J. H. Walther
S. Martin
ETH Zentrum, CLT
CH-8092 Zürich

Fall semester 2022

Set 5 - MPI Part II

Issued: November 23, 2022
Hand in (optional): December 7, 2022 08:00

Grading: To get full credits solve **all** of the questions.

Question 1: Diffusion (100 points)

The diffusion of a substance can be described by the equation

$$\frac{\partial c(x,y,t)}{\partial t} = D\left(\frac{\partial^2 c(x,y,t)}{\partial x^2} + \frac{\partial^2 c(x,y,t)}{\partial y^2}\right),$$

where c is the concentration of the substance at position (x, y) and at time t, and D is the diffusion constant. The diffusion process happens in the domain $|x| < L/2$ and $|y| < L/2$. The concentration is zero on the boundaries of the domain. The initial concentration is

$$c(x,y,0) = \begin{cases} 1, & \text{if } |x| < L/4 \text{ and } |y| < L/4, \\ 0, & \text{otherwise.} \end{cases}$$

a) The skeleton code solves the equation on a uniform grid using a central finite difference scheme in space and forward Euler time integration. Parallellize the code by filling parts marked by `TODO` in the functions `advance` and `main`. Use a tiling decomposition scheme (i.e., distribute the rows evenly to the MPI processes). How to run the code:

- `make` to compile the code
- `make run` to run single core (please not in the login node on euler!). If not locally on the laptop, use `sbatch launch_single.sh` to submit a job via slurm.
- to run multicore use: `mpirun -n x ./diffusion D L N`. You can also use `sbatch launch_single.sh` to submit a job via slurm.

The implementation can be found in `solution_code/diffusion.cpp`.

Grading scheme:
Inside `main`:

- 4p: `MPI_Init(&argc, &argv)`
- 4p: `MPI_Comm_rank(MPI_COMM_WORLD, &rank)`
- 4p: `MPI_Comm_size(MPI_COMM_WORLD, &procs)`
- 4p: `MPI_Finalize()`

Inside `advance`:

- 4p: For taking care that the rank handling the *bottom boundary* exchanges data only for the *upper row* of its sub-domain.
- 4p: For taking care that the rank handling the *upper boundary* exchanges data only for the *bottom row* of its sub-domain.
- 4p: For exchanging data on both sides (upper/lower) of sub-domains that are not at the upper/lower domain boundaries.
- 4p: For using the correct MPI datatype (`MPI_DOUBLE`).
- 4p: For *not* exchanging the right and left ghost cells of a row, i.e. exchanging only `N_` elements instead of `N_+2`.
- 8p: For MPI solutions that do not cause deadlocks.

Total: 44pt

b) For a given time compute the integral of $c(x, y, t)$ over the domain (total ammount of the substance). Fill the missing MPI parts in `compute_diagnostics`, and plot the result as a function of time using $D = 1$, $L = 2$ and $N = 100$. When run correctly, the code will output file called `diagnostics.dat`. To plot this data use `python plot_diagnostics.py` (module load python).

The integral of the concentration (density) over the whole domain represents the total mass of the system. The total mass decreases over time because of the boundary conditions: the concentration outside the domain is zero, and since mass diffuses from high to low conentrations, then any mass that was initial in the domain is diffusing towards the boundaries and exiting our domain of interest. The time evolution of the concentration is shown in Fig. 1. The implementation can be found in `solution_code/diffusion.cpp`.

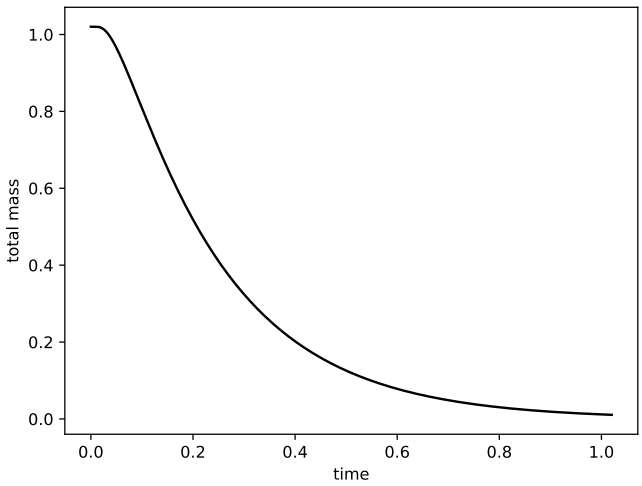

Figure 1: Numerical approximation to the integral of $c(x, y, t)$ over time using $D = 1$, $L = 2$ and $N = 100$.

Grading scheme:

- **6p: For correct use of `MPI_Reduce`**

- **4p: For the correct plot**

Total: 10pt

c) For a given time compute the histogram of $c(x, y, t)$ in the function `compute_histogram` by implementing the missing MPI parts marked by `TODO`, and plot or print the resulting histogram for $t = 0.5$ using $D = 1$, $L = 2$ and $N = 100$.

The histogram using a timestep $\Delta t = 1.0203 \times 10^{-4}$, taken at the step where the time is closest to $t = 0.5$ is shown in Fig. 2.

The implementation can be found in `solution_code/diffusion.cpp`.

Output from the code:

```
 ...
t = 0.499847 ammount = 0.125751
t = 0.499949 ammount = 0.12569
=====================================
Output of compute_histogram():
bin[0] = 1760
bin[1] = 1352
bin[2] = 1152
bin[3] = 1056
bin[4] = 908
bin[5] = 868
bin[6] = 796
bin[7] = 736
bin[8] = 716
bin[9] = 656
Total elements = 10000
dt:  0.00010203
```

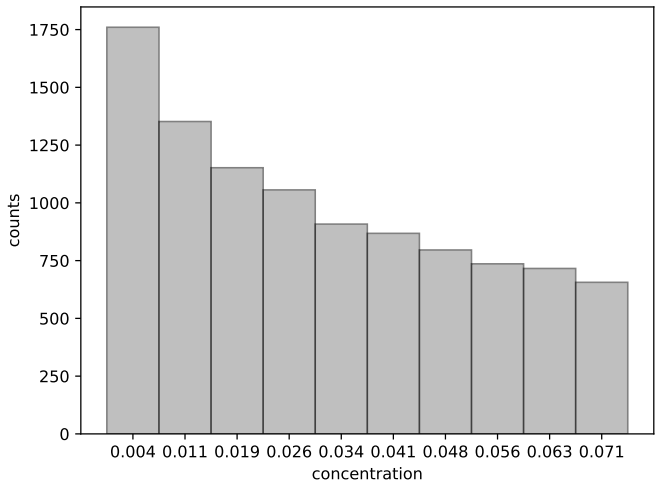

Figure 2: Histogram of $c(x, y, t)$ over the entire domain, at $t = 0.5$ using $D = 1$, $L = 2$ and $N = 100$.

Grading scheme:

- **6p: For correct use of `MPI_AllReduce`**
- **4p: For correct use of `MPI_Reduce`**
- **2p: For the correct plot or the same (± 10) print output**

Total: 12pt

d) Suggest other ways to divide the real-space domain between processes with the aim of minimizing communication overhead. Prove your argument by computing the message communication size for the tiling domain decomposition and for your suggestion.

Assume a square domain of size NxN is split in P rectangular tiles (stripes) along the $y-$ (or $x-$) directions, with one tile per process. Then the maximum number of elements that need to be communicated between neighboring processes is $2N$, since two boundary rows of grid-points need to be communicated to two neighboring processes. This applies for all tiles that are not on the domain boundaries of the domain (since in our exercise, the boundary conditions are not periodic so data do not need to be communicated at the domain boundaries). For the boundary tiles the message communication size is N since only one row of data needs to be communicated.

Another approach is to split the grid into square tiles with one tile per process. The total amount of communication per process if the domain is divided into square tiles is $2[N_x/P_x + N_y/P_y]$ where $N_{x,y}$ and $P_{x,y}$ is the number of grid points and processes in each direction. As we are doing sparse matrix-vector multiplication the computation scales as $N_x N_y / P_x P_y$. If we keep the total grid size N fixed and increase the total number of processes P the computational load per process will decrease as $1/P$. The communication will stay constant for rectangular tiles, but will decrease as $1/\sqrt{P}$ for square tiles. Hence finite difference discretization should scale much better if we use square tiles rather than rectangular.

For the boundaries, instead of sending and receiving on each edge of the sub-domain we can send twice the data on one side per dimension and shift the local grid. The next time-step we send from the opposite side and shift the grid back. This will reduce communication overhead and can have a noticeable effect for a small local domain.

Points distribution (*total 18 points*):

- **6 points for providing an alternative domain decomposition strategy that requires fewer data to be communicated.**
- **6 points for correct computation of the message communication size for the strips domain decomposition.**
- **6 points for correct computation of the proposed strategy's message communication size.**

e) Make a strong and weak scaling plot up to 48 cores. Justify what is happening in your plots. Make at least five different numbers of cores runs (e.g. [1, 12, 24, 36, 48] or [1, 2, 4, 8, 16]). For the strong scaling plot use: N = {1024, 2048, 4096, 8192}, in other words, plot at least four lines (if too slow use smaller N's). For the weak scaling plot, use N =

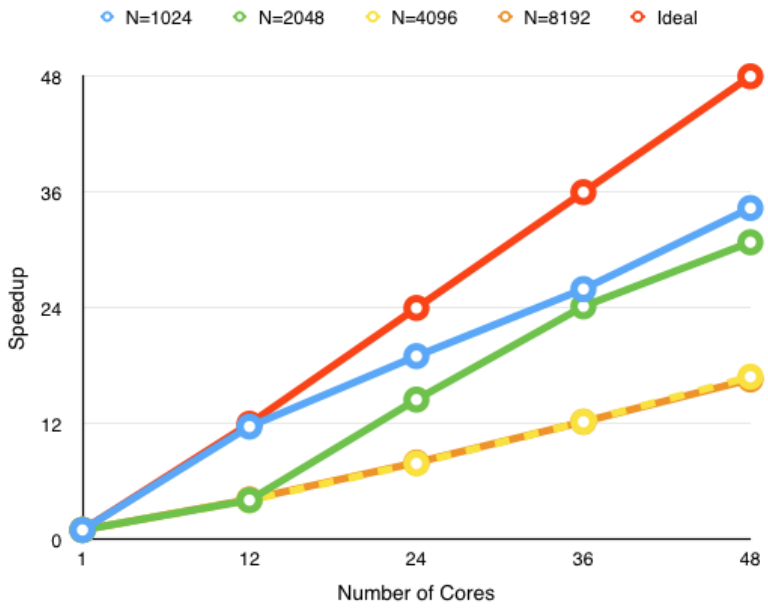

Figure 3: Strong scaling Speedup for 100 time-steps. N^2 is the number of local grid points

1024 and N = 2048 (if those are taking too long use smaller N's). Use D = 1, L = 2 and modify the number of timesteps `step = 100`. Do not forget to state which CPU you ran the tests on! You can use the `run.sh` script to run for different number of cores. On euler use `run.sh` via `sbatch launch.sh`. Draw the plots either directly on the paper (which is the way you will do it on the exam). Or use any desired ploting scripts.

Looking at the strong scaling (Fig. 3), for a total grid size $N = 2^{10}$ the sub-matrix fits entirely into the cache minimizing memory access and resulting in almost perfect scaling. We observe similar behavior for $N = 2^{11}$, as soon as the application utilizes the second NUMA node. As we increase the system size we exceed the cache size and additional memory accesses reduce performance.

In the weak scaling (Fig. 4) the total computational domain per process is kept constant. More specifically, the global grid dimension N_g is equal to $N\sqrt{(P)}$, where P is the number of processes and N is the grid dimension when running on a single core ($P = 1$). The drop in the scaling for 12 cores is attributed to the fact that the problem size fits entirely into the cache for $P = 1$. We also observe that the efficiency remains constant as the number of NUMA nodes increases.

Points distribution (*total 16 points*):

- **4 points for the strong scaling speedup plot for at least five points and four different N.**
- **4 points for the right justification for the strong scaling plot.**
- **4 points for the strong scaling plot.**
- **4 points for the right justification for the weak scaling plot.**

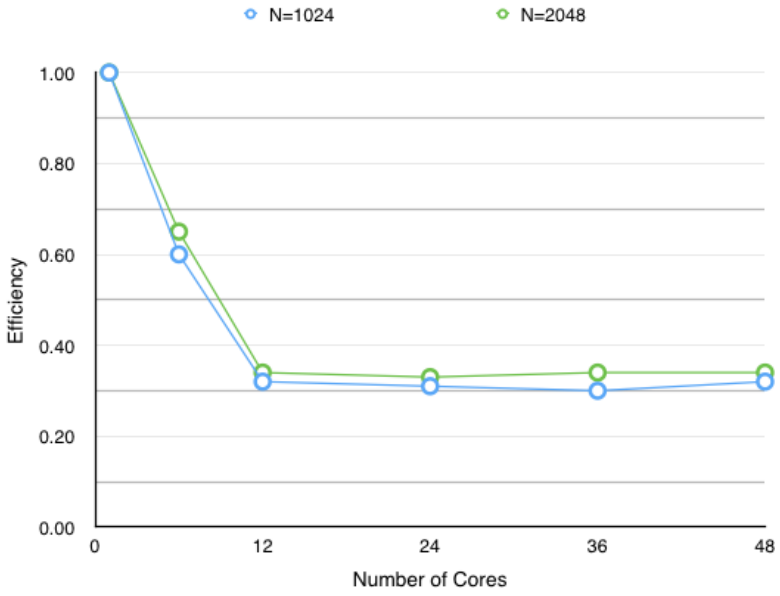

Figure 4: Weak scaling Efficiency for 100 time-steps. N^2 is the number of local grid points