High Performance Computing for Science and Engineering I.

Exercise Set 5

Florian Mahlknecht

2019-12-06

1 Gaining intuition in the equations

When considering two vortices Γ_1 and Γ_2 in their respective initial positions:

$$\mathbf{x}_1(0) = (\frac{\Delta}{2}, 0)$$
$$\mathbf{x}_2(0) = (\frac{-\Delta}{2}, 0)$$

we can evaluate the presented equations of the velocity field:

$$u_x(\mathbf{x}_1, 0) = 0 \qquad \iff y = 0$$
$$u_y(\mathbf{x}_1, 0) = \frac{\Gamma_2}{2\pi}\frac{\Delta}{\Delta^2} = \frac{\Gamma_2}{2\pi\Delta}$$
$$u_x(\mathbf{x}_2, 0) = 0 \qquad \iff y = 0$$
$$u_y(\mathbf{x}_2, 0) = -\frac{\Gamma_1}{2\pi}\frac{\Delta}{\Delta^2} = -\frac{\Gamma_2}{2\pi\Delta}$$

Thus, we get for the case in which $\Gamma_1 = \Gamma_2 = \Gamma$:

$$\dot{\mathbf{x}}_1(t) = \begin{pmatrix}0\\1\end{pmatrix}\frac{\Gamma}{2\pi\Delta}t \tag{1}$$
$$\dot{\mathbf{x}}_2(t) = \begin{pmatrix}0\\-1\end{pmatrix}\frac{\Gamma}{2\pi\Delta}t \tag{2}$$

Similarly for $\Gamma_1 = -\Gamma_2 = \Gamma$:

$$\dot{\mathbf{x}}_1(t) = \begin{pmatrix}0\\-1\end{pmatrix}\frac{\Gamma}{2\pi\Delta}t \tag{3}$$
$$\dot{\mathbf{x}}_2(t) = \begin{pmatrix}0\\-1\end{pmatrix}\frac{\Gamma}{2\pi\Delta}t \tag{4}$$

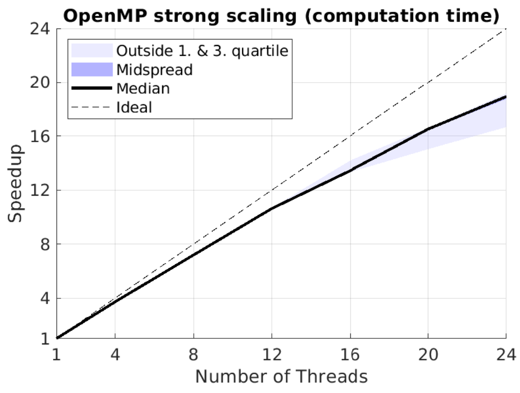
2 OpenMP scaling

The general implementation of the particles iterator is given in listing 1. Note that for the usage in MPI, for convenience a parameter has been added which eventually excludes self interaction, if the applied sources are actually the same.

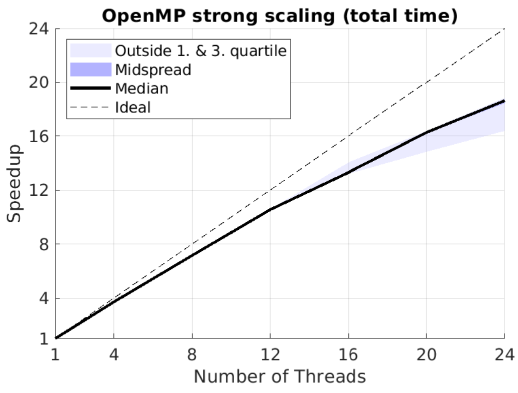
```
void compute_interaction(const ArrayOfParticles & sources,
          ArrayOfParticles & targets, bool excludeSelfInteraction = true)
{
#pragma omp parallel for
  for (size_t j = 0; j < sources.size(); ++j) {
    for (size_t i = 0; i < targets.size(); ++i) {

      if (excludeSelfInteraction && i == j)
        continue; // exclude self interaction

      auto denominator = (SQUARE(targets.pos_x(j) - sources.pos_x(i))
            + SQUARE(targets.pos_y(j) - sources.pos_y(i)));

      targets.vel_x(j) += sources.gamma(i) / (2 * M_PI) *
            (- (targets.pos_y(j) - sources.pos_y(i))) / denominator;
      targets.vel_y(j) += sources.gamma(i) / (2 * M_PI) *
            (targets.pos_x(j) - sources.pos_x(i)) / denominator;
    }
  }
}
```

Listing 1: Serial particles iterator implementation

Figure 1 shows the scale up of the shared memory parallelization. As expected already, the dominant contributor to the total time is the computation itself, which is the reason why there is basically no difference in between figs. 1a and 1b. The performance was measured for $N = 3600$.



(a) Main computation time



(b) Total time

Figure 1: OpenMP strong scaling

With Paraview the results have furthermore been tested for consistency, yielding the expected picture, shown in fig. 2.

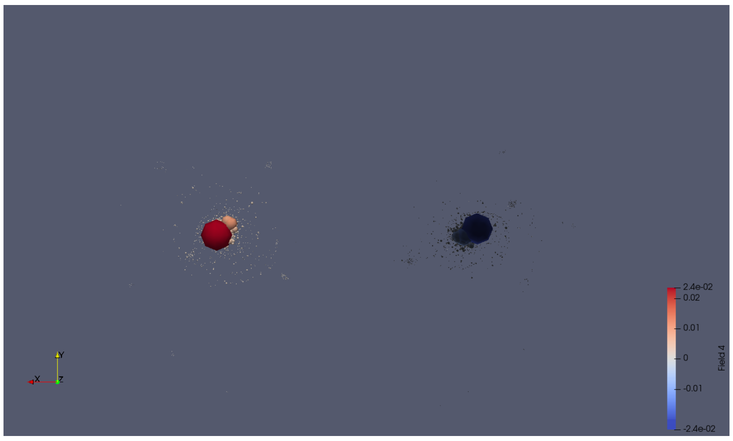
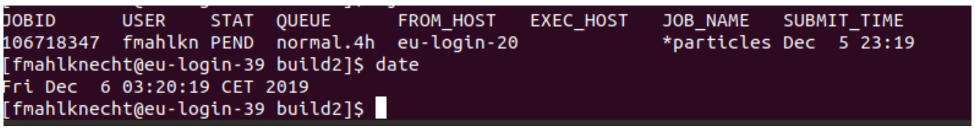


Figure 2: Paraview OpenMP data check

For what regards the weak scaling of the openMP implementation, we need to emphasize the fact the the problem complexity is N^2. This means that a correctly scaled problem for 2 times the number of threads, is $N' = \sqrt{2}\,N$.

Unfortunately, jobs on Euler are not executed in a timely fashion as expected, see fig. 3.



```
JOBID      USER    STAT  QUEUE      FROM_HOST    EXEC_HOST   JOB_NAME   SUBMIT_TIME
106718347  fmahlkn PEND  normal.4h  eu-login-20              *particles Dec  5 23:19
[fmahlknecht@eu-login-39 build2]$ date
Fri Dec  6 03:20:19 CET 2019
[fmahlknecht@eu-login-39 build2]$
```

Figure 3: Euler queue not elaborating tasks in time spans more than 4h

For this reason, a scaled down version has to be executed locally, yielding highly unreliable results, since a large enough problem size would be required in order to make the scaling independent from N.

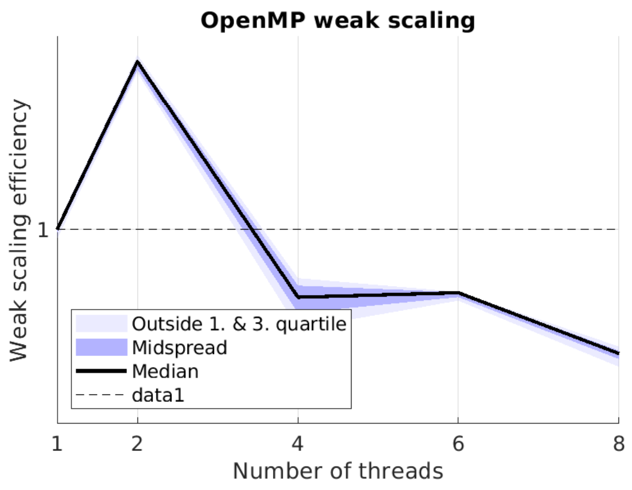


Figure 4: OpenMP Weak scaling (local notebook results in lack of Euler)

This unpredictable setup yields indeed an unexpected peak, being of no further meaning for what regards high performance computing.

3 MPI implementation

The core of the MPI implementation is shown in listing 2.

```
for (int i = 1; i < mpi_size; ++i) {
  auto rankTo = (mpi_rank + i) % mpi_size;
  auto rankFrom = (mpi_rank + mpi_size - i) % mpi_size;

  assert(n_particles == particles.size());

  MPI_Request mpiReqs[6];

  auto &particles_mpi = (i%2==0) ? particles_mpi1 : particles_mpi2;
  auto &particles_calc = i == 1 ? particles : ((i%2==0) ? particles_mpi2
                                                        : particles_mpi1);

  MPI_Isend(particles.pos_x(), n_particles, MPI_VALUE_T, rankTo, //...);
  MPI_Irecv(particles_mpi.pos_x(), n_particles, MPI_VALUE_T, rankFrom, //... );

  MPI_Isend(particles.pos_y(), n_particles, MPI_VALUE_T, rankTo, //...);
  MPI_Irecv(particles_mpi.pos_y(), n_particles, MPI_VALUE_T, rankFrom, //...);

  MPI_Isend(particles.gamma(), n_particles, MPI_VALUE_T, rankTo, //...);
  MPI_Irecv(particles_mpi.gamma(), n_particles, MPI_VALUE_T, rankFrom, //...);

  compute_interaction(particles_calc, particles, i == 1);

  // synchronize to be able to use the buffers again...
  MPI_Waitall(6, mpiReqs, MPI_STATUSES_IGNORE);

  if (i == mpi_size-1) {
    // compute last one directly here
    compute_interaction(particles_mpi, particles, i == 1);
  }
}
```

Listing 2: MPI communication implementation

By using references we can conveniently change the buffers during the iterations without copying in the loops.

For the same inconvenient scheduling reasons as before (see fig. 3), only results from the local notebook are available, with $N = 500$ as a starting point.

Since now the problem is split up into sub-problems for each MPI node, it is important to emphasize that N needs to be dividable without residue by the number of nodes, since for simplicity our implementation relies on this. The numbers for weak scaling have been chosen to be:

- $p = 1, N = 500$
- $p = 2, N = 706$
- $p = 3, N = 870$

- $p = 4$, $N = 1000$
- $p = 8$, $N = 1400$

Note again that the small number of particles was chosen in lack of Euler, to be able to get results on the local notebook, therefore not representable. On the notebook just different processes are handled by the operating system's scheduler. Anyhow the results are shown in fig. 5:

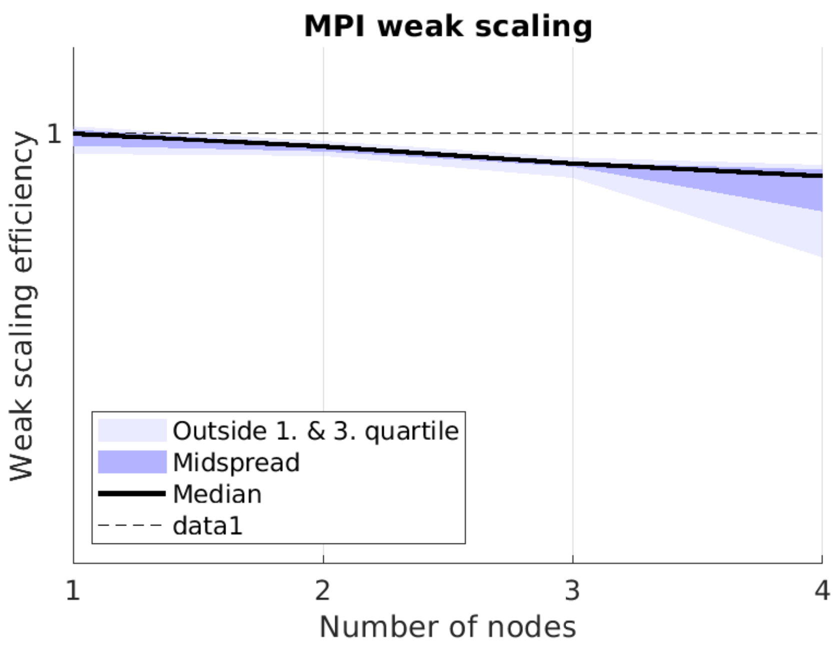


Figure 5: MPI weak scaling (local notebook results in lack of Euler)

Given the unrepresentative nature, strong scaling is not presented.

3.1 Validation

Again the actual output of the simulation has been successfully validated in Paraview, just with fewer numbers of particles due to the computing power constrained. However, consistent MPI communication and results have been achieved, see fig. 6:

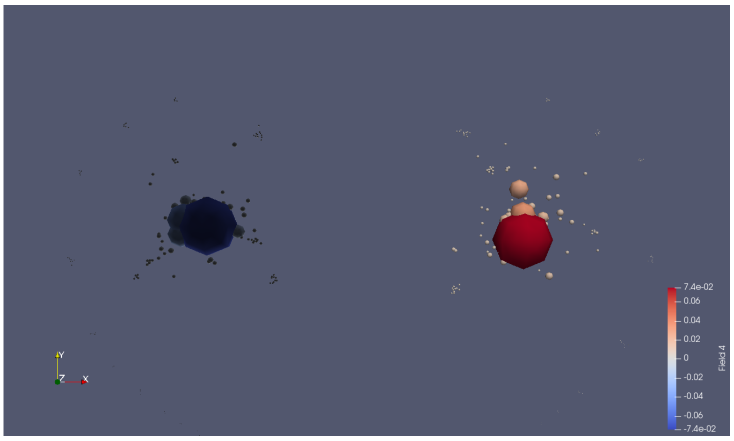


Figure 6: Paraview MPI data check

3.2 Hybrid implementation

For the hybrid implementation (even though not working on Euler), just the previously for OpenMP developed `SerialParticlesIterator_parallel.h` can be used (see code attached). There is no need for OpenMP threads to send MPI messages, so there is no conflict in using just the parallel version to locally calculate the results.

A disadvantage of this straight forward solution is the fact that threads are continuously spawned and joined in the MPI communication loop.

Florian Mahlknecht
fmahlknecht@student.ethz.ch
19-945-351