MPI_Comm_rank(comm, &rank);
50      MPI_Comm_size(comm, &size);
51
52      MPI_Request sendRequest;
53      std::vector<MPI_Request> requests(size, MPI_REQUEST_NULL);
54      int numRequests = requests.size();
55
56      int numParticles = x.size();
57
58      // Create a vector that stores all the data of one rank.
59      // (xData, yData, gammaData)
60      std::vector<double> allRankData;
61      allRankData.insert(allRankData.end(), x.begin(), x.end());
62      allRankData.insert(allRankData.end(), y.begin(), y.end());
63      allRankData.insert(allRankData.end(), gamma.begin(), gamma.end());
64      // Create a vector that has the according size to receive the messages.
65      std::vector<double> receivedMessages(size * numParticles * 3, 0.0);
66
67      // Send data to the other ranks.
68      // Use MPI_Isend so that the message is sent, but rank returns immediately from the function.
69      int sendVecSize = 3 * numParticles;
70      for (int i = 0; i < size; i++) {
71          if (i == rank)
72              continue;
73          MPI_Isend(&allRankData[0], sendVecSize, MPI_DOUBLE, i, rank, comm, &sendRequest);
74      }
75
76      // Receive data.
77      for (int i = 0; i < size; i++) {
78          if (i == rank)
79              continue;
80          MPI_Irecv(&receivedMessages[i*sendVecSize], sendVecSize, MPI_DOUBLE, i, i, comm, &requests[i]);
81      }
82
83      // Work on data of this rank (interaction of each particle with all other particles).
84      // This can be done while the messages are being received by the above code (computation and
85      // communication overlap).
86      double xsum, ysum, xi, yi, r;
87      for (int i = 0; i < numParticles; i++) {
```

Figure 19. Implementation of the computeVelocities() function using MPI non-blocking and communication-computation overlap (Part 1/2).

```
    // This can be done while the messages are being received by the above code (computation and
    // communication overlap).
    double xsum, ysum, xi, yi, r;
    for (int i = 0; i < numParticles; i++) {
        xsum = 0.0;
        ysum = 0.0;
        xi = x[i];
        yi = y[i];
        for (int k = 0; k < numParticles; k++) {
            // Implementation of the discretized form of the PDE for diffusion.
            r = gamma[k] / (2 * M_PI * (epsSq + (xi-x[k])*(xi-x[k]) + (yi-y[k])*(yi-y[k])));
            xsum += - (yi - y[k]) * r;
            ysum += (xi - x[k]) * r ;
        }
        // Update velocities.
        u[i] = xsum;
        v[i] = ysum;
    }

    // Work on data of other ranks as soon as it is received.
    // (Computation and communication overlap).
    int lim = size - 1;
    for (int i = 0; i < lim; i++) {
        int index;
        // Wait until new data is received.
        MPI_Waitany(numRequests, requests.data(), &index, MPI_STATUS_IGNORE);

        // Loop over all particles of this rank.
        for (int i = 0; i < numParticles; i++) {
            xsum = 0.0;
            ysum = 0.0;
            xi = x[i];
            yi = y[i];
            for (int k = 0; k < numParticles; k++) {
                // index*sendVecSize obtains the position of data of a received rank.
                // + numParticles*x obtains the values of interest (x, y or gamma).
                // + k loop over all these values of interest.
                r = receivedMessages[index*sendVecSize + numParticles*2 + k] / (2 * M_PI * (epsSq +
                    (xi-receivedMessages[index*sendVecSize + k]) *
                    (xi-receivedMessages[index*sendVecSize + k]) +
                    (yi-receivedMessages[index*sendVecSize + numParticles + k]) *
                    (yi-receivedMessages[index*sendVecSize + numParticles + k])));

                xsum += - (yi - receivedMessages[index*sendVecSize + numParticles + k]) * r;
                ysum += (xi - receivedMessages[index*sendVecSize + k]) * r;
            }
            // Update velocities.
            u[i] += xsum;
            v[i] += ysum;
        }
    }
}
```

Figure 20. Implementation of the computeVelocities() function using MPI non-blocking and communication-computation overlap (Part 2/2).

In order to provide each rank with the same amount of work the data set needed to be distributed equally among all ranks in the main function (the precondition of data size % 2 = 0 was set).

```
191  int main(int argc, char** argv)
192  {
193      // Init MPI.
194      MPI_Init(&argc, &argv);
195
196      // Total number of particles must have a modulo 2 of zero.
197      if (argc != 2) {
198          fprintf(stderr, "usage: %s <total number of particles>\n", argv[0]);
199          exit(1);
200      }
201
202      MPI_Comm comm = MPI_COMM_WORLD;
203      int rank, nranks;
204      MPI_Comm_rank(comm, &rank);
205      MPI_Comm_size(comm, &nranks);
206
207      const int nglobal = std::atoi(argv[1]);
208
209      if (nglobal % nranks != 0) {
210          fprintf(stderr,
211                  "expected n to be a multiple of the number of ranks.\n");
212          exit(1);
213      }
214
215      // TODO initialize the data for each rank.
216      // Each rank needs to work on the same number of particles.
217      const int n = nglobal / nranks;              // TODO
218      const double extents = 1.0 / nranks;         // TODO
219      const double startX = -0.5 + extents * rank; // TODO
220      const double endX = startX + extents;
221
222      // Size of a timestep.
223      const double dt = 1e-4;
224      // Small value to prevent a denominator of zero.
225      const double epsSq = 1e-3;
226      const double endTime = 2.5;
227      // Frequency of how often the results are written to a csv file.
228      const double dumpFreq = 0.1;
229
230      // Write results to csv file every 1000 steps.
231      const int dumpEvery = dumpFreq / dt;
232      const int numSteps = endTime / dt;
233
234      // state of the simulation
235      std::vector<double> x(n);
236      std::vector<double> y(n);
237      std::vector<double> gamma(n);
238
```

Figure 21. Implementation of the main() function. The provided data size needs to be distributed equally among the MPI ranks. (Part 1/2).

```
238
239     // workspace
240     std::vector<double> u(n);
241     std::vector<double> v(n);
242
243     // Init. x, y, and gamma.
244     initialConditions(startX, endX, x, y, gamma);
245
246     for (int step = 0; step < numSteps; ++step) {
247         if (step % dumpEvery == 0) {
248             // Write results to a csv file (24 files are created in total).
249             const int id = step / dumpEvery;
250             dumpToCsv(comm, id, x, y, gamma);
251         }
252         // Compute updated velocites u, v.
253         computeVelocities(comm, epsSq, x, y, gamma, u, v);
254         // Go ahead one step in time (dt).
255         forwardEuler(dt, u, v, x, y);
256     }
257
258     MPI_Finalize();
259
260     return 0;
261 }
262
```

Figure 22. Implementation of the main() function. The provided data size needs to be distributed equally among the MPI ranks. (Part 2/2).

Results.

The following figures show some impressions of the results modelled with Paraview. One data set was simulation worked on a data set of 100 particles, the other one worked on a data set of 8,000 points.

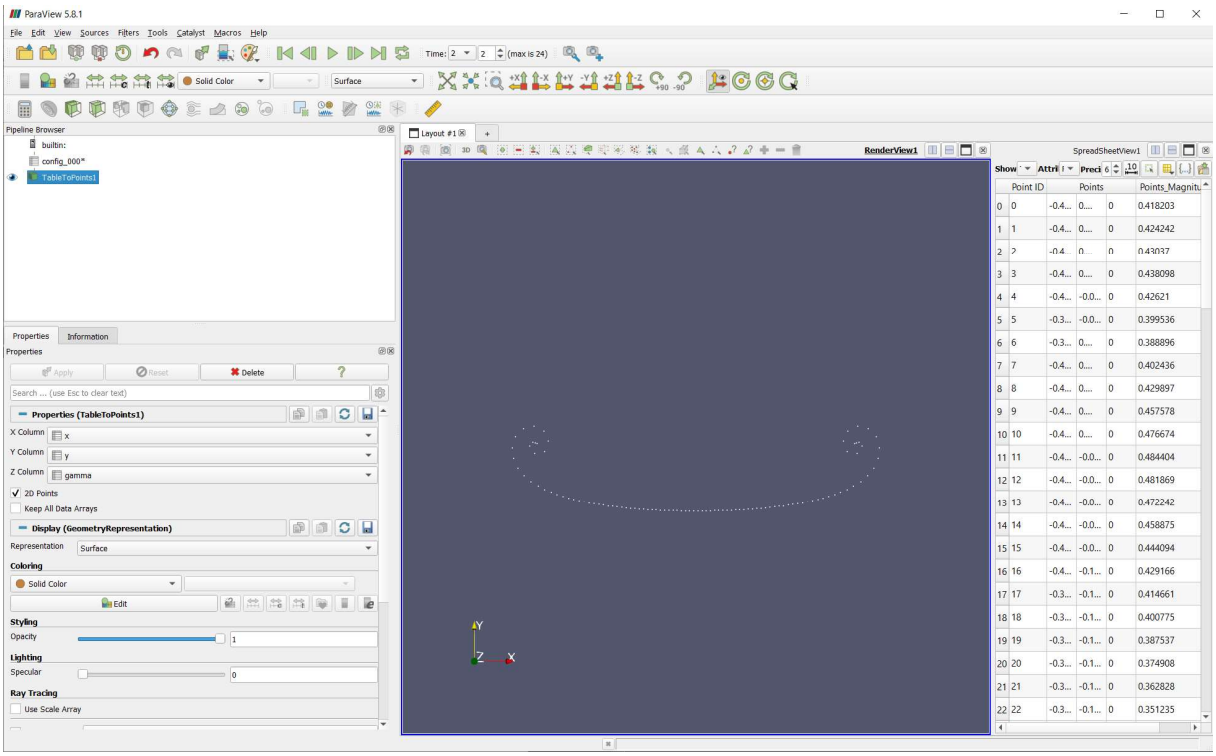


Figure 23. Paraview figure of the results with **100 particles** (Part 1/2).

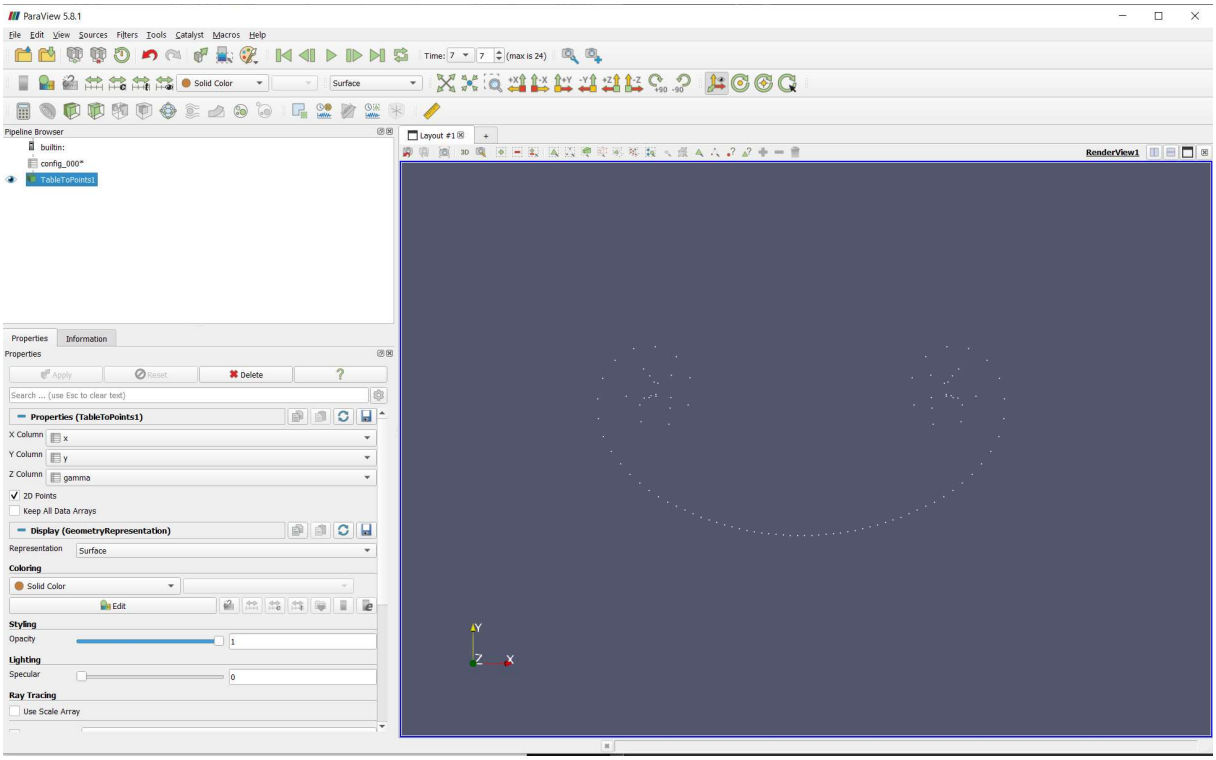
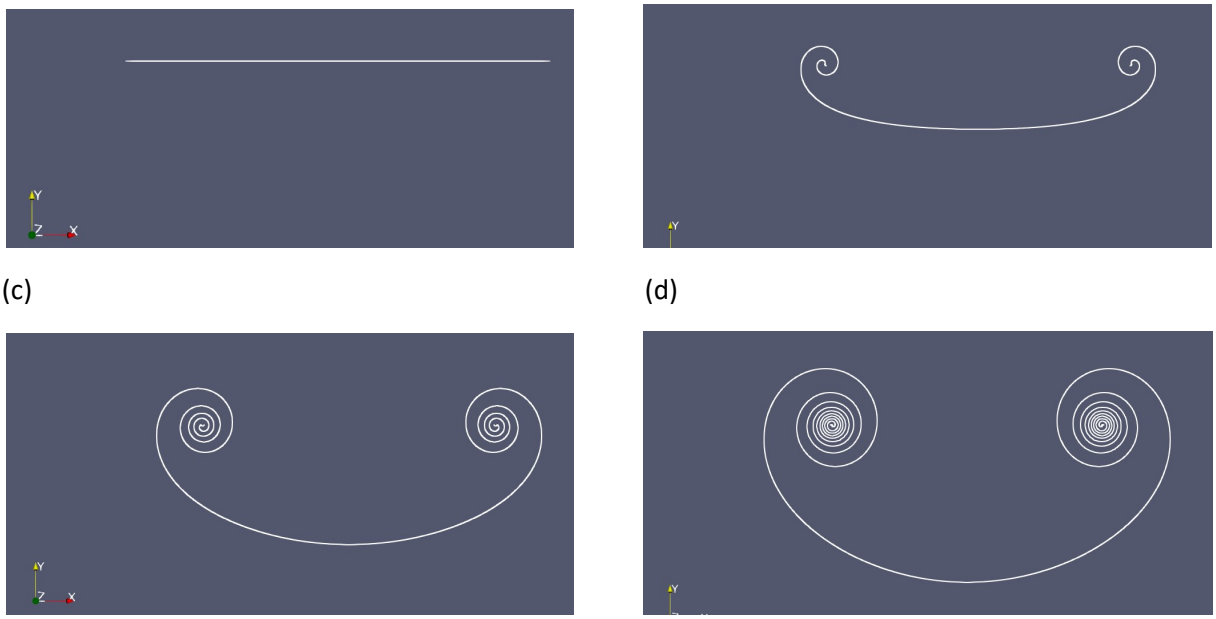


Figure 24. Paraview figure of the results with **100 particles** (Part 2/2).

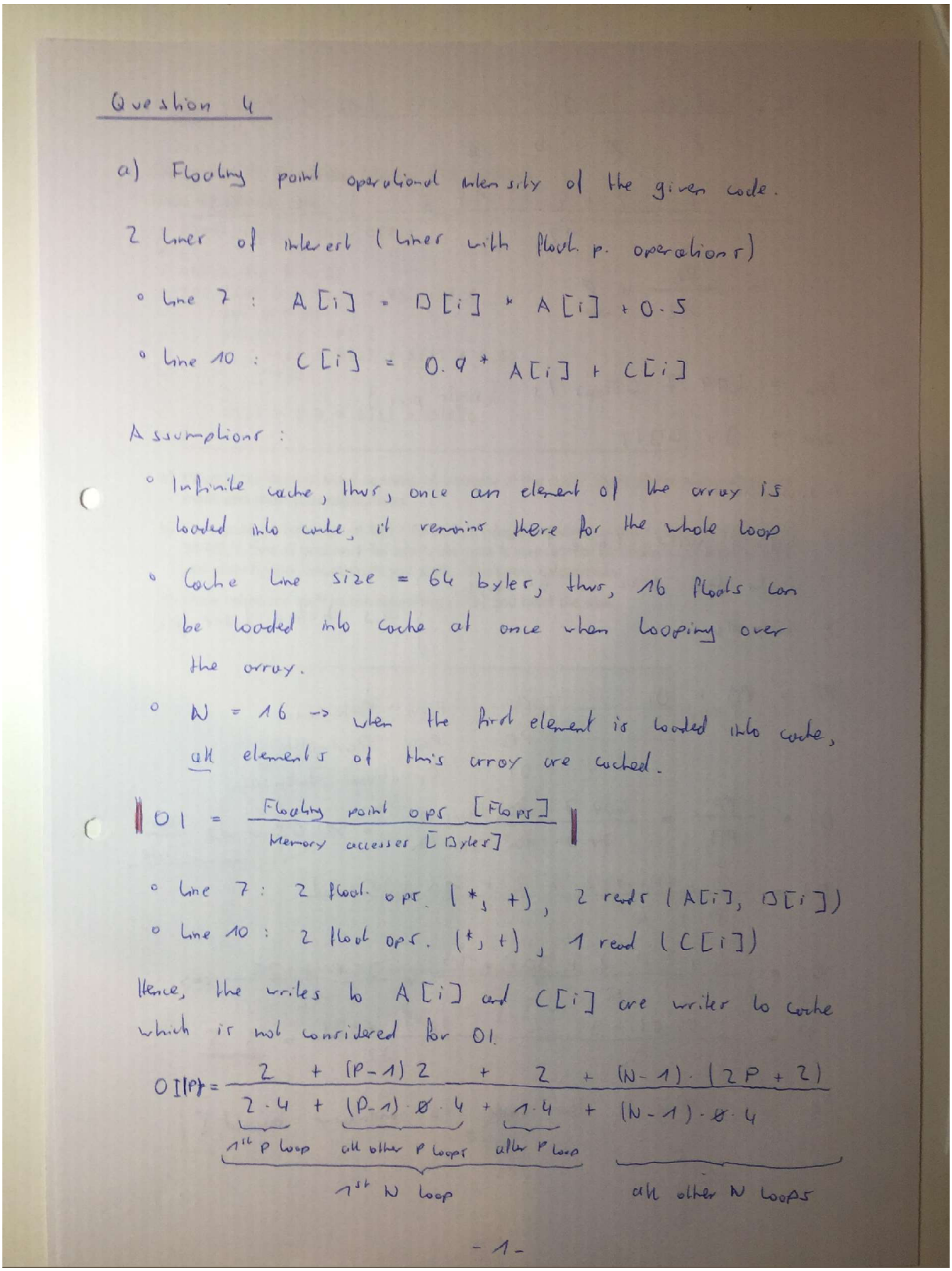
(a) (b)



(c) (d)

Figure 25. Results of the simulation with the MPI non-blocking code using a data set of 8000 points. (a) to (d) show the evolution of the particle positions in space over time.

Question 4: Roofline Model



Question 4

a) Floating point operational intensity of the given code.

2 lines of interest (lines with floating point operations)

- line 7: A[i] = B[i] * A[i] + 0.5
- line 10: C[i] = 0.9 * A[i] + C[i]

Assumptions:

- Infinite cache, thus, once an element of the array is loaded into cache, it remains there for the whole loop
- Cache line size = 64 bytes, thus, 16 floats can be loaded into cache at once when looping over the array.
- N = 16 -> when the first element is loaded into cache, all elements of this array are cached.

$$OI = \frac{\text{Floating point ops [Flops]}}{\text{Memory accesses [Bytes]}}$$

- line 7: 2 floating ops (*, +), 2 reads (A[i], B[i])
- line 10: 2 floating ops (*, +), 1 read (C[i])

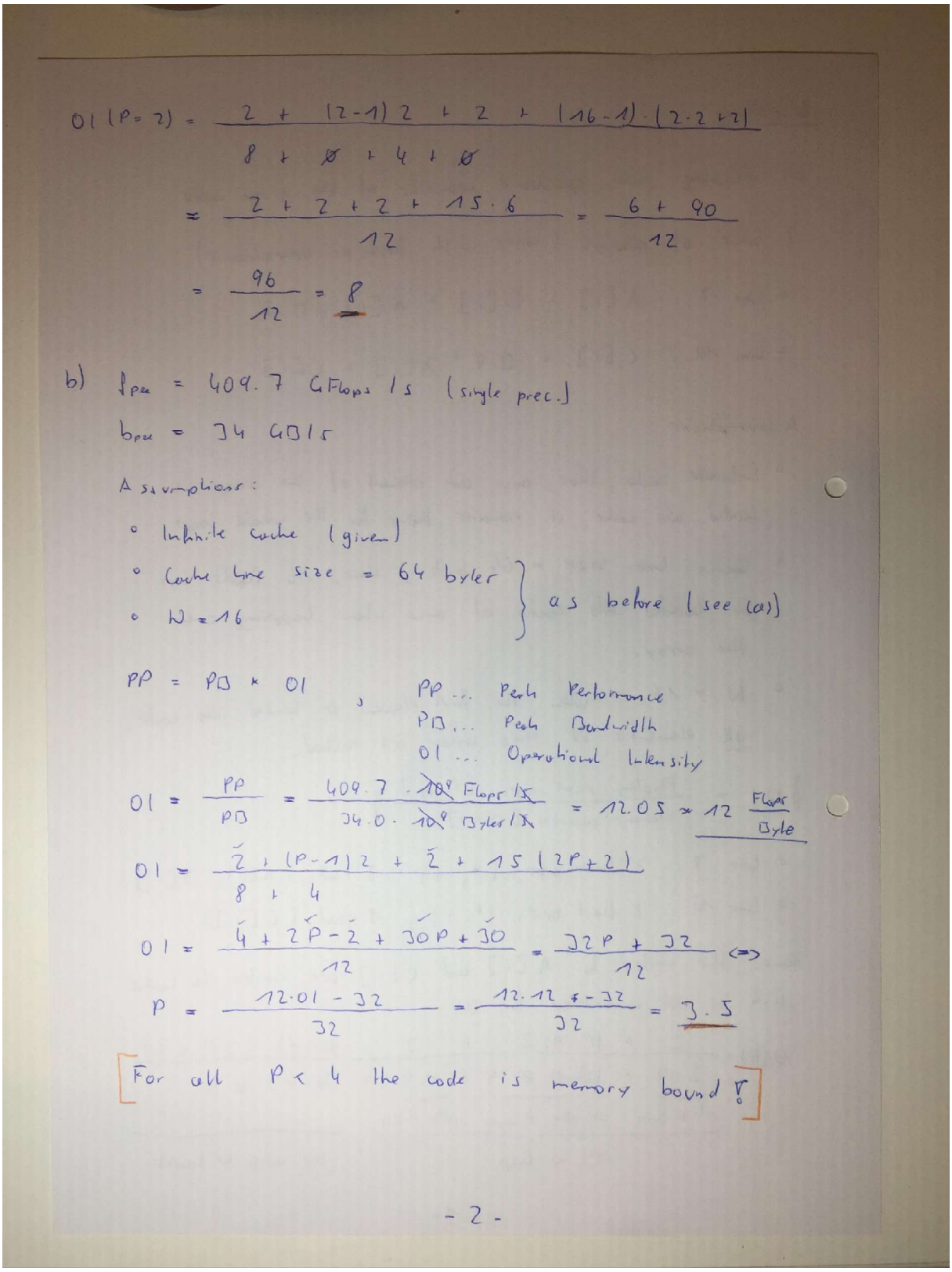
Hence, the writes to A[i] and C[i] are written to cache which is not considered for OI.

$$OI(P) = \frac{2 + (P-1)\,2 + 2 + (N-1)\cdot(2P+2)}{\underbrace{2\cdot 4}_{1^{st}\,P\,loop} + \underbrace{(P-1)\cdot 0\cdot 4}_{all\,other\,P\,loops} + \underbrace{1\cdot 4}_{after\,P\,loop} + \underbrace{(N-1)\cdot 0\cdot 4}_{all\,other\,N\,loops}}$$

1^{st} N loop

- 1 -

Figure 26. Roofline model calculations (Part 1/2).



$$OI(P=2) = \frac{2 + (2-1)\,2 + 2 + (16-1)\cdot(2\cdot 2+2)}{8 + \cancel{0} + 4 + \cancel{0}}$$

$$= \frac{2 + 2 + 2 + 15\cdot 6}{12} = \frac{6 + 90}{12}$$

$$= \frac{96}{12} = 8$$

b) $f_{peak} = 409.7$ GFlops/s (single prec.)

$b_{peak} = 34$ GB/s

Assumptions:

- Infinite cache (given)
- Cache line size = 64 byte
- $N = 16$

} as before (see (a))

$PP = PB \times OI$, PP ... Peak Performance, PB ... Peak Bandwidth, OI ... Operational Intensity

$$OI = \frac{PP}{PB} = \frac{409.7 \cdot \cancel{10^9}\ \text{Flops/s}}{34.0 \cdot \cancel{10^9}\ \text{Byte/s}} = 12.05 \approx 12\ \frac{\text{Flop}}{\text{Byte}}$$

$$OI = \frac{2 + (P-1)\,2 + 2 + 15\,(2P+2)}{8 + 4}$$

$$OI = \frac{4 + 2P - 2 + 30P + 30}{12} = \frac{32P + 32}{12} \Leftrightarrow$$

$$P = \frac{12\cdot OI - 32}{32} = \frac{12\cdot 12 - 32}{32} = 3.5$$

For all $P < 4$ the code is memory bound!

- 2 -

Figure 27. Roofline model calculations (Part 2/2).

Question 4: Roofline Model (30 points)

Given the following code:

```
1  float A[N], B[N], C[N];
2  ...
3  const int P = 2;
4  for (int i = 0; i < N; ++i) {
5      int j = 0;
6      while (j < P) {
7          A[i] = B[i] * A[i] + 0.5;
8          ++j;
9      }
10     C[i] = 0.9 * A[i] + C[i];
11 }
```

a) What is the floating point operational intensity of the code? State all the assumptions you made and show your calculations.

b) For a peak performance of 409.7 GFLOP/s (single precision) and a memory bandwidth of 34 GB/s, find all positive P for which the code is memory bound. Assume an infinite cache and state further assumptions you made. Show your calculations.

c) Draw below the roofline corresponding to (b) and label the axes.

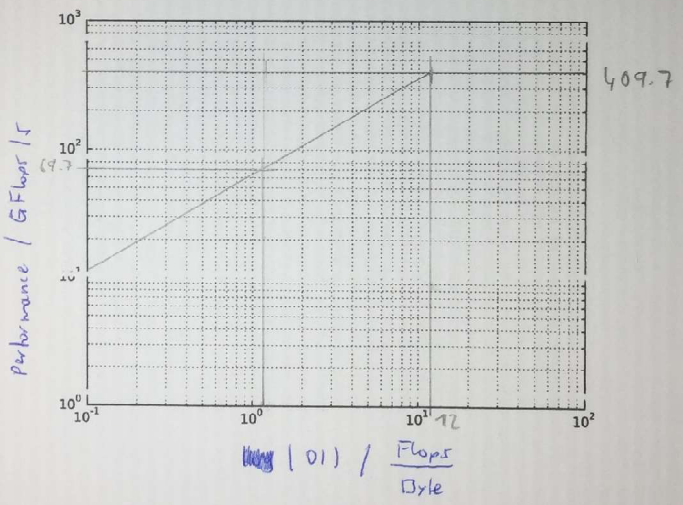


Figure 28. Roofline model sketch.