P. Hadjidoukas, E. Dermatas, E. Gallopoulos

Set 1 - MPI and OpenMP

Issued: November 27, 2023

Question 1: Quadratic Form

We want to compute the quadratic form

$$Q = \vec{v}^T \cdot A \cdot \vec{w} = \sum_{ij} v_i A_{ij} w_j \tag{1}$$

in parallel employing both distributed and shared memory parallelism with the hybrid MPI + OpenMP programming model.

For this exercise we consider the following matrix A of size $n \times n$ and the vectors v and w both of size n:

$$A_{ij} = \frac{i+2j}{n}$$
$$v_i = 1 + \frac{2}{i+0.5}$$
$$w_i = 1 - \frac{i}{3}$$

Given the $n \times n$ matrix A and the vectors v, w prepared in the provided skeleton code `quadraticform/skeleton.c` compute the quadratic form of Equation 1 for the target hybrid programming model and put your solution `main_hybrid.c`. Apply the appropriate compiler optimization flags and exploit SIMD vectorization (compiler-based, OpenMP or explicit with SSE/AVX instructions) so as to maximize the performance of your code.

Report the execution time, FLOPs, and the speedup for your implementation for a specific problem parameters and various PxT configurations, where P is the number of MPI processes and T is the number of OpenMP threads (e.g. seq, $1x1$, $1x2$, $1x4$, $2x2$, $4x1$) .

Optionally, you can apply any further optimizations to the code.

Question 2: 2D Diffusion and MPI I/O

Heat flow in a medium can be described by the diffusion equation

$$\frac{\partial \rho(\boldsymbol{r}, t)}{\partial t} = D \nabla^2 \rho(\boldsymbol{r}, t), \tag{2}$$

where $\rho(\boldsymbol{r}, t)$ is a measure for the amount of heat at position $\boldsymbol{r}$ and time t and the diffusion coefficient D is constant.

Let's define the domain Ω in two dimensions as $\{x, y\} \in [-1, 1]^2$. Equation 2 then becomes

$$\frac{\partial \rho(x,y,t)}{\partial t} = D\left(\frac{\partial^2 \rho(x,y,t)}{\partial x^2} + \frac{\partial^2 \rho(x,y,t)}{\partial y^2}\right). \tag{3}$$

Equation 3 can be discretized with a central finite difference scheme in space and explicit Euler in time to yield:

$$\frac{\rho_{r,s}^{(n+1)} - \rho_{r,s}^{(n)}}{\delta t} = D\left(\frac{\rho_{r-1,s}^{(n)} - 2\rho_{r,s}^{(n)} + \rho_{r+1,s}^{(n)}}{\delta x^2} + \frac{\rho_{r,s-1}^{(n)} - 2\rho_{r,s}^{(n)} + \rho_{r,s-1}^{(n)}}{\delta y^2}\right) \tag{4}$$

where $\rho_{r,s}^{(n)} = \rho(-1 + r\delta x, -1 + s\delta y, n\delta t)$ and $\delta x = \frac{2}{N-1}$, $\delta y = \frac{2}{M-1}$ for a domain discretized with $N \times M$ gridpoints.

We use open-boundary conditions

$$\rho(x, y, t) = 0 \quad \forall\, t \geq 0 \,\text{and}\, (x, y) \notin \Omega \tag{5}$$

and an initial density distribution

$$\rho(x, y, 0) = \begin{cases} 1 & |x, y| < 1/2 \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

An MPI implementation of the 2D diffusion equation is provided in `mpiio/diffusion2d_mpi.c`. The parallelization is applied to the routines that initialize and advance the system. The code decomposes the domain using a tiling decomposition scheme, i.e. distributes the rows evenly to the MPI processes. Moreover, `mpiio/diffusion2d_mpi_nb.c` provides an equivalent implementation that uses non-blocking communication to achieve an overlap of data transfer with computation. The `compute_diagnostics` routine computes an approximation to the integral of ρ over the entire domain. The MPI version of the `compute_diagnostics` routine is based on the MPI reduction operation.

An indicative series of snapshots of the simulation in time are shown in the figure.

The provided MPI solutions implement a 2D domain decomposition that divides the square domain of size NxN into P rectangular tiles (stripes) along the $y-$ direction, with one tile per process. 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 to 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.

a) Become familiar with the provided parallel implementations of the 2D diffusion equation.

b) Report the execution time of the code with the non-blocking communication on 1 and 4 processors, for $D = 1$, $L = 1$, $T = 1000$, $dt = 1e - 9$, for $N = 1024$, $N = 2048$ and $N = 4096$. Does the implementation achieve perfect computation-communication overlap? For benchmarking, you must build the execution with `make perf=1`, to disable the computation of diagnostics.

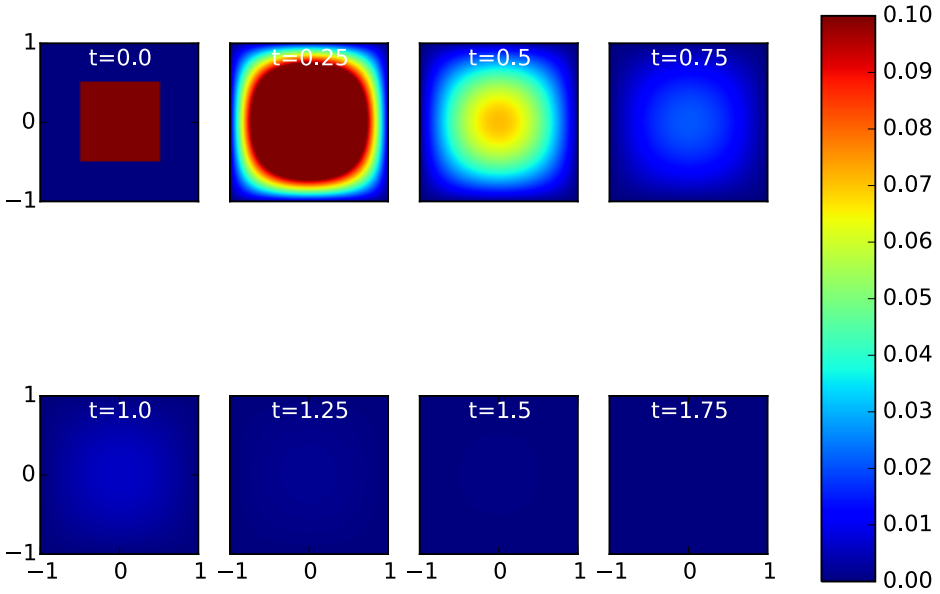

Figure 1: Time evolution of the system for a grid of 128x128 points.

c) In the provided skeleton code `mpiio/diffusion2d_mpi_nb_io.cpp`, you find a slightly extended version of the MPI code for the 2D diffusion problem. The main function includes two functions (`write_density_vis()` and `write_density()`) that are correct and meaningful only when the application runs with a single process, i.e. sequentially. These functions dump the density field to a text and binary file, respectively. Moreover, we provide an empty `write_density_mpi()` function: use MPI I/O in this routine to generate a single **binary** file for all density values. To verify your implementation, you can compare the generated binary file with that produced by `write_density()`. Another approach is to implement a separate sequential code that reads the binary file and computes the sum of the density values.

   A valid set of parameters to use in your experiments is the following: $D = 1$, $L = 1$, $N = 128$, $T = 50000$, $dt = 0.00001$. The corresponding command is:
   `mpirun -n 1 ./diffusion2d_nb_mpi_nb_io 1 1 128 50000 0.00001`
   The number of steps can be lower.

d) Implement the `write_density_mpi_z()` routine, which is a straightforward extension of the `write_density_mpi()` routine and differs only in the part that compresses the local buffer of density values before this is written to the file. How can you verify the correctness of your implementation?

Hints:

- You do not need to store the two rows that contain the ghost data of each rank.
- You can initially implement the mechanism for a single MPI process and regular I/O (binary or not).
- Given that you write only doubles in the output file, you can use a binary reader or the printdata.sh script to check your implementation. Adjust the problem size and the number of ranks to simplify your checks.

- Verify your solution for larger numbers of processes by comparing the output binary file with that generated by the single process experiment (but for the same problem size and number of steps).
- If needed, ask for additional hints for this or any other question of the exercise.

Question 3: MPI bug hunting and asynchronous communication

In the following MPI code, also available in `async/ex01q3.c`, rank 0 distributes a number ($= M * size$) of input values to all ranks of the MPI application. Each input value is processed by the `do_work()` function, which has a uniform but significant execution time.

1. Identify and explain possible issues in the MPI code.
2. Explain how you can address the above issues.
3. Provide a solution code.

```
1  // void do_work(int);
2  // rank: MPI process id
3  // size: number of MPI processes
4
5  int M = 2;  //  any value > 1
6  int input;
7
8  if (rank == 0) {
9      srand48(time(0)); // initialize the random seed
10
11     int N = M*size;
12     for (int i = 0; i < N; i++) {
13         input = lrand48() % 1000;    // some random value
14         MPI_Send(&input, 1, MPI_INT, i%size, 100, MPI_COMM_WORLD);
15     }
16 }
17
18 for (int i = 0; i < M; i++) {
19     MPI_Recv(&input, 1, MPI_INT, 0, 100, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
20     do_work(input);
21 }
```

Hints:

- Make sure your solution does not introduce or imply correctness issues (e.g. race conditions).
- Avoid any assumptions about the communication protocol and the number of MPI processes.
- Try to overlap communication with computation.