NTNU
Det skapende universitet

Parallell partikkeltransport implementert i Python med MPI

Simen Mikkelsen

Master i fysikk og matematikk
Innlevert: februar 2017
Hovedveileder: Jon Andreas Støvneng, IFY
Medveileder: Tor Nordam, IFY

Norges teknisk-naturvitenskapelige universitet
Institutt for fysikk

TFY4900 – Master Thesis in Physics

Parallel Particle Transport in Python Using Message Passing Interface

Simen Mikkelsen

Supervisors

Jon Andreas Støvneng
Tor Nordam

February 2017

Department of Physics
Norwegian University of Science and Technology

NTNU
Norwegian University of
Science and Technology

Abstract

A high performance computing program for calculating particle transport in parallel has been written in Python using the SciPy-stack and mpi4py. The 4th order Runge-Kutta method was used as the numerical integrator for calculating particle transport in a velocity field given by an analytical expression, and the movement of each particle was assumed to be independent of other particles. This is a common assumption when considering for example transport of a dissolved chemical in water.

The program was run in parallel on several computers, including the supercomputer Vilje at NTNU and the execution time for different number of processes was measured and compared to each other. The performance was found to be quite well. The speed-up was measured, and performance was found to scale quite well, with super-linear scaling at 8 to 16 ranks for a high number of particles. When using more than one node, the extra execution time because of latency in the communication between nodes outweighed the performance gain in dividing the work on even more processes, for the system sizes tested in this thesis.

In conclusion, it has been demonstrated that the combination of Python and MPI can be used to write scientific source code for parallel simulation of the particle transport problem. The resulting code is quite compact and readable, compared to a similar code in Fortran. A tentative comparison suggests that the performance of the code written in the high-level programming language Python is not significantly worse than the performance of a code written in Fortran.

Sammendrag

Det er skrevet et program i Python for å beregne parallell partikkeltransport. Som numerisk integrator ble fjerdeordens Runge-Kutta-metode brukt for å beregne partikkeltransporten i et hastighetsfelt gitt ved et analytisk uttrykk. Bevegelsen av hver partikkel er antatt å være uavhengig av de andre, en vanlig antagelse ved transport av løste partikler i vann.

Programmet er kjørt i parallell på flere datamaskiner, inkludert superdatamaskinen Vilje ved NTNU. Kjøretiden for ulikt antall av parallelle prosesser ble målt og sammenlignet. Ytelsen ble funnet til å være meget bra. Hastighetsøkningen var bedre enn lineær skalering ved bruk av mellom 8 og 16 parallelle prosesser. Ved kjøring på flere enn 16 prosesser ble imidlertid ytelsen ikke bedre. Dette kommer av at programmet det da må kommuniseres mellom noder, som er tregere enn å kommunisere innad i en node.

Det er demonstrert at kombinasjonen av Python og *Message Passing Interface* kan bli brukt til å skrive vitenskapelig kildekode for parallelle simuleringer av partikkeltransport. Koden er svært kompakt og lesbar, sammenlignet med lignende kode i Fortran. En tentativ sammenligning viser at ytelsen til det mer høynivå programmeringsspråket Python ikke er signifikant dårligere enn til en tilsvarende kode skrevet i Fortran.

Preface

This project is the product of my work in TFY4900, the master project in physics in the last semester of the integrated, five year Master of Technology programme *Physics and Mathematics* at The Norwegian University of Science and Technology (NTNU). The work was carried out during the autumn semester of 2016 and the first seven weeks of 2017.

Originally I planned to write my original master thesis in the Spring semester of 2014, but a period of illness delayed my work. I have been full time employed at Sopra Steria since August 2014. In the Summer of 2016 I got a new supervisor and started with a new thesis. And I have done the project work during a study leave and in the evenings and weekends during my full time job. Therefore the final deadline have been postponed, I am truly grateful for the understanding from the Department of Physics and their cooperation for making my Master Thesis a reality.

Trondheim, Monday 20$^{\text{th}}$ February, 2017

Simen Mikkelsen

Acknowledgment

I would really like to thank my supervisor, Tor Nordam, Associate Professor at Department of Physics, for his encouraging guidance and superb help throughout the period. And I owe my fiancée, Beate Sildnes, and my parents, Berit and Arne Mikkelsen a big thank you for all their support which helped me keep my spirit and always backing me up during my work with this project.

S.M.

Contents

List of Figures

List of Tables

1 Introduction

Particle transport in gases and liquids is a well-known problem with a wide range of applications in physics including passive flow tracers, diffusion and advection. Calculating the movement of particles can be a labour-intensive task if the number of particles is huge. But if the movement of all particles in a given system is independent of each other, the trajectory of each particle can be calculated without taking the other particles into consideration. Parallel computing is a perfect candidate to use for solving problems like this, because the calculation time can be drastically reduced.

We are interested to look into efficient ways to calculate particle transport and concentration in a system. Because particle concentration is a spatial attribute, information about the spatial distribution of particles needs to be determined at a local level equal to the area of interest for the concentration. Calculation of the concentration can be done for several areas at once, as long as the particle distribution is known. The transport of particles can be simplified to a problem where each particle's trajectory is independent of the other particles, so each path can be calculated individually and in parallel. Therefore parallel computing can be used to speed up the calculation of both the concentration and the transport of particles.

In parallel computing a problem is divided into parts and distributed on several processes on a computing system. Each process can work on it's own part of the problem at the same time as the others. This accomplishes shorter execution time for solving the problem, but an increased cost on hardware and writing code that can be parallelised. Also it must be possible to divide a part of the problem into independent parts. There is also an extra cost used to communicate between processes, which needs to be done for solving the part of the problem where for example knowledge of the whole system is required.

For problems involving particle transport two main methods of dividing the problem to separate processes exist. Either each process can be assigned to specific particles, or a process can be assigned to all particles in given areas. These two methods have different impact on the transportation part and the calculation of the concentration part of the problem. In the first method particles are never sent from one process to another in the transport step. I.e. there is no communication of information about particles between processes. But communication is necessary when computing the concentration if the particles in the area of interest belong to different processes. In the last method a process needs to send a particle to another process when the particle moves to an area belonging to another process.

So there may be need for communication work in the transport step. But it is faster to calculate the concentration, because communication between processes is only necessary when calculating the concentration near the border between areas belonging to different processes.

Supercomputers refer to a computer-system with high level of computational capacity. Computer power is aggregated over several nodes, usually each node have multiple processors which often have several cores. Supercomputers deliver much higher performance than ordinary computers, so it is called High Performance Computing (HPC). To benefit from the supercomputers' high capacity, the problem solving should be done in parallel. In practice all computing in HPC are done in parallell.

Traditionally low-level compiler programming languages, such as Fortran or C, are the dominant programming languages used in HPC, due to their superior advantages in speed and memory usage. But interpreted languages, like Python, is popular due to reduced development time, faster prototyping and less complex code. The SciPy-stack[1], including NumPy, SciPy, Matplotlib et cetera, makes precompiled high performance libraries, like MKL (Intel Math Kernel Library)[2] and LAPACK (Linear Algebra PACKage)[3], available in Python. Lately, approximately in the last five years, Python for HPC has received more attention. At the world largest HPC-conference *Supercomputing*[4], they have hold a Python HPC workshop each year since 2011[5]. An example of the use of Python in HPC can be seen in [1], where a computational fluid dynamics (CFD) solver was written in Python, using mpi4py and Numpy, and achieved comparable performance to a similar solver in a compiled language.

To compute the transport of particles affected by a velocity field a numerical iterative method have to be used, however, we will not focus on the numerical methods in this thesis. In the present project we shall implement a parallel particle transport algorithm in Python using mpi4py[6] for parallelisation and the 4th order Runge-Kutta method to calculate the transportation of particles. We shall divide particles on processes, called ranks in mpi4py, like method two described above, i.e. each rank are given specific areas and a particle belongs to a process based on the area it reside in. We will examine the performance of this algorithm with different parameters, including number of particles and how areas are assigned to

[1] www.scipy.org/stackspec.html
[2] https://software.intel.com/en-us/intel-mkl
[3] www.netlib.org/lapack/
[4] http://sc16.supercomputing.org/
[5] www.dlr.de/sc/pyhpc2016
[6] http://pythonhosted.org/mpi4py/

processes. We will also compare the execution time between a desktop computer and a supercomputer.

2 Theory

In this chapter theory central to this project is presented. This includes mathematical and physical foundations, numerical analysis and high performance computing.

2.1 Fluid mechanics

2.1.1 Navier-Stokes equations

The Navier-Stokes equations govern the motion of viscous fluids and gases. In the case of a compressible Newtonian fluid, i the equation reads

$$\rho\left(\frac{\partial \mathbf{v}}{\partial t} + \mathbf{v}\cdot\nabla\mathbf{v}\right) = -\nabla p + \nabla\cdot\mathbb{T} + \mathbf{f}. \tag{1}$$

Here $\mathbf{v}$ is the advection, also referred to as fluid velocity or drift, p is the fluid pressure, ρ is the fluid density, $\mathbb{T}$ is the deviatoric stress tensor, which has order two, and $\mathbf{f}$ represents an external force acting on the continuum, for example gravity, an electric field acceleration and so on.

Equation (1) represents the conservation of momentum, while the continuity equation represents the conservation of mass. With no sources or sinks in the area of interest the continuity equation reads

$$\frac{\partial \rho}{\partial t} + \nabla\cdot(\rho\mathbf{v}) = 0. \tag{2}$$

2.1.2 Advection and diffusion

Particle transport can be modelled by tracking matter as numerical particles moving through a velocity field, frequently pre-calculated by some external source. Such a particle has a number of properties, including position and mass. Possibly the simplest model is to use the point particle, this method has an advantage of greatly simplifying the calculations, and at the same time it is an accurate model if the number of particles is sufficiently large.

For non-interacting particles transport by advection can be computed individually for each particle, the problem is trivially parallelisable because the movement of one particle is independent of all other particles, so there does not need to be

any communication between the processes when calculating the movement of the particles.

The advection-diffusion equation reads

$$\frac{\partial c}{\partial t} = \nabla \cdot (D\,\nabla c) - \nabla \cdot (\mathbf{v}c), \tag{3}$$

where $c(\mathbf{x}, t)$ is the concentration, $D(\mathbf{x}, t)$ is the diffusion coefficient and $\mathbf{v}(\mathbf{x}, t)$ is the advection. For constant diffusivity Eq. (3) is equivalent to an ensemble of random walkers [2] with step lengths ∂x and ∂t given by

$$D = \frac{(\partial x)^2}{2\,\partial t}$$

and with a drift, given by $\mathbf{v}(\mathbf{x}, t)$, in the limit

$$(\partial x)^2 \to 0, \partial t \to 0,$$

taken such that D is finite.

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \mathbf{v}\partial t + \partial x \boldsymbol{\xi}, \tag{4}$$

where $\boldsymbol{\xi}$ is a vector with length 1, and random orientation. In this project we look at advection without diffusion, so $D = 0$ and $\mathbf{v} \neq 0$. Hence, we consider particles moving in a velocity field with no random walk contribution.

2.2 Numerical integration

The problem of particles moving with a velocity field, $\mathbf{v}(\mathbf{x}, t)$ amounts to solving the ordinary differential equation (ODE)

$$\dot{\mathbf{x}} = \mathbf{v}(\mathbf{x}, t). \tag{5}$$

To solve numerically, introduce discrete time

$$t_n = t_0 + nh, \tag{6}$$

where h is called time step and for convenient notation

$$\mathbf{x}_n = \mathbf{x}(t_n). \tag{7}$$

The Euler method is a first-order numerical procedure for solving ODEs with given initial values. It is given by

$$\mathbf{x}_{n+1} = \mathbf{x}_n + h\mathbf{k}, \qquad \mathbf{x}(t_0) = \mathbf{x}_0. \tag{8}$$

Where $\mathbf{k} = \mathbf{f}(t_n, \mathbf{x}_n)$ and $\mathbf{x}(t_0) = \mathbf{x}_0$ is the initial value.

The Euler method is the simplest example of a class of methods known as Runge-Kutta methods (see p. 128 in [4]) and it is a first-order method where the global error, the total accumulated error, is proportional to the time step size, h.

The 4th order Runge-Kutta method is the most known member of the Runge-Kutta methods, as in a 4th order method the global error is in the order of $O(h^4)$.

The 4th order Runge-Kutta method is given by

$$\mathbf{x}_{n+1} = \mathbf{x}_n + \tfrac{h}{6}\left(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4\right), \qquad t_{n+1} = t_n + h, \tag{9}$$

where

$$\begin{aligned}
\mathbf{k}_1 &= f(t_n, \mathbf{x}_n), \\
\mathbf{k}_2 &= f(t_n + \frac{h}{2}, \mathbf{x}_n + \frac{h}{2}\mathbf{k}_1), \\
\mathbf{k}_3 &= f(t_n + \frac{h}{2}, \mathbf{x}_n + \frac{h}{2}\mathbf{k}_2), \\
\mathbf{k}_4 &= f(t_n + h, \mathbf{x}_n + h\mathbf{k}_3).
\end{aligned} \tag{10}$$

The advantages of the 4th order Runge-Kutta method include the possibility of taking longer steps, so the total calculation work can be lowered, reducing the execution time, even if it requires more function evaluations per step.

The step size can be dynamic, so it is smaller when accuracy is required and larger elsewhere, to decrease the amount of steps, and thus execution time. But this is not a focus in this project.

2.3 Velocity field

In a particle transport application, the velocity field, $\mathbf{v}$ in Eq. (1), can frequently come from some external source, e.g. from an ocean circulation model, from (CDF) or from measurements. In all cases the assumption is that the presence of the particles does not affect the velocity field.

The velocity field used in this project is a double gyre model consisting of two counter-rotating vortices with a harmonically oscillating line in between [7]. This system represents a rotating fluid, and the velocity field gives the velocity of an infinitesimal fluid element at a given time and position. Movement of point particles in this system satisfy the non-autonomous dynamical system defined by the velocity field in the xy-plane, defined in the region $x \in [0, 2], y \in [0, 1]$, given by:

$$\begin{aligned} v_x &= -\pi A \sin(\pi f(x,t)) \cos(\pi y) \\ v_y &= \pi A \cos(\pi f(x,t)) \sin(\pi y) \frac{\partial f(x,t)}{\partial x}, \end{aligned} \tag{11a}$$

where

$$\begin{aligned} f(x,t) &= a(t)x^2 + b(t)x \\ a(t) &= \epsilon \sin(\omega t) \\ b(t) &= 1 - 2\epsilon \sin(\omega t). \end{aligned} \tag{11b}$$

The parameters A, ϵ and ω are chosen to fit to the properties of the system. In this project the values of the parameters are $A = 0.1$, $\epsilon = 0.25$ and $\omega = 1$. One property of the velocity field is that all values of $v_x(t)$ and $v_y(t)$ equal zero at the boundaries of the region.

This system represents an incompressible fluid, completely filling the 2D box given by $0 \leq x \leq 2,\ \ 0 \leq y \leq 1$. For this system, $\dfrac{\partial \rho}{\partial t} = 0$, and $\nabla \cdot \mathbf{v} = 0$, so Eq. (2) is satisfied.

2.4 Parallel computing

A common method used in parallelisation of scientific computations is *Single Instruction, Multiple Data (SIMD)*, also called *Single Program, Multiple Data (SPMD)*. The concept can be described by executing equal instructions on different data in parallel. In our case the algorithm for transport of particles in the velocity field is the same everywhere, but particles on which the algorithm are used are split in two or more groups based on position.

For parallelisation of scientific source code, two widely used methods are *Shared Memory Parallelisation* and *Message Passing*. *Shared Memory Parallelisation* benefits from being easy and fast to implement, e.g. a for-loop can be parallelised with minimal changes in the source code if each iteration of the loop is independent

of the others. But a drawback is that shared memory is required, all processors must have access to the shared memory, which means that it only works on one single machine, or node with shared memory, at a time, without communication to other nodes without shared memory. OpenMP is an example of software used for *Shared Memory Parallelisation*.

An example[7] of OpenMP used in Python in conjunction with Cython. Here you can see the small changes needed in the source code starting at line 10, including allocating memory for the for-loop at line 15.

```
1  from cython.parallel import prange, parallel
2  cimport openmp
3
4  def testpar(int n):
5      cdef int i,numthreads
6      cdef int * squared
7      cdef int * tripled
8      cdef size_t size = 10
9
10     with nogil,parallel(num_threads=4):
11         numthreads = openmp.omp_get_num_threads()
12         squared =  malloc(sizeof(int) * n)
13         tripled =  malloc(sizeof(int) * n)
14
15         for i in prange(n,schedule='dynamic'):
16             squared[i] = i*i
17             tripled[i]= i*i*i + squared[i]
18
19         free(squared)
20         free(tripled)
```

For more complicated scientific source code, *Message Passing* is more frequently used because of the important advantage gained from the possibility to communicate with other nodes through a network or interconnect with very high throughput and very low latency, e.g. InfiniBand[8]. Disadvantages of *Message Passing* include longer development time, because it requires larger modification in the source code which leads to more complicated code, and explicit memory management and communication.

[7] https://goparallel.sourceforge.net/parallel-processing-with-pytho
[8] http://www.infinibandta.org/

2.4.1 Load balancing

When parallelising a task there is little gain in total performance if the work load is not distributed equally so that one process does most of the work. Load balancing aims to optimize resource use, maximize throughput, minimize response time, and avoid overload of any single resource. Good load balancing means that all processors do approximately the same amount of work. Otherwise the algorithm can be slower than necessary if several processors are idle while waiting for the one with the most work to finish.

In this case of particle transport, we can assume that the amount of work per particle per time step is constant. Then, the ideal case with perfect load balancing would be if all processors have the same number of particles when calculating the transport of particles. In our case the physical area of the problem is divided into several cells and each cell is assigned to a processor. The shape of these cells does affect the load balancing because it determines whether the particles are distributed evenly.

In order to investigate load balancing, we define the following metric:

$$P_{\mathrm{LB}} = \frac{N_{p_{max}}}{\overline{N_p}} \tag{12}$$

where for a given time, t_n, $N_{p_{max}}$ is the maximum number of particles assigned to a single rank and $\overline{N_p}$ is the average, simply $\dfrac{N_p}{\text{number of ranks}}$.

2.5 Measuring parallel computing performance

When examining how much benefit parallelisation of a problem yields, we must separate the part of the problem which can be divided into independent parts and solved in parallel, and the part which can not be divided and must be solved sequentially in serial. In general, the greater the ratio between the parallelisable part and the serial part is, the larger the benefit gain achieved by using parallel computing is. The advantages of parallel computing, including faster execution time, allowing larger problems to be solved or get more exact solution, often outweigh the disadvantages. Disadvantages include increased hardware cost and longer development time, because writing and executing parallel programs are more complex than serial programs. The increased time and cost of development is not discussed in detail in this thesis.

There are several ways to measure the performance of parallel computing. The most basic one is the price/performance ratio, which come into consideration when deciding the hardware needed for a task. It is usually measured by dividing the cost of the parallelisation, with respect to increased development time and higher hardware cost, on the elapsed run time after parallelisation. For scientific purposes the speed-up and efficiency are a more qualitative measure. The speed-up of a program, S, can be measured by comparing the execution time when running the program in parallel to the execution time on a system which run the program in serial. It is given by dividing the original execution time, $T(1)$, on one processor, by the execution time using p processors, $T(p)$, as

$$S = \frac{T(1)}{T(p)}. \tag{13}$$

Usually only one version of the parallel algorithm is written and then being run using one processor to measure $T(1)$ and p processors to measure $T(p)$. Ideally a separate serial algorithm should be used when measuring $T(1)$, because the serial algorithm usually have a slightly better performance, but this is in general not important.

Efficiency, ϵ, is related to the price-performance ratio. It can be defined as

$$\epsilon = \frac{T(1)}{pT(p)} = \frac{S}{p}, \tag{14}$$

where p is the number of processors. An efficiency close to unity means that the parallel computing scales well and the hardware are used effectively, a low efficiency means the resources are not used to their fullest potential. The speed-up is discussed in the next sections, where also a different, and more useful, metric is presented.

2.5.1 Amdahl's law

Amdahl's law [5] describes the maximum theoretical speed-up a given task can achieve if a system that is solving a problem with fixed size is given improved resources, usually by increasing the number of processors. Such a task will have a sequential part that does not benefit from the improved resources and therefore must be run in serial, with execution time T_s, and a part that benefits from the improvements and can be run in parallel, a parallelisable part using time $T_p(p)$, dependent on the number of processes. Amdahl's law does not take into account that real world problems usually have serial overhead, including increased workload

when they are solved on several processors because of communication between processes also take time, hence it gives a maximum speed-up. The execution time for the whole task before the improvement of resources is denoted T and is given by

$$T = T(1) = T_s + T_p(1). \tag{15}$$

The fraction of work done in the parallelisable part is given by $f = T_p(1)/T$ and the speed-up of the parallelisable part, given the improved resources, is given by the factor $s(p) = T_p(1)/T_p(p)$. The theoretical execution time of the whole task is given by

$$T(p) = T_s + T_p(p) + \kappa = (1 - f)T + \frac{f}{s(p)}T + \kappa, \tag{16}$$

where κ is the parallelization overhead, which we will assume to be negligible. In the case of a problem with fixed size W, the theoretical speed-up can be expressed

$$S(s) = \frac{TW}{T(p)W} = \frac{1}{(1 - f) + \frac{f}{s(p)}}. \tag{17}$$

This equation is Amdahl's law. It is limited to problems with fixed size.

2.5.2 Gustafson's law

In practice better resources will often be used to solve a larger problem or solve a problem more accurately in the same time as the original, instead of solving it faster with the same accuracy. Gustafson's law gives a measure for this. Gustafson's law gives the expected speed-up for a system which is given improved resources

$$S(s) = 1 - f + sf, \tag{18}$$

where S is the theoretical speed-up of the execution of the whole task, s is the speed-up of the execution of the part that benefits from the improvement of resources, the parallelisable part, and f is the ratio of workload of the parallelisable part.

2.5.3 Karp–Flatt metric

In 1990 Karp and Flatt [6] introduced a new, improved metric for measuring the performance of parallel algorithms. It does take into account serial overhead. Amdahl's law in it simplest form can be written as

$$T(p) = T_s + \frac{T_p}{p}. \tag{19}$$

Let e be the fraction of the serial part of the whole problem, $e = T_s/T(1) = T_s/T$.

Then

$$T(p) = T_e + \frac{T(1-e)}{p}, \tag{20}$$

and from the definition of speed-up from Eq. (13),

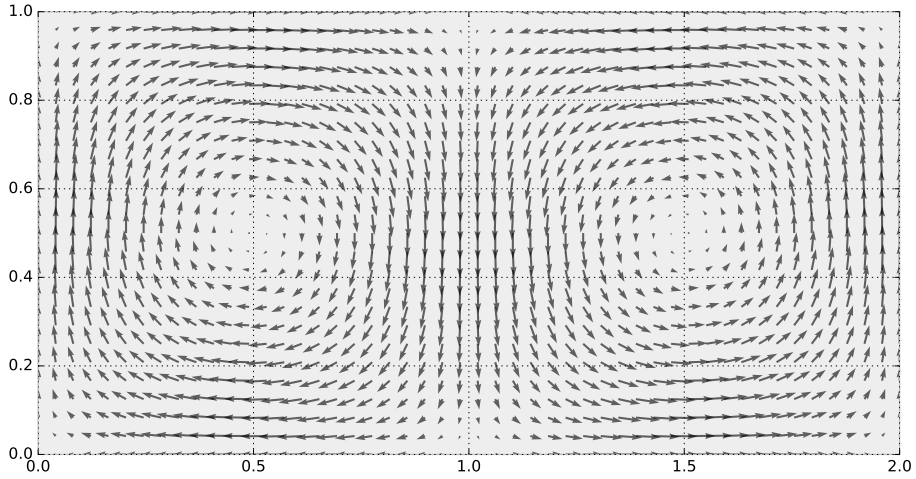
$$S^{-1} = \frac{T(p)}{T} = e + \frac{(1-e)}{p}. \tag{21}$$

Solved for e gives the new metric

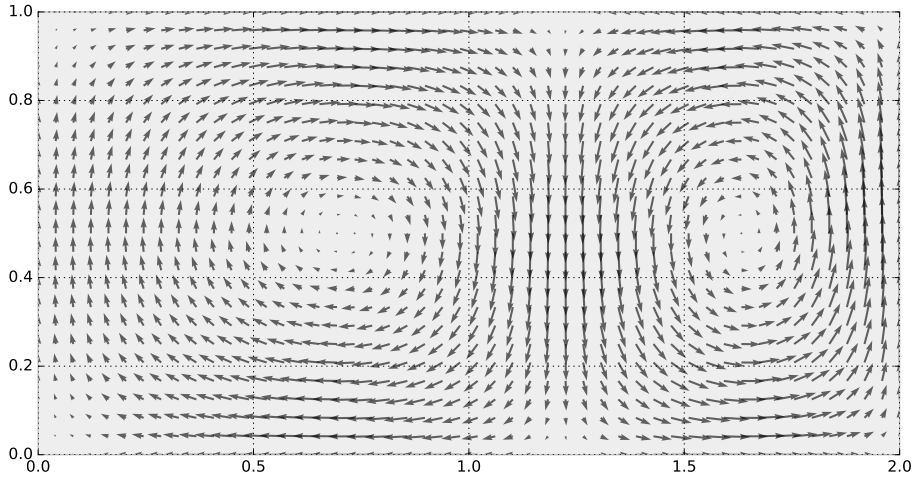
$$e = \frac{\frac{1}{S} - \frac{1}{p}}{1 - \frac{1}{p}}. \tag{22}$$

Use of this new metric is useful, because Eq. (19) is incomplete, e.g, it assumes no serial overhead, including perfectly load balanced work, i.e., that the workload is distributed evenly on all processors and thus all processors compute for the same amount of time and the parallelisation overhead is neglected.

The Karp-Flatt metric is used to experimentally determine e, the serial fraction of the parallel computation, and it can detect sources of inefficiency when solving a problem on several processors, e.g. limited parallelism or increase in algorithmic or architectural overhead.



(a)



(b)

Figure 1: The velocity field Eq. (11a) is given a so called double gyre: two counter-rotating vortices with their centres oscillating horizontally. Here vector plots at two different times are shown; (a) at $t = 0$ s and (b) at $t = 2$ s.

3 Method

The source code for implementation of the problem presented in this thesis is written as part of my work for this thesis. The code is included in Appendix A.

3.1 Implementation

In this chapter the term rank refers to a single thread (also called process) in the parallel computing system. The total number of processes is limited by the computing system and determined as a parameter when running the program. In the following implementation the program runs in parallel with one copy on each rank. The problem is given by:

- We consider the 2D box given by

$$0 \leq x \leq 2,\ 0 \leq y \leq 1 \tag{23}$$

- The velocity field is given by Eq. (11a).
- N_p particles at initial positions in a evenly spaced grid given by $0.95 \leq x \leq 1.05,\ \ 0.45 \leq y \leq 0.55$.
- Particles are transported (advected) for a given time, T_{max}.
- At given time intervals particles are checked if they belong to another rank given by their position. If they do, they are communicated to the new rank.
- The particles properties are written to disk at these time intervals.

The box is divided into several cells which are distributed as even as possible among ranks. All particles are first assigned to their corresponding cell based on the particle's initial position. Then transport of particles is done in parallel by applying the velocity field Eq. (11a) and calculating the trajectories using the 4th order Runge-Kutta method Eq. (9) with a given time step, h. After a given amount of time the transportation algorithm pauses. Now the algorithm checks for each rank if some of it's particles have been transported to a position belonging to a cell in another rank, and if so is the case these particles are moved from their old ranks to the new ranks. In other words each rank which have to send or receive particles need to communicate with other ranks, and the program execution at a

rank can not continue before it has sent and received all particles in question. Data is then written to disk and the transport algorithm continues. This communication procedure is described in detail in Ch. 3.1.1.

Each rank save the properties of all its particles in parallel. At this step calculations like particle concentration are carried out. It is an advantage to have all particles in proximity at the same rank, so less communication is needed when calculating a spatial property. When one cycle as described here is completed, a new cycle of transportation and communication of particles follows.

In Fig. 2 some particles that originally belonged to a rank have been transported to a cell belonging to another rank after a transportation cycle. So they now have to be sent to another rank before the next transportation cycle.

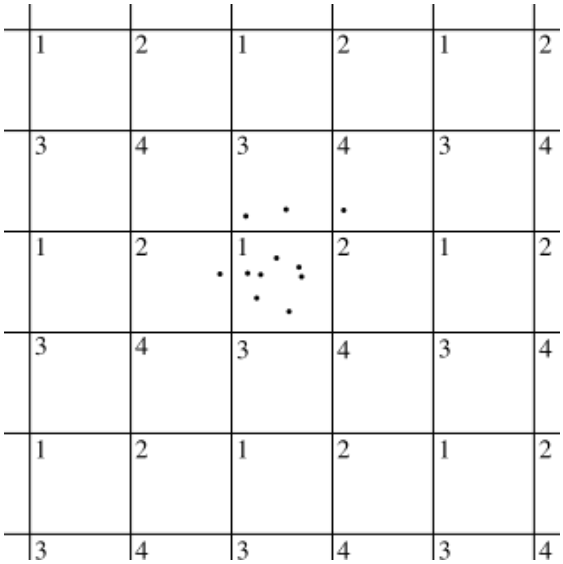


Figure 2: A 2D-system of particles divided into quadratic cells using a *pencil* decomposition and distributed among four ranks. Here a subset of particles is shown after particle transport. All these 11 particles initially resided in a cell in the middle of the figure, which belongs to rank 1. Now four of them have moved to cells belonging to other ranks and thus need to be sent to a new rank. This 2D decomposition can be applied in 3D as shown in Fig. 5. The four ranks are divided in a given way resulting in a periodicity like *121212* and *343434*, but it could also be divided in other ways.

3.1.1 Communication array

Let us say we have 2D-system of particles divided into cells as a 2D version of the *pencil* decomposition distributed among four ranks, as shown in Fig. 2. After a given transportation time all particles have to be moved to the correct rank based on its new position before a potential calculation of concentration and/or similar aggregate spatial properties can be done.

First the number of particles that now resides in a cell belonging to a different rank needs to be determined so we know "how much communication we need to do" and how much memory to allocate for effective memory management. This is done using a so called *Communication Array*. An example of this array is show in Fig. 3.

The routine for filling out the Communication Array is as follows.

- For each particle, check which rank it belongs to, i.e. which rank the cell at the particle positions reside to.
- If the particle is still in a cell owned by the same rank as before, i.e the current *running rank*, do nothing.
- The particles in a new cell not owned by the *running rank* must be sent to a new rank.
- Count the total number of particles which need to be sent to each of the other ranks.

Each rank now knows how many particles it will send to other ranks, but a rank has no knowledge of how many particles it will receive from other ranks. But this information can be received from the other ranks. To exchange this knowledge the following strategy is applied.

- Initialize a quadratic array with data type integer with size number of ranks times number of ranks, filled with zeros.
- Rows in this array represent the total number of particles to be sent.
- Columns represent the total number of particles to be received, this information needs to be gathered from the other ranks.

- Note: This array is usually not diagonal symmetric.
- E.g. row number 2 tells how many particles to be sent from rank 2 to each of the other ranks.
- At this time, each rank have enough information to fill its own row. Do this.
- Call the MPI communicator function `Allreduce` with the sum operator `MPI.SUM`.
- Each rank does now have a complete copy of the Communication Array, and thus know how many particles it should recieve.

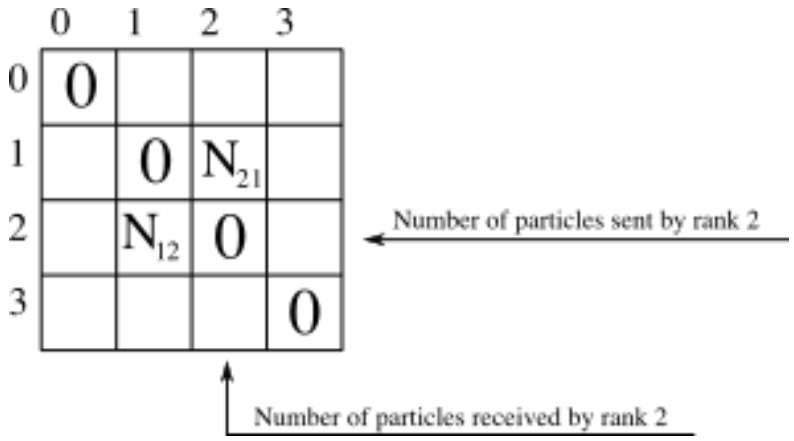


Figure 3: The global Communication Array showing N_{ij}, where N_{ij} is the number of particles to be sent from rank j to rank i. Note that the elements along the diagonal is always 0, because no rank sends or receives particles to itself. Each rank fills out its own row, and the values are communicated to other rows via the MPI-function `Allreduce` using the `MPI.SUM` operator.

3.1.2 Communication of particles

Now the actual transmission of particles can be done. To do this one or more temporary arrays to hold particles to be sent, and one or more temporary arrays to hold particles to be received, must be made. There is at least two ways to do this, the simplest one is to make an array with size equal to the number of ranks times max number of particles to be sent from a rank, and vice versa for particles to be received to a rank. The other is to make a list of arrays, where each array

keeps particles to be sent to a given rank and its length equals the number of particles to be sent to the given rank, and similar for particles to be received to a rank.

The first method has an advantage in its simplicity, but a disadvantage in a possible huge memory required. So we choose to implement the second method. To send a particle to a new rank each relevant property of a particle needs to be communicated, e.g. the coordinates for each applicable dimension. Mass, size and other properties can be communicated if necessary, either all information can be stored in the same array, or it can be divided into on array for each property. Each rank must make a lists of arrays for the particles they are sending, if they send any particles at all, and vice versa if they shall receive some particles. The lists of arrays can be implemented as shown here:

- For each other rank initialize an array with length equal to the number of elements which will be sent to that rank.
- For each other rank initialize an array with length equal to the number of elements which will be received from that rank.
- Iterate over all (local) particles, if the particle should be sent to a rank, copy the particle's properties to the corresponding elements in *send-list-of-arrays*, (then iterate the counter of *send-list-of-arrays*).
- Set the local particle as non-active.

Then the communication procedure follows:

- Set up a non-blocking receive function to accept particles from all other ranks to the *receive-lists-of-arrays*.
- Set up an non-blocking send function to transfer particles to all other ranks from the *send-list-of-arrays*.
- Use the MPI function `wait` with reference to all send and receive functions, so the process wait for all communication to be finished.

Now all ranks have sent and received their particles and the following data management has to be done.

- Move all active particles to the front of the *local array*.
- Expand or shrink the local array with respect to the number of incoming particles.
- Add any incoming particles to the *local array* and mark them as active.

The communication of particles is now done and a new transport of particles can be done and a new communication occurs.

3.2 Parallel computing

3.2.1 Dividing work on processes

How a problem is divided in parts in order to solve using parallel computing, influences the performance. In this project the area of interest, the 2D box given by Eq. (23), is divided along the x-dimension in different number of equally wide cells. Other ways to do this can also be done, but we focus on how the number of cells, and thus the width of each cell, influence the execution time with respect to the number of ranks.

Obviously the number of cells must be equal to or greater than the number of processes, or else some processes will not have any particles at all. Usually smaller cells give a better load balancing, but this is a trade-off, because if a particle has to be communicated to another rank more often because of small cell size, the effeciency can go down.

Dynamic scaling of cells is an advanced method to use to ensure better load balancing, but this will naturally introduce more work to be done in the algorithm, so it is another trade-off.

3.3 Python

In addition to using the mpi4py package for Python bindings to the Message Passing Interface library, the program was implemented with standard Python libraries, such as OS, and libraries from the Scientific Computing Tools for Python, the so called SciPy-stack[9]. SciPy includes packages such as NumPy for powerful

[9] https://www.scipy.org/

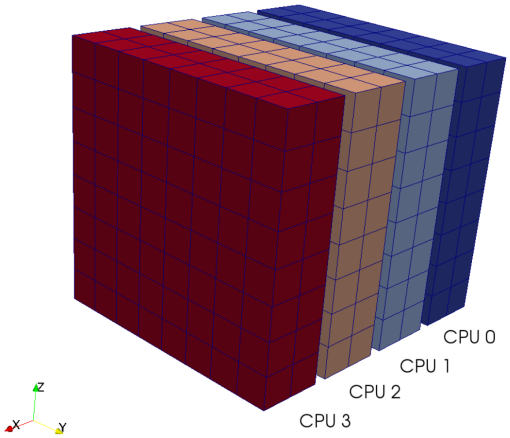


Figure 4: An example of a decomposition of a 3D volume by dividing along one dimension, called a *slab* decomposition. The volume is divided into ranks along only one axis in this case with 4 processors. This is a simple form of mesh generation with an important limitation: Given a cubic mesh of size N^3 the number of processors N_{proc} must be less or equal to N. Figure copied from [1].

N-dimensional array objects used in this project to store properties of the particles, including the coordinates, and computations in the transport part of the problem are done directly on these objects. NumPy arrays are well suited for scientific computation on large arrays because it is fast and have better memory management then standard Python arrays as discussed in Ref. [3].

After each communication step each rank saves four NumPy arrays with the properties of the particles as binary files in the *.npy* format. Matplotlib is used for plotting.

3.4 Performance testing

The program has been tested with Python 2.7 on Vilje[10], a supercomputer at NTNU running SUSE Linux Enterprise Server 11 with SGI MPT[11] as the MPI library. Vilje has a total of 1404 Intel Xeon E5-2670 *Sandy Bridge* nodes, on 19.5 racks, with 2 8-core processors at 2.6 GHz per node. Each 8 core share 10 MB of L3 cache. And it has 2 GB DDR3 1600 MHz-SDRAM memory per core and the interconnect between nodes is Infiniband.

[10] https://www.hpc.ntnu.no/display/hpc/Vilje
[11] https://www.nas.nasa.gov/hecc/support/kb/sgi-mpt_89.html

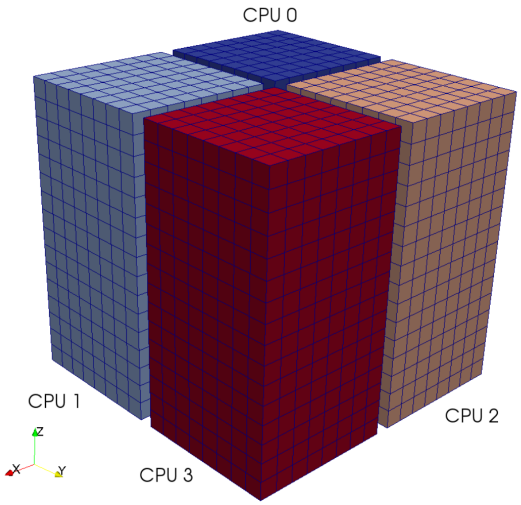


Figure 5: An example of a decomposition of a 3D volume by dividing along two dimensions, also known as a *pencil* decomposition. The volume is divided along two axes. The decomposition shown uses 4 processors, two along each decomposed axis. The maximum number of processes for a cubic mesh of size N^3 is N^2. Figure copied from [1].

The application has been run with particle numbers in the range of 10^3 to $5 \cdot 10^7$ on up to 128 processors on 8 nodes where the area of interest has been divided into 128 to 8192 cells. The time step in the 4th order Runge-Kutta integrator was 0.005 s. At the start of the algorithm and after each 100th time step communication between ranks occurred. T_{max} had values chosen between 9.5 s and 9999.5 s, the number of communications was between 20 and 20 000.

The program was also tested on a desktop computer with Intel i7 CPU, 2 sockets each with 4 cores at 2.7 GHz, 64 GB of memory. In addition simulation was run on a desktop computer with 1 socket with 4 cores, Intel i7 CPU 2.7 GHz, 32 GB of memory.

4 Results and discussion

Simulations where run on several systems with different hardware specifications. Parameters, including number of particles, number of cells, number of ranks (processor used), was varied and the results compared for different number of ranks and other parameters.

Fig. 6 shows the execution time for the program vs. number of ranks used, run on a desktop computer with 8 cores with a Linux operating system. The number of particles, cells and number of communication was varied. Some of the data is shown in Tab. 1.

We see that the program scales well for up to 4 ranks. Since the computer has 8 cores we would expect that it would scale well for up to 8 ranks. For more than 8 ranks there is no true parallelism, the increase in execution time here is expected. The execution time for 7 communications is just slightly longer than for 4. Thus we can conclude that the transportation of particles requires a major part of the total computational capacity, and the communication of particles and writing to disk only requires a fraction of the work load.

Figs. 7, 8, 9, 10, 11, 12 show similar graphs for execution on the supercomputer Vilje.

Fig. 14 shows similar graph for execution on a desktop computer with 4 cores.

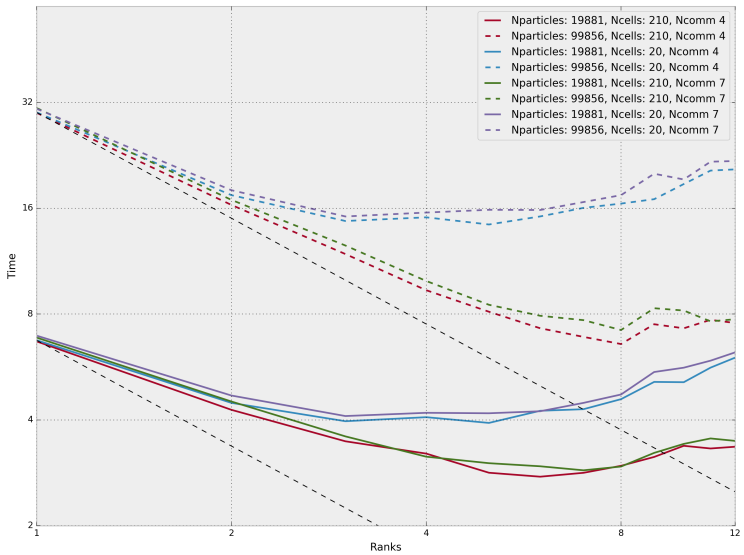


Figure 6: Execution time t/s vs. number of ranks p for different values of $N_{\mathrm{particles}}$, N_{cells} and $N_{\mathrm{communications}}$. The number of integrator time steps is the same, but communication between ranks have occurred more often where $N_{\mathrm{communications}} = 7$. The dashed black line indicates the slope of perfect scaling with respect to 1 rank. ($\sim 1/N_{\mathrm{ranks}}$).

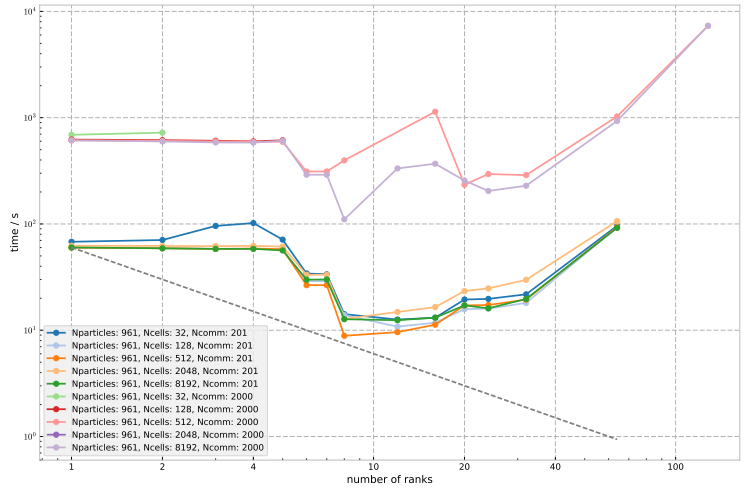


Figure 7: Execution time t/s vs. number of ranks p for different values of $N_{\mathrm{particles}}$, N_{cells} and $N_{\mathrm{communications}}$. The dashed black line indicates the slope of perfect scaling with respect to 1 rank. ($\sim 1/N_{\mathrm{ranks}}$).

Table 1: Measured execution time t in seconds for a given number of particles, $N_{\text{particles}}$, cells, N_{cells}, $N_{\text{communications}}$ and ranks, p. T_{max} is kept constant. Run on a desktop computer. We see that the execution time shrinks when p increases from 1 to 6 and then increases when p is larger than 7. A cause can be that the problem is not easily divided on 7 ranks and for each extra rank used more work is done on communication between ranks and this outweighs the benefit gained from the fact that each ranks has less work to do in the transport part of the problem.

$N_{\text{particles}}$	N_{cells}	p	$N_{\text{communications}}$	T_{max}	dt	t/s
19881	20	1	4	3	0.005	6.742
19881	20	1	7	3	0.005	6.946
19881	210	1	4	3	0.005	6.689
19881	210	1	7	3	0.005	6.858
19881	20	4	4	3	0.005	4.072
19881	20	4	7	3	0.005	4.191
19881	210	4	4	3	0.005	3.208
19881	210	4	7	3	0.005	3.144
19881	20	8	4	3	0.005	4.583
19881	20	8	7	3	0.005	4.724
19881	210	8	4	3	0.005	2.962
19881	210	8	7	3	0.005	2.947
19881	20	12	4	3	0.005	6.012
19881	20	12	7	3	0.005	6.223
19881	210	12	4	3	0.005	3.355
19881	210	12	7	3	0.005	3.487
99856	20	1	4	3	0.005	29.978
99856	20	1	7	3	0.005	30.699
99856	210	1	4	3	0.005	29.889
99856	210	1	7	3	0.005	30.849
99856	20	4	4	3	0.005	15.074
99856	20	4	7	3	0.005	15.554
99856	210	4	4	3	0.005	9.353
99856	210	4	7	3	0.005	9.935
99856	20	8	4	3	0.005	16.494
99856	20	8	7	3	0.005	17.431
99856	210	8	4	3	0.005	6.574
99856	210	8	7	3	0.005	7.207
99856	20	12	4	3	0.005	20.634
99856	20	12	7	3	0.005	21.796
99856	210	12	4	3	0.005	7.547
99856	210	12	7	3	0.005	7.733

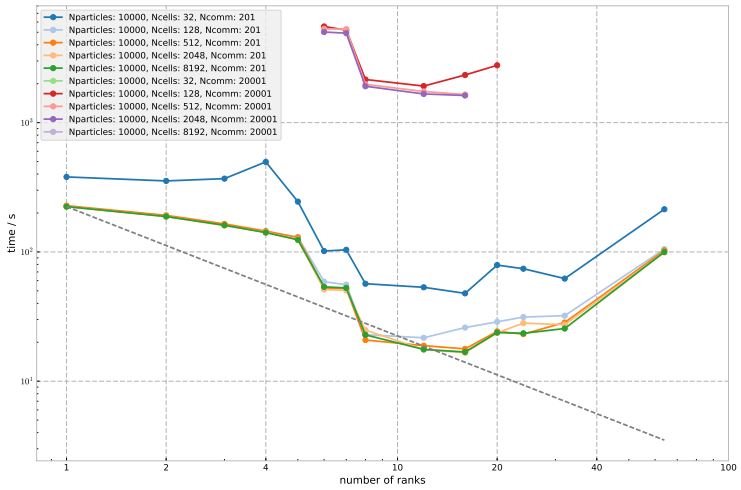


Figure 8: Execution time t/s vs. number of ranks p for different values of $N_{\mathrm{particles}}$, N_{cells} and $N_{\mathrm{communications}}$. The dashed black line indicates the slope of perfect scaling with respect to 1 rank. ($\sim 1/N_{\mathrm{ranks}}$).

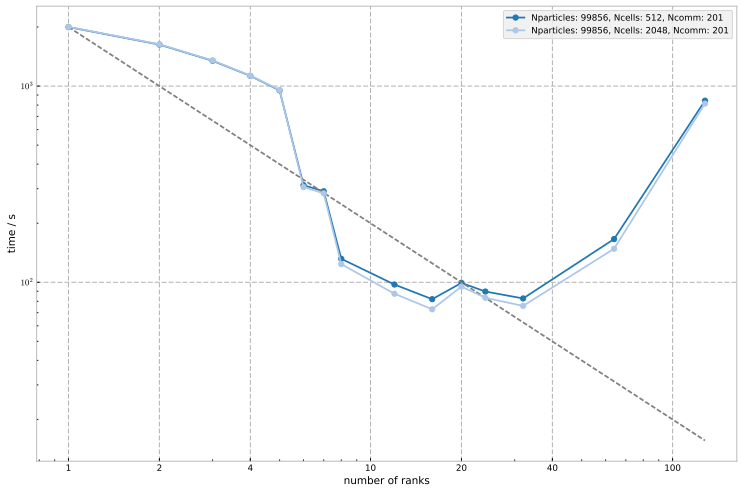


Figure 9: Execution time t/s vs. number of ranks p for different values of $N_{\mathrm{particles}}$, N_{cells} and $N_{\mathrm{communications}}$. The dashed black line indicates the slope of perfect scaling with respect to 1 rank. ($\sim 1/N_{\mathrm{ranks}}$).

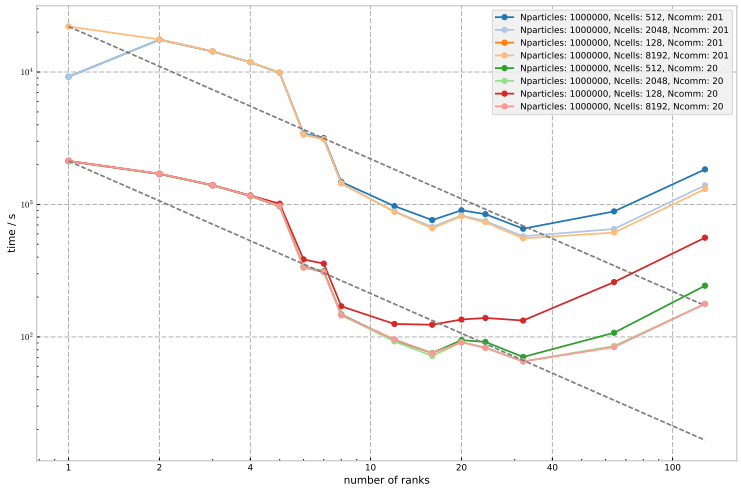


Figure 10: Execution time t/s vs. number of ranks p for different values of $N_{\mathrm{particles}}$, N_{cells} and $N_{\mathrm{communications}}$. The dashed black line indicates the slope of perfect scaling with respect to 1 rank. ($\sim 1/N_{\mathrm{ranks}}$).

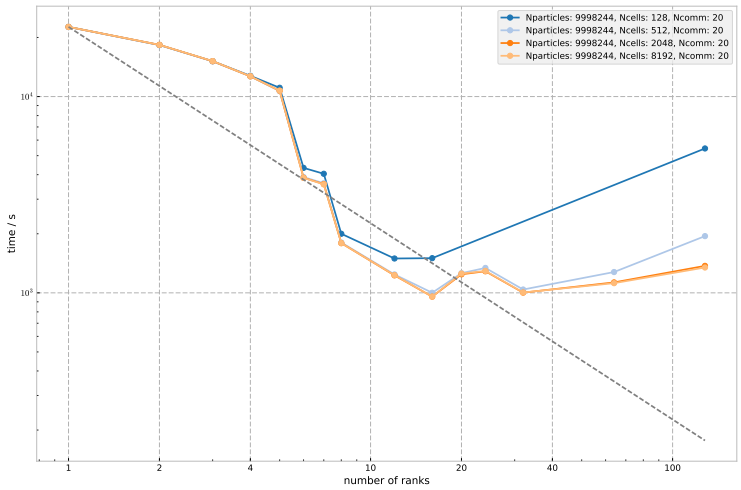


Figure 11: Execution time t/s vs. number of ranks p for different values of $N_{\mathrm{particles}}$, N_{cells} and $N_{\mathrm{communications}}$. The dashed black line indicates the slope of perfect scaling with respect to 1 rank. ($\sim 1/N_{\mathrm{ranks}}$).

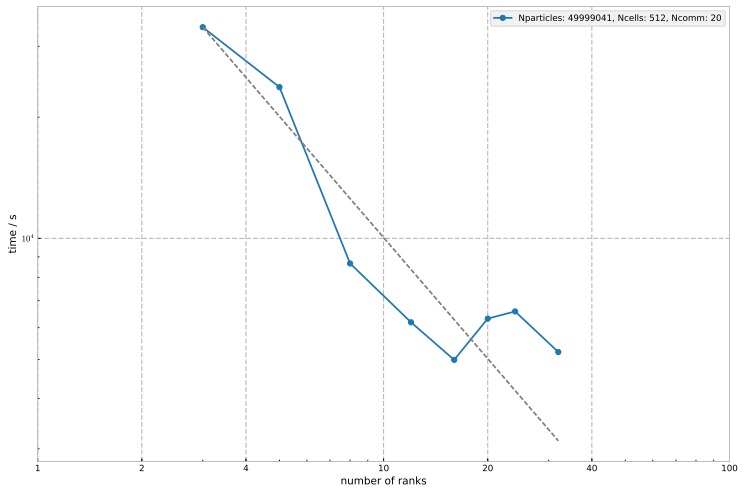


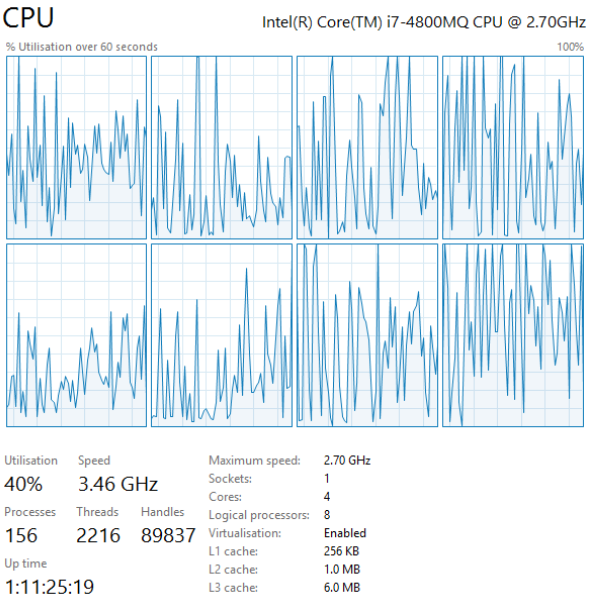
Figure 12: Execution time t/s vs. number of ranks p for different values of $N_{\mathrm{particles}}$, N_{cells} and $N_{\mathrm{communications}}$. The dashed black line indicates the slope of perfect scaling with respect to 3 ranks. ($\sim 1/N_{\mathrm{ranks}}$))

We observe in the Figs. 7, 8, 9, 10, 11, 12 that the execution time generally decreases from 1 to 16 ranks, and then increases for more than 16 ranks. The reason for this is that each node at the supercomputer have 16 cores and when using more than 16 cores communication has to be done between nodes. Communication between nodes is handled by the Infiniband interconnect, while quite fast, this is still slower than communication internally on one node.

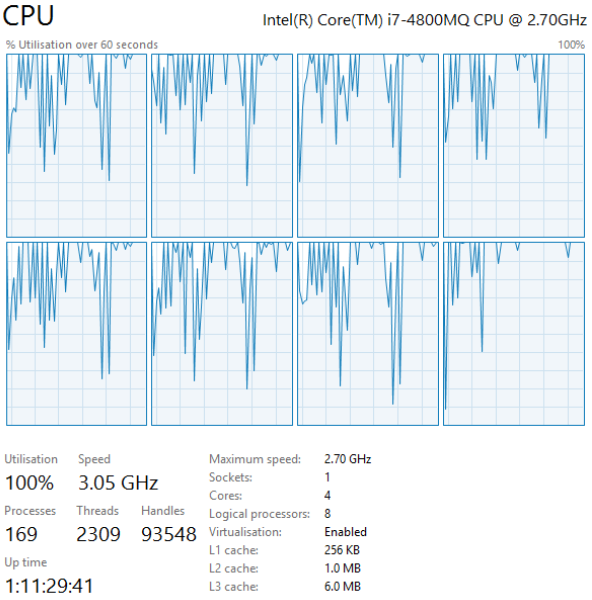
Figs. 9, 11, 10 show that the speed-up scaled quite well with super-linear scaling between 8 and 16 ranks for particle numbers greater than 10000. This may be explained by the fact that the use of more processors allows faster read and write speed between the CPU and the memory, because smaller amount of data have to be in each CPU's memory, and thus a larger fraction of the data can be kept in lower level of cache, which is faster than higher level of cache.

For a small number of particles Figs. 7 and 8 show that the execution time decreases very slightly, if at all, from 1 to 4-5 ranks. This may be caused by the fact that one node runs the program for 1 to 5 ranks at the same time, to save CPU time at Vilje. So the shared cache can be used by another instance of the program and thus the performance per instance goes down. One may wonder why this does not also happen for a larger number of particles.

Fig. 13 shows the CPU utilisation on a desktop computer running Windows 10. It is interesting to note that the utilisation is under 50 % for 2 ranks, while for 8 ranks the utilisation is close to 100 %. This effect is reflected in the blue graphs in Fig. 14, where we observe that the execution time goes up from 1 to 2 ranks, not as expected. This could be due to Windows 10 does not allow the program to use all the CPU capacity at lower number of ranks, but when the program runs on a higher number of ranks it is allowed to utilise all the CPU capacity. It does not really run in parallel. This shows that not only the computer architecture affects the performance, but also the operating system and how it delegates the resources. The difference in increasing execution time for a higher number of ranks than cores on the Linux system show in Fig. 6 compared to the Windows system shown in Fig. 13 is interesting, but not investigated further.



(a) 2 ranks



(b) 8 ranks

Figure 13: CPU utilisation from 0 s to 60 s while executing the program with 2 and 8 ranks respectively using a desktop computer with 4 cores, running Windows 10. The computer does not really use only one processor for each rank, like a truly parallel computation, but divide the work on all available processors.

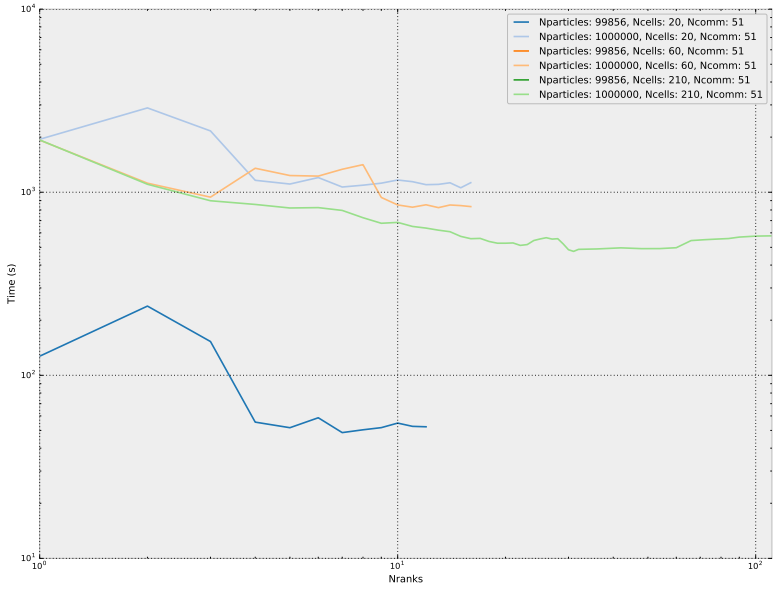


Figure 14: Execution time t/s vs. number of ranks p for different values of $N_{\mathrm{particles}}$ and N_{cells} for a test run on computer with four cores. For some test the execution time first increases for an increasing number of ranks, then decreases up to about 10 ranks. After monitoring the CPU load it seems like the operating system, Windows 10, does not utilise the full potential of the CPUs when running at 1 to approximately 4 ranks. So the execution time is not optimal. For a very large number of ranks the execution time is not increasing. This is because the CPU does not have to communicate with another CPU on another node, but the code runs in pseudo parallel at the same computer.

4.1 Calculate Metrics

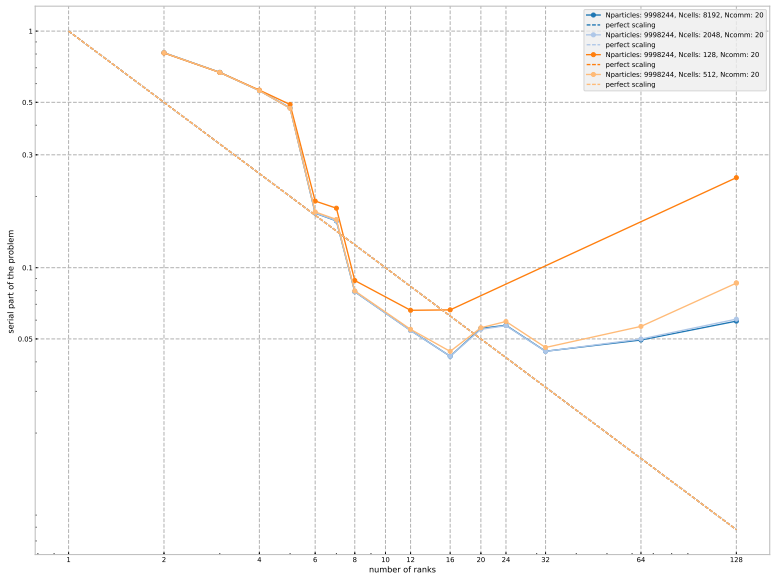


Figure 15: Karp-Flatt metric shown for different ranks.

In Fig. 15 we see that the serial fraction decreases down to 16 ranks, but then it increases. This indicates that parallel overhead is a contributing factor to the poor speed-up for higher than 16 ranks. The extra work need to be done when communicating between nodes is most probably the main factor for this.

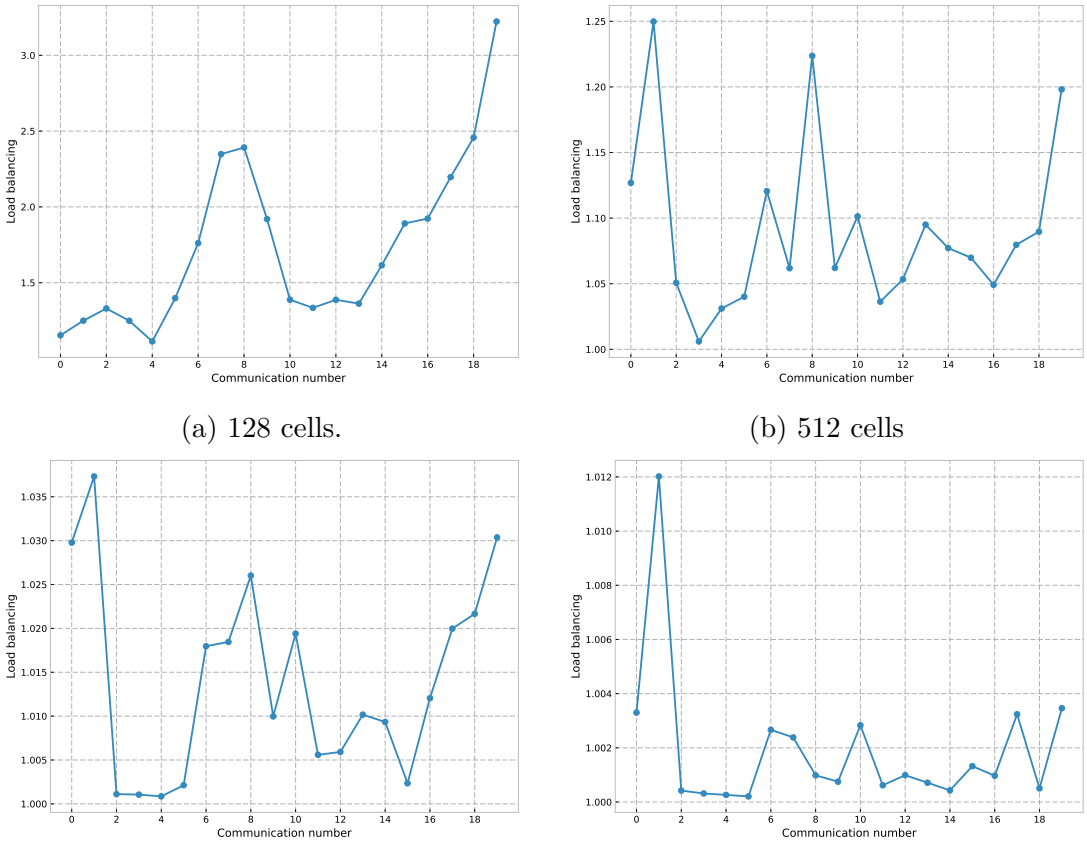
4.2 Load balancing

Load balancing, P_{LB}, is calculated using Eq. (12). For a given run the load balancing is calculated for each time when data was written, and the mean load balancing is the average of each of them.

A load balancing value of 1 is perfect. Load balancing is best at higher number of cells, especially when the number of cells are high compared to the number of ranks, as show in Tab. 2 and Figs. 17 and 16. This may be because the probability for an even particle distribution over different cells is is higher if the number of cell is higher and thus smaller, then particles are more evenly distributed on ranks.

Table 2: Load balancing averaged over all communications for different number of cells for 16, 32 and 128 ranks. It seems like a higher number of cells compared to ranks gives a better load balancing.

$N_{\text{particles}}$	$N_{\text{communications}}$	p	N_{cells}	$\overline{P_{\text{LB}}}$
10000000	20	16	128	1.734
10000000	20	16	512	1.091
10000000	20	16	2048	1.014
10000000	20	16	8192	1.002
10000000	20	32	512	1.221
10000000	20	32	8192	1.005
10000000	20	128	128	13.876
10000000	20	128	512	8.729
10000000	20	128	2048	8.113
10000000	20	128	8192	8.015



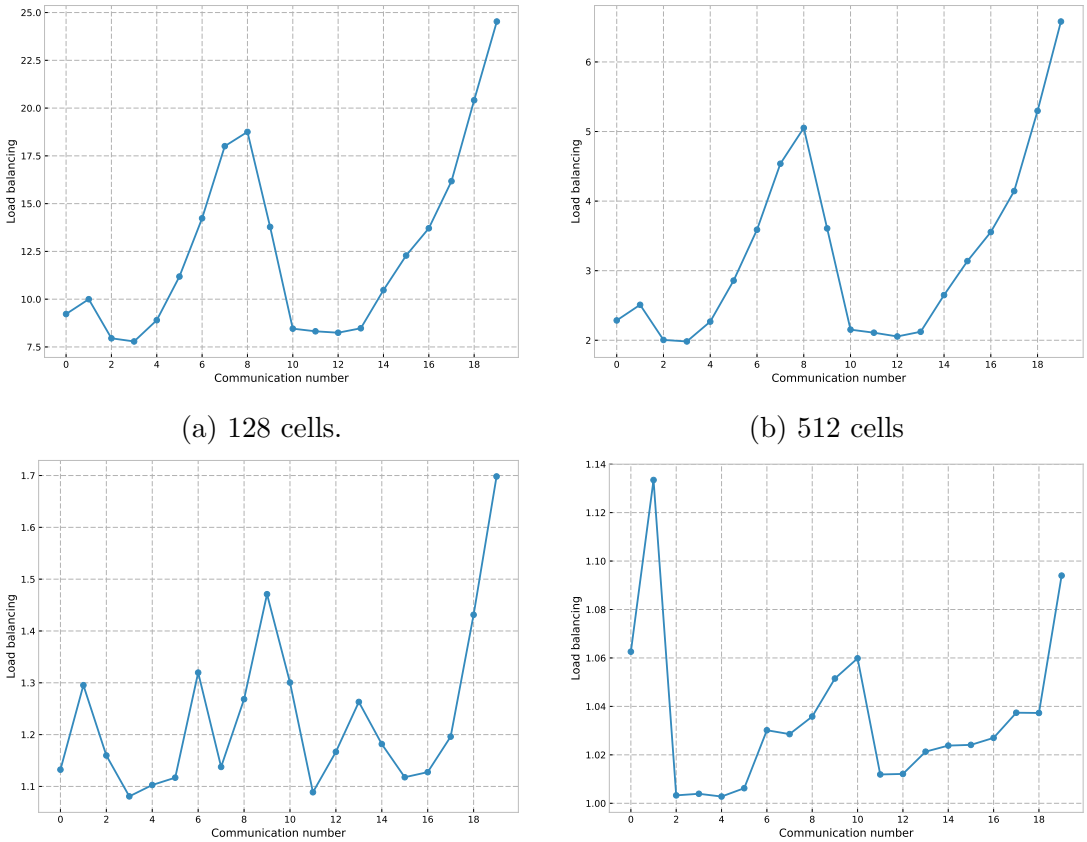
(a) 128 cells.

(b) 512 cells

(c) 2048 cells

(d) 8192 cells

Figure 16: Load balance for 10^7 particles and 16 ranks and different number of cells. Shown for each time when data was written to disk. A value of 1 is perfect, larger values indicate poor load balance. The load balance follows the same trend for all number of cells, and is better for a higher number of cells. This is expected because a higher number of cells increases the probability of particles being evenly distributed on ranks.



(a) 128 cells.

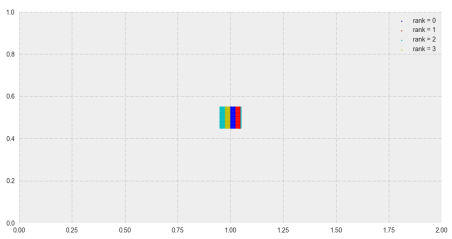
(b) 512 cells

(c) 2048 cells

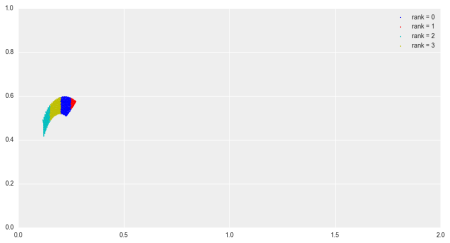
(d) 8192 cells

Figure 17: Load balance for 10^7 particles and 128 ranks and different number of cells. Shown for each time when data was written to disk. A value of 1 is perfect, larger values indicate poor load balance. The load balance follows the same trend for all number of cells, and is better for a higher number of cells. This is expected because a higher number of cells increases the probability of particles being evenly distributed on ranks.

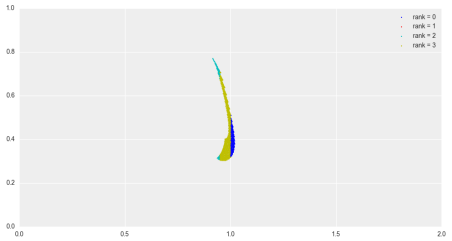
One thing to note in our specific case is that we use an initial particle grid centred at the middle of the area of interest, and at some periods of time the particle grid can be moved so the shape is vertical, as shown in Fig. 18c. So a division of cells as a chess board grid could give better load balancing, but it also requires some more complex source code.



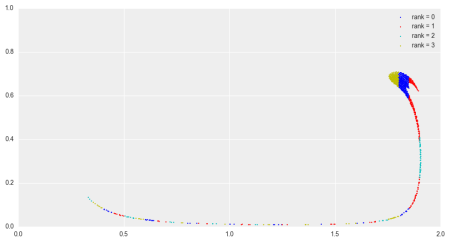
(a) $t = 0$ s



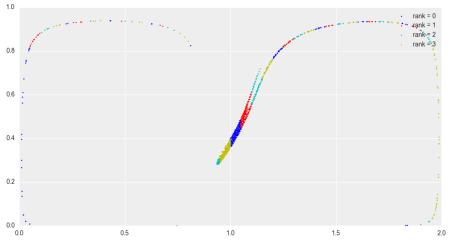
(b) $t = 5$ s



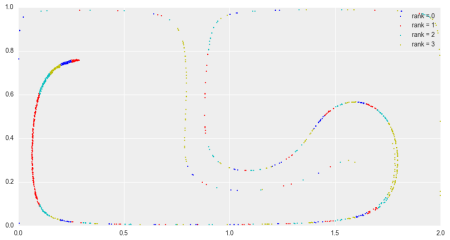
(c) $t = 10$ s



(d) $t = 15$ s



(e) $t = 20$ s



(f) $t = 25$ s

Figure 18: Particles shown at different time in the transport procedure, with a time step of 0.5 seconds. This example shows 40 cells divided between four ranks. Particles are coloured according to rank, visualising the division of the area.

5 Conclusion

A program for calculating particle transport in parallel on an arbitrary number of processes was written in Python. The 4th order Runge-Kutta method was used as the numerical integrator. Calculations were done directly on Numpy-arrays, which was the data type used for storing the particle's properties. The program was run in parallel for 1 to 128 processes on the supercomputer Vilje at NTNU. Different parameters were varied, including the number of particles and how they were divided on different processes. The execution time was recorded and compared to simulations using varying degrees of parallelism and the load balancing, speed-up and Karp-Flatt metric was measured for some specific selection of the parameters.

The speed-up scaled quite well with super-linear scaling between eight and sixteen processes at some selection of parameters. This could be because the use of more processors allows faster read and write speed between the CPU and the memory, because smaller amount of data have to be in each CPU's memory, and thus a larger fraction of the data can be kept in lower level of cache, which is faster than higher level of cache.

The program demonstrates quite reasonable performance, when compared to results of [8] while the systems simulated where not exactly the same. For the results shown in Fig. 12, the Python program used approximately 24 CPU hours (execution time times the number of processes used) to simulate 2000 time steps for $5 \cdot 10^7$ particles. Compare to approx 7680 CPU hours for 10^9 particles in [8], using a Fortran code with MPI.

5.1 Suggestions for further work

- To increase the load balancing and thus the performance it may be an idea to implement the cells, which are divided on ranks, like a chess board grid.
- Code profiling can be run to measure performance and memory usage for different part of the code. Probably if some bottlenecks will be found, and one should focus on optimising that part of the code.
- A similar program can be implemented in a low level computer language, like Fortran, and the performance of the two programs can be compared.
- Diffusion could be added in the transport of particles, and several simulations with a contribution from a random diffusion could be run to compare the performance over different transport patterns.

- Simulations can be run on Vilje with a number of particles an order of magnitude higher than done in this project.
- Simulations can be run on other supercomputers to compare it with the performance at Vilje.

References

[1] Mikael Mortensen, Hans Petter Langtangen. High performance Python for direct numerical simulations of turbulent flows, *Computer Physics Communications*, 203: 53-65, June 2016.

[2] Visser, Andre W. Lagrangian modelling of plankton motion: from deceptively simple random walks to Fokker-Planck and back again, *Journal of Marine Systems*, 70: 287-299, 2008.

[3] T. E. Oliphant. Python for Scientific Computing, *Computing in Science Engineering*, 9: 10-20, 2007.

[4] Ernst Hairer, Syvert P. Nørsett, Gerhard Wanner. Solving Ordinary Differential Equations I (2nd Revised. Ed.): Nonstiff Problems. *Springer-Verlag New York, Inc.*, New York, NY, USA. 1993.

[5] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities, *Proceedings of FIPS Spring Joint Computer Conference 30, Atlantic City, NJ.*, 1967.

[6] Alan H. Karp, Horace P. Flatt. Measuring Parallel Processor Performance, *Communications of the ACM*, 33, 5: 539-543, May 1990.

[7] K. Onu, F. Huhn, G. Haller. LCS Tool: A computational platform for Lagrangian coherent structures, *Journal of Computational Science*, 7: 26-36, 2015.

[8] Tor Nordam, Raymond Nepstad, Thomas Janke Memo: Parallel particle transport, *SINTEF Materials and Chemistry*, project 102008977: 25th May 2016.

A
Source code

```
1  # main file for Python parallell particle transport
2  # main.py
3
4  # example to run:
5  """
6  mpiexec -n 4 python main.py
7  """
8  # where 4 is the number of processes
9
10 # first, get number of particles from the parameter file
11 # it should be equal particle grid initialised by IO.py
12 from parameters import nParticles, t_max
13 nParticles = int(nParticles)
14
15 # to import the path to the other py-files
16 import sys
17 # all files are now copied to a new local folder
18 # sys.path.append('/home/ntnu/simenmi/PyPPT/')
19
20 import mpi4py.MPI as MPI
21
22 comm = MPI.COMM_WORLD
23 rank = comm.Get_rank()
24 mpi_size = comm.Get_size()
25
26 from time import time
27 import datetime # for timestamp in timing.txt
28
29 # wait for all ranks and start the timer
30 comm.Barrier()
31 if rank == 0:
32     tic = time()
33
34 import os
35 import numpy as np
36 import IO
37 import transport #
38 #import plot # for plotting directly in main.py, currently commented
      out
39 import matplotlib
40 matplotlib.use('agg')
41 import communication
42
43 ## VARIABLES start
44
```

```
45 # integrator timestep
46 dt    = 0.005
47 # number of timesteps between each communication event
48 Ndt = int(0.5/dt)
49
50 # simulation start time
51 t_0 = 0
52 # t_max is imported from parameters
53
54 ## VARIABLES end
55
56 print('\nnParticles', nParticles)
57 print('rank', rank)
58 print('total number of ranks', mpi_size)
59
60 # get initial positions: from file, place at random coordinates or
       use another method
61
62 # initial particle grids must be created from IO.py
63
64 ids, active, XY = IO.load_grid_of_particles_input(rank, time = t_0)
65 # using N as the actual number of particles if nParticles from
       parameters.py is inaccurate
66 # nParticles is only used for saving output. N is used when writing
       to timing.txt
67 if rank == 0:
68     N = np.sum(active)
69
70 # start at initial time
71 t = t_0
72
73 ###
74 # MAIN LOOP:
75 print('\nstart time = %s' % t)
76
77 # communicating before first transport, to avoid bad load balancing
       at the beginning
78 print('This is rank %s, communicating before mainloop' % (rank))
79 ids, XY, active = communication.exchange(comm, mpi_size, rank, ids,
      XY[0,:], XY[1,:], active)
80
81 # save initial values after particles are exchanged to the correct
       rank at start
82 #print('Saving particles before first transportation')
83 IO.save_grid_of_particles_output(nParticles, ids, active, XY, t, rank
       , mpi_size)
84 #plot.plot(rank, XY[:,active], t, dt)
85
86 while t + dt <= t_max:
```

```
87      # take Ndt timesteps
88
89      # only take active particles into transport function
90      # non-active particles should be in the end of the arrays,
91      # so if we want we can use the index as XY[:,:np.sum(active)]
            instead
92      print('This is rank %s, transporting %s particles' % (rank, np.
            sum(active)))
93      # flush stdout to print all output up to this point
94      sys.stdout.flush()
95      # transport and return new t
96      XY[:,active], t = transport.transport(XY[:,active], active, t,
            Ndt, dt)
97      #plot.plot(rank, XY[:,active], t, dt, name = '_after_transport-
            before_comm_')
98
99      # Then communicate
100     '''
101     # all variables taken in by exchange() are local variables for
            the given rank (except mpi_size)
102     def exchange(communicator,
103                  mpi_size,
104                  rank,
105                  particle_n,
106                  particle_id,
107                  particle_x,
108                  particle_y,
109                  particle_active):
110
111             return (particle_id,
112                  particle_x,
113                  particle_y,
114                  particle_active)
115     '''
116     print('This is rank %s, ready for communication' % (rank))
117     sys.stdout.flush()
118     ids, XY, active = communication.exchange(comm, mpi_size, rank,
            ids, XY[0,:], XY[1,:], active)
119
120     ### Then calculate concentration or other spatial property here
121     #print('Saving particles in while loop')
122     IO.save_grid_of_particles_output(nParticles, ids, active, XY, t,
            rank, mpi_size)
123     comm.Barrier()
124     #plot.plot_from_files(t, mpi_size, dt)
125     print('\nt = %s' % t)
126
127 ###
128 # saving parameters and execution time to file
```

```
129
130 # save timing.txt at relative path
131 # current directory
132 cdir = os.path.dirname(os.path.realpath(__file__))
133 timefilename  = os.path.join(cdir, os.pardir , 'timing_nPart_%s.txt'
      % nParticles)
134
135 comm.Barrier()
136 if rank == 0:
137     toc = time()
138     if not os.path.exists(timefilename):
139         timefile = open(timefilename, 'w')
140         timefile.write('Npart\tNcells\tNranks\tNcomm\tTmax\tdt\tTime\
              tlogTimestamp\n')
141         timefile.close()
142     with open(timefilename, 'a') as timefile:
143         Ncomm = 1 + int(t_max / dt / Ndt)
144         timefile.write('%s\t%s\t%s\t%s\t%.5f\t%.5f\t%.5f\t%s\n' % (N,
              communication.cell_x_n, mpi_size, Ncomm, t_max, dt, toc -
              tic, datetime.datetime.now()))
```

```
# script with functions to use in main
# this script handle the "transport of particles"-part
# transport.py

'''
def timestep(X, dt, t):
    # Use double gyre and RK4 to calculate new functions
    return X
'''
import numpy as np

## FUNCTIONS start

# doublegyre velocity field
# splitted up in x- and y-components for readability
def doublegyre(x, y, t, A, e, w):
    a = e * np.sin(w*t)
    b = 1 - 2*e*np.sin(w*t)
    f = a*x**2 + b*x
    return np.array([
            -np.pi*A*np.sin(np.pi*f) * np.cos(np.pi*y),
                # x component of velocity
             np.pi*A*np.cos(np.pi*f) * np.sin(np.pi*y) * (2*a*x + b)
                 # y component of velocity
        ])

# wrapper function to pass to integrator
# XY is a two-component vector [x, y]
def f(XY, t):
    # Parameters of the velocity field
    A = 0.1
    e = 0.25 # epsilon
    w = 1    # omega
    return doublegyre(XY[0,:], XY[1,:], t, A, e, w)

# 4th order Runge-Kutta integrator. XY is a two-component vector [x,
    y]
def rk4(XY, t, dt, f):
    k1 = f(XY,            t)
    k2 = f(XY + k1*dt/2, t + dt/2)
    k3 = f(XY + k2*dt/2, t + dt/2)
    k4 = f(XY + k3*dt,   t + dt)
    return XY + dt*(k1 + 2*k2 + 2*k3 + k4) / 6

# function to calculate a trajectory from an initial position XY0 at
    t = 0,
# moving forward until t = t_max, using the given timestep and
    integrator
```

```
44 # XY is a two-component vector [x, y]
45 def trajectory(XY, current_time, Ndt, dt, integrator, f):
46     t = current_time
47     # number of timesteps
48     #Nt = int(t_max/dt)
49     # loop over all timesteps
50     #for i in range(1, Ndt+1):
51     for i in range(Ndt):
52     ## TODO: what's the difference between the last line and the line
           over it?
53         XY = integrator(XY, t, dt, f)
54         t += dt
55     # return entire trajectory and current time
56     return XY, t
57
58 def transport(XY, particle_active, current_time, Ndt, dt):
59     # only active particles are passed from main
60     # XY is a two-component vector [x, y]
61     # loop over grid and update all positions
62     # this is where parallelisation would happen, since each position
           is independent of all the others
63
64     # array to hold all grid points after transport
65     #XY1 = np.zeros((2, particle_n))
66     # keep only the last position, not the entire trajectory
67     XY, t = trajectory(XY, current_time, Ndt, dt, rk4, f)
68     return XY, t
69
70 ## FUNCTIONS end
```

```
# script with functions to use in main-file
# this script handle the "communication of particles between ranks"-part
# communication.py

'''
def exchange(X):
    # Handle all the communication stuff here
    # return the updated particle arrays
    # (which may be of a different length now)
    return X
'''
### future work:
#implement 'find biggest factor'-function or another dynamical function to determine number of cells in each direction

import sys
import numpy as np
import mpi4py.MPI as MPI

##  INITIALISING start
# number of cells in each direction (we only divide x-direction)
from parameters import nCells
cell_x_n = nCells
cell_y_n = 0
cell_n = cell_x_n + cell_y_n

# scaling factor when expanding/scrinking local arrays
scaling_factor = 1.25 ## variable
shrink_if = 1/(scaling_factor**3)

# the particles are defined with its properties in several arrays
# one tag for each properties which are communicated to other ranks
# tags: id, x-pos, y-pos
# other properties: active-status
tag_n = 3

# buffer overhead to use in memory reservation for non-blocking communication
buffer_overhead = 1000

# spatial properties
x_start = 0
x_end = 1
y_start = 0
y_end = 1

x_len = x_end - x_start
```

```
46 y_len = y_end - y_start
47
48 ## INITIALISING end
49
50 ## secondary FUNCTIONS start
51
52 # function to find the corresponding rank of a cell
53 # this function determines how the cells are distributed to ranks
54 # simple function when dividing only in one direction
55 def find_rank_from_cell(cell_id, mpi_size):
56     return int(cell_id % mpi_size)
57
58 # function to find the corresponding cell of a position
59 # this function determines how the cells are distributed
      geometrically
60 # simple function when dividing only in one direction
61 def find_cell_from_position(x, y):
62     return int(((x - x_start)/(x_len))*(cell_x_n)) # for 1D
63
64 # send_n_array: array to show how many particles should be sent from
      one rank to the others
65 # filled out locally in each rank, then communicated to all other
      ranks
66 # rows represent particles sent FROM rank = row number (0 indexing)
67 # column represent particles sent TO rank = row number (0 indexing)
68 # function to fill out the array showing number of particles need to
      be sent from a given rank given thelocal particles there
69 # local particles are the particles who belonged to the rank before
      the transport of particles.
70 # some of the particles may have to been moved to a new rank if they
      have been moved to a cell belonging to a new rank
71 # send_to: array to show which rank a local particle needs to be sent
       to. or -1 if it should stay in the same rank
72 def global_communication_array(mpi_size, rank, particle_n, particle_x
      , particle_y, particle_active):
73     #print('global com.array, particle n:', particle_n)
74     # reset arrays telling which particles are to be sent
75     send_to = np.zeros(particle_n, dtype=int) # local
76     send_to[:] = -1
77     send_n_array = np.zeros((mpi_size, mpi_size), dtype=int)
78     for i in range(particle_n):
79         # only check if the particle is active
80         if particle_active[i]:
81             # find the rank of the cell of which the particle (its
                  position) belongs to
82             particle_rank = find_rank_from_cell(
                  find_cell_from_position(particle_x[i], particle_y[i]),
                   mpi_size)
83             # if the particle's new rank does not equal the current
```

```
                rank (for the given process), it should be moved
84              if particle_rank != rank:
85                  send_n_array[int(rank)][int(particle_rank)] =
                        send_n_array[int(rank)][int(particle_rank)] + 1
86                  send_to[i] = particle_rank
87                  # converted indices to int to not get 'deprecation
                        warning'
88      return send_to, send_n_array
89
90  # function to reallocate active particles to the front of the local
        arrays
91  # active_n = number of active particles after deactivation of
        particles sent to another rank, but before receiving.
92  # aka. particles that stays in its own rank
93  def move_active_to_front(particle_id, particle_x, particle_y,
        particle_active, active_n):
94      #print('move_active_to_front(), particle_active, active_n:',
            particle_active.dtype, active_n)
95      particle_id[:active_n] = particle_id[particle_active]
96      particle_x[:active_n] = particle_x[particle_active]
97      particle_y[:active_n] = particle_y[particle_active]
98      # set the corresponding first particles to active, the rest to
            false
99      particle_active[:active_n] = True
100     particle_active[active_n:] = False
101     return particle_id, particle_x, particle_y, particle_active
102
103 ## secondary FUNCTIONS end
104
105 ## main FUNCTION start
106
107 # all variables taken in by exchange() are local variables for the
        given rank (except mpi_size)
108 def exchange(communicator,
109              mpi_size,
110              rank,
111              #particle_n, # could also be calculated in function:
                    particle_n = np.size(particle_id)
112              particle_id,
113              particle_x,
114              particle_y,
115              particle_active):
116
117     #print('mpi_size from main module', mpi_size)
118     #print('mpi_size from communication module',
            mpi_size_communication_module)
119     #print('rank from main module', rank)
120     #print('rank from communication module',
            rank_communication_module)
```

```
121
122     # compute "global communication array"
123     # with all-to-all communication
124
125     # length of local particle arrays
126     particle_n = np.size(particle_id)
127     # note: not necessary equal to number of active particles
128
129     send_to, send_n = global_communication_array(mpi_size, rank,
            particle_n, particle_x, particle_y, particle_active)
130
131     # all nodes receives results with a collective 'Allreduce'
132
133     # mpi4py requires that we pass numpy objects (byte-like objects)
134     send_n_global = np.zeros((mpi_size, mpi_size), dtype=int)
135     communicator.Allreduce(send_n, send_n_global , op=MPI.SUM)
136
137     # each rank communicate with other ranks if it sends or receives
            particles from that rank
138     # this information is now given in the "global communication
            array"
139
140     # point-to-point communication of particles
141
142     # using list of arrays for communication of particle properties
143     # initializing "communication arrays": send_*** and recv_***
144     # send_**: list of arrays to hold particles that are to be sent
            from a given rank to other ranks,
145     #   where row number corresponds to the rank the the particles
            are send to
146     # recv_**: list of arrays to hold particles that are to be
            received from to a given rank from other ranks,
147     #   where row number corresponds to the rank the particles are
            sent from
148
149     send_id = []
150     send_x = []
151     send_y = []
152     recv_id = []
153     recv_x = []
154     recv_y = []
155
156     # total number of received particles
157     received_n = np.sum(send_n_global, axis = 0)[rank]
158
159     for irank in range(mpi_size):
160
161         # find number of particles to be received from irank (sent to
                current rank)
```

```
162             Nrecv = send_n_global[irank, rank]
163             # append recv_id with the corresponding number of elements
164             recv_id.append(np.zeros(Nrecv, dtype = np.int64))
165             recv_x.append(np.zeros(Nrecv, dtype = np.float64))
166             recv_y.append(np.zeros(Nrecv, dtype = np.float64))
167
168             # find number of particles to be sent to irank (from current
                    rank)
169             Nsend = send_n_global[rank, irank]
170             # append send_id with the corresponding number of elements
171             send_id.append(np.zeros(Nsend, dtype = np.int64))
172             send_x.append(np.zeros(Nsend, dtype = np.float64))
173             send_y.append(np.zeros(Nsend, dtype = np.float64))
174
175         # counter to get position in send_** for a particle to be sent
176         send_count = np.zeros(mpi_size, dtype=int)
177
178         # iterate over all local particles to allocate them to send_** if
                they belong in another rank
179         for i in range(particle_n):
180             # if particle is active (still a local particle) and should
                    be sent to a rank (-1 means that the particle already is
                    in the correct rank)
181             if (particle_active[i] and send_to[i] != -1):
182                 # fill the temporary communication arrays (send_**) with
                        particle and it's properties
183                 send_id[send_to[i]][send_count[send_to[i]]] = i
184                 send_x[send_to[i]][send_count[send_to[i]]] = particle_x[i
                        ]
185                 send_y[send_to[i]][send_count[send_to[i]]] = particle_y[i
                        ]
186
187                 # deactivate sent particle
188                 particle_active[i] = False
189                 # increment counter to update position in temporary
                        communication arrays (send_**)
190                 send_count[send_to[i]] = send_count[send_to[i]] + 1
191
192         # actual exchange of particle properties follows
193
194         # must convert the list of arrays which are to be communicated to
                numpy objects (byte-like objects)
195         # this is not done before because np.ndarrays does not support a
                "list of arrays" if the arrays does not have equal dimensions
196         #send_id_np = np.array(send_id)
197         #recv_id_np = np.array(recv_id)
198         #send_x_np = np.array(send_x)
199         #recv_x_np = np.array(recv_x)
200         #send_y_np = np.array(send_y)
```

```
201     #recv_y_np = np.array(recv_y)
202
203     # requests to be used for non-blocking send and receives
204     send_request_id = [0] * mpi_size
205     send_request_x = [0] * mpi_size
206     send_request_y = [0] * mpi_size
207
208     recv_request_id = [0] * mpi_size
209     recv_request_x = [0] * mpi_size
210     recv_request_y = [0] * mpi_size
211
212     # sending
213
214     for irank in range(mpi_size):
215         if (irank != rank):
216             # number of particles rank sends to irank
217             Nsend = send_n_global[rank, irank]
218             # only receive if there is something to recieve
219             if (Nsend > 0):
220                 #print('rank:', rank, 'sending', Nsend, 'particles to
                        ', irank)
221                 # use tags to separate communication of different
                        arrays/properties
222                 # tag uses 1-indexing so there will be no confusion
                        with the default tag = 0
223                 send_request_id[irank] = communicator.isend(send_id[
                        irank][0:Nsend], dest = irank, tag = 1)
224                 send_request_x[irank]  = communicator.isend(send_x[
                        irank][0:Nsend],  dest = irank, tag = 2)
225                 send_request_y[irank]  = communicator.isend(send_y[
                        irank][0:Nsend],  dest = irank, tag = 3)
226
227     # receiving
228
229     for irank in range(mpi_size):
230         if (irank != rank):
231             # number of particles irank sends to rank (number of
                    particles rank recieves from irank)
232             Nrecv = send_n_global[irank, rank]
233             # only receive if there is something to recieve
234             if (Nrecv > 0):
235                 #print('rank:', rank, 'receiving', Nrecv, 'particles
                        from', irank)
236                 buf_id = np.zeros(Nrecv+buffer_overhead, dtype = np.
                        int64)
237                 buf_x = np.zeros(Nrecv+buffer_overhead, dtype = np.
                        float64)
238                 buf_y = np.zeros(Nrecv+buffer_overhead, dtype = np.
                        float64)
```

```
239
240                 # use tags to separate communication of different
                        arrays/properties
241                 # tag uses 1-indexing so there will be no confusion
                        with the default tag = 0
242                 recv_request_id[irank] = communicator.irecv(buf =
                        buf_id, source = irank, tag = 1)
243                 recv_request_x[irank]  = communicator.irecv(buf =
                        buf_x,  source = irank, tag = 2)
244                 recv_request_y[irank]  = communicator.irecv(buf =
                        buf_y,  source = irank, tag = 3)
245
246     # obtain data from completed requests
247     # only at this step is the data actually returned.
248     for irank in range(mpi_size):
249         if irank != rank:
250             # if there is something to receive
251             if send_n_global[irank, rank] > 0: # Nrecv > 0
252                 recv_id[irank][:] = recv_request_id[irank].wait()
253                 recv_x[irank][:] = recv_request_x[irank].wait()
254                 recv_y[irank][:] = recv_request_y[irank].wait()
255
256     #print('recv_id_np:', recv_id_np)
257     #print("recv_x_np:", recv_x_np)
258     #print("recv_y_np:", recv_y_np)
259
260     # make sure this rank does not exit until sends have completed
261     for irank in range(mpi_size):
262         if irank != rank:
263             # if there is something to send
264             if send_n_global[rank, irank] > 0: # Nsend > 0
265                 send_request_id[irank].wait()
266                 send_request_x[irank].wait()
267                 send_request_y[irank].wait()
268
269     # total number of received and sent particles
270     # total number of active particles after communication
271     sent_n      = int(np.sum(send_n_global, axis = 1)[rank])
272     received_n  = int(np.sum(send_n_global, axis = 0)[rank])
273     active_n    = int(np.sum(particle_active))
274
275     # move all active particles to front of local arrays
276     if (active_n > 0):
277         particle_id, particle_x, particle_y, particle_active =
                move_active_to_front(particle_id, particle_x, particle_y,
                particle_active, active_n)
278
279     # resize local arrays if needed
280
```

```
281     # current scaling factor = 1.25, set in top of file
282
283     # check if local arrays have enough free space, if not, allocate
            a 'scaling_factor' more than needed
284     if (active_n + received_n > particle_n):
285         new_length = int(np.ceil((active_n + received_n)*
                scaling_factor))
286         # if new length is not equal old length: resize all local
                arrays
287         if new_length != particle_n:
288             #print('extending arrays to new length:', new_length)
289             # with .resize-method, missing/extra/new entries are
                    filled with zero (false in particle_active)
290             ### TODO: change from resize function to method
291             particle_active = np.resize(particle_active, new_length)
292             particle_id     = np.resize(particle_id, new_length)
293             particle_x      = np.resize(particle_x, new_length)
294             particle_y      = np.resize(particle_y, new_length)
295
296             # particle_active.resize(new_length, refcheck = False)#
                    refcheck = True by default
297             # particle_id.resize(new_length, refcheck = False)
298             # particle_x.resize(new_length, refcheck = False)
299             # particle_y.resize(new_length, refcheck = False)
300
301     # check if local arrays are bigger than needed (with a factor:
            shrink_if = 1/scaling_factor**3)
302     # old + new particles < shrink_if*old_size
303     # if they are, shrink them with a scaling_factor
304     if (active_n + received_n < shrink_if*particle_n):
305         new_length = int(np.ceil(particle_n/scaling_factor))
306         # if new length is not equal old length: resize all local
                arrays
307         if new_length != particle_n:
308             #print('shrinking arrays to new length:', new_length)
309             ### TODO: change from resize function to method
310             particle_active = np.resize(particle_active, new_length)
311             particle_id     = np.resize(particle_id, new_length)
312             particle_x      = np.resize(particle_x, new_length)
313             particle_y      = np.resize(particle_y, new_length)
314
315             # particle_active.resize(new_length, refcheck = false)#
                    refcheck = true by default
316             # particle_id.resize(new_length, refcheck = false)
317             # particle_x.resize(new_length, refcheck = false)
318             # particle_y.resize(new_length, refcheck = False)
319
320     # add the received particles to local arrays
321
```

```
322     # unpack (hstack) the list of arrays, (ravel/flatten can not be
            used for dtype=object)
323
324     if received_n > 0:
325         particle_id[active_n:active_n+received_n] = np.hstack(recv_id
                )
326         particle_x[active_n:active_n+received_n] = np.hstack(recv_x)
327         particle_y[active_n:active_n+received_n] = np.hstack(recv_y)
328     # set the received particles to active
329         particle_active[active_n:active_n+received_n]  = np.ones(
                received_n, dtype = np.bool)
330
331     # optional printing for debugging
332     # print values for debugging
333         #print('particle_n (old value):', particle_n)
334         #print("old active_n:", active_n)
335         #print("sent_n:", sent_n)
336         #print("received_n:", received_n)
337         #print("new active_n:", np.sum(particle_active))
338         #print('new length of local arrays:', np.size(particle_id))
339         #print("new local particles:", particle_id)
340         #print("new active particles:", particle_active*1) # *1 to
                turn the output into 0 and 1 instead of False and True
341     else:
342         print("\nno received particles")
343
344     # print global array
345     if rank == 0:
346         print('\nglobal_array:\n', send_n_global)
347
348     # return the updated particle arrays
349     return (particle_id,
350             np.array([  particle_x,  # x component
351                         particle_y]),# x component
352             particle_active)
```

```
#parameters.py

nCells       = 8192#2048
nParticles   = 1e4#1e5
# nParticles will be rounded to the nearest perfect square
#t_max in seconds
t_max        = 10000#9.5#100

# all the other parameters are set directly in the other files
```

```
1  # script with functions to use in main
2  # this script handle the 'saving, loading and initialising of
       particle files'
3  # IO.py
4
5  # particle arrays will be initialized in input folder if this script
       is run by itself
6
7  import numpy as np
8  import os # for IO-paths
9
10 ##  INITIALISING start
11 file_extension = '.npy'
12 input_folder = 'input'
13 output_folder = 'output'
14
15 particle_x_name = 'particle_x'
16 particle_y_name = 'particle_y'
17 id_name = 'particle_id'
18 active_name = 'particle_active'
19
20 ## INITIALISING end
21
22 ## VARIABLES start
23
24 # relative paths
25 cdir = os.path.dirname(os.path.realpath(__file__))
26 IO_path_input  = os.path.join(cdir, input_folder)
27 IO_path_output = os.path.join(cdir, output_folder)
28 ## VARIABLES end
29
30 ## FUNCTIONS start
31 def save_array_binary_file_input(array, name, time, rank):
32     file_path = os.path.join(IO_path_input, 'time%s_%s_rank%s' % (
          time, name, rank))
33     np.save(file_path, array)
34
35 def save_array_binary_file_output(nPart, array, name, time, rank,
       mpi_size):
36     file_path = os.path.join(IO_path_output, 'nPart%s_time%s_%s_rank%
          s_MPIsize%s' % (nPart, time, name, rank, mpi_size))
37     np.save(file_path, array)
38
39 ######
40 # this function is not used, the load is done directly in "
       load_grid_of_particles_input"
41 def load_array_binary_file_input(name, time, rank):
42     file_path = os.path.join(IO_path_input, 'time%s_%s_rank%s%s' % (
```

```
        time, name, rank, file_extension))
43      return np.load(file_path)
44
45  # this function is not used, the load is done directly in "
        load_grid_of_particles_output"
46  def load_array_binary_file_output(name, time, rank, mpi_size):
47      file_path = os.path.join(IO_path_output, 'time%s_%s_rank%
            s_MPIsize%s%s' % (time, name, rank, mpi_size, file_extension))
48      return np.load(file_path)
49  ######
50
51  # when saving in
52  def save_grid_of_particles_input(ids, active, XY, time, rank):
53      # XY is a two-component vector [x, y]
54      save_array_binary_file_input(ids,      id_name,          time, rank
            )
55      save_array_binary_file_input(active,   active_name,      time, rank
            )
56      save_array_binary_file_input(XY[0,:], particle_x_name, time, rank
            )
57      save_array_binary_file_input(XY[1,:], particle_y_name, time, rank
            )
58
59  def save_grid_of_particles_output(nPart, ids, active, XY, time, rank,
        mpi_size):
60      # XY is a two-component vector [x, y]
61      save_array_binary_file_output(nPart, ids,      id_name,
            time, rank, mpi_size)
62      save_array_binary_file_output(nPart, active,   active_name,
            time, rank, mpi_size)
63      save_array_binary_file_output(nPart, XY[0,:], particle_x_name,
            time, rank, mpi_size)
64      save_array_binary_file_output(nPart, XY[1,:], particle_y_name,
            time, rank, mpi_size)
65
66  def save_empty_grid_input(time, rank):
67      save_array_binary_file_input(np.ndarray(0, dtype=int ),    id_name
            ,          time, rank)
68      save_array_binary_file_input(np.ndarray(0, dtype=bool),
            active_name,      time, rank)
69      save_array_binary_file_input(np.ndarray(0),
            particle_x_name, time, rank)
70      save_array_binary_file_input(np.ndarray(0),
            particle_y_name, time, rank)
71
72  def load_grid_of_particles_input(rank, time):
73      # XY is a two-component vector [x, y]
74      file_path_id = os.path.join(IO_path_input, 'time%s_%s_rank%s%s' %
            (time, id_name, rank, file_extension))
```

```
75      file_path_active = os.path.join(IO_path_input, 'time%s_%s_rank%s%
            s' % (time, active_name, rank, file_extension))
76      file_path_x = os.path.join(IO_path_input, 'time%s_%s_rank%s%s' %
            (time, particle_x_name, rank, file_extension))
77      file_path_y = os.path.join(IO_path_input, 'time%s_%s_rank%s%s' %
            (time, particle_y_name, rank, file_extension))
78      return (np.load(file_path_id),
79              np.load(file_path_active),
80              np.array([np.load(file_path_x), np.load(file_path_y)])
81              )
82
83  def load_grid_of_particles_output(nPart, rank, time, mpi_size):
84      # XY is a two-component vector [x, y]
85      file_path_id     = os.path.join(IO_path_output, 'nPart%s_time%s_%
            s_rank%s_MPIsize%s%s' % (nPart, time, id_name, rank, mpi_size,
             file_extension))
86      file_path_active = os.path.join(IO_path_output, 'nPart%s_time%s_%
            s_rank%s_MPIsize%s%s' % (nPart, time, active_name, rank,
            mpi_size, file_extension))
87      file_path_x      = os.path.join(IO_path_output, 'nPart%s_time%s_%
            s_rank%s_MPIsize%s%s' % (nPart, time, particle_x_name, rank,
            mpi_size, file_extension))
88      file_path_y      = os.path.join(IO_path_output, 'nPart%s_time%s_%
            s_rank%s_MPIsize%s%s' % (nPart, time, particle_y_name, rank,
            mpi_size, file_extension))
89      return (np.load(file_path_id),
90              np.load(file_path_active),
91              np.array([np.load(file_path_x), np.load(file_path_y)])
92              )
93
94  def create_grid_of_particles(N, w):
95      # create a grid of N evenly spaced particles
96      # N is rounded down to nearest perfect square
97      N = (int(np.sqrt(N)))**2
98      print('Creating grid of particles')
99      print('N: %s, w: %s' % (N, w))
100     # covering a square patch of width and height w
101     # centered on the region 0 < x < 2, 0 < y < 1
102     ids = np.arange(N)
103     active = np.ones(N, dtype=bool)
104     x  = np.linspace(1.0-w/2, 1.0+w/2, int(np.sqrt(N)))
105     y  = np.linspace(0.5-w/2, 0.5+w/2, int(np.sqrt(N)))
106     x, y = np.meshgrid(x, y)
107     return ids, active, np.array([np.ravel(x), np.ravel(y)])
108 ## FUNCTIONS end
109
110 if __name__ == '__main__':
111     from parameters import nParticles
112     # initialize grid
```

```
113     n = nParticles
114     w = 0.1
115     t = 0
116     total_ranks = 64*2
117     i, a, xy = create_grid_of_particles(n, w)
118     save_grid_of_particles_input(i, a, xy, t, rank=0)
119     # save empty grids for the other ranks, in this case for 64 total
            ranks
120     for i in range(1, total_ranks):
121         print('creating empty arrays for rank:', i)
122         save_empty_grid_input(t, i)
123     # writing information of nParticles
124     n = (int(np.sqrt(n)))**2
125     print('N was rounded to: ', n)
126     text_file = open("nPart-%s-was_created.txt" % n, "w")
127     text_file.close()
```

```
1  # script with plot functions to use in main or stand alone
2  # this script handles plotting of particles
3  # plot.py
4
5  import os
6  import numpy as np
7  import matplotlib.pyplot as plt
8
9  import seaborn as sns # plot style
10
11 import IO
12
13 plt.style.use('bmh')
14
15 ##  INITIALISING start
16
17 figsize_x = 12
18 figsize_y = 6
19 # specify x- and y limits,
20 x_min = 0
21 x_max = 2
22 y_min = 0
23 y_max = 1
24
25 # file extension
26 file_extension = '.pdf'
27 #file_extension = '.png'
28
29 ## INITIALISING end
30
31 ## FUNCTIONS start
32
33 # XY is a two-component vector [x, y]
34 def plot(rank, XY, time, dt, name = ''):
35     fig = plt.figure(figsize = (figsize_x, figsize_y))
36     #plt.scatter(x, y, lw = 0, marker = '.', s = 1)
37     plt.scatter(XY[0,:], XY[1,:], marker = '.', s = 5)#, linewidth =
           1)# s = size
38     # add text showing time, and set plot limits
39     plt.text(1.65, 0.9, 'rank = %s\n$t = %s$' % (rank, time), size =
           36)
40     plt.xlim(x_min, x_max)
41     plt.ylim(y_min, y_max)
42     #plt.show()
43     plt.savefig(os.path.join('plots', 'particles_time%s%s_rank%s_dt%s
           %s' % (time, name, rank, dt, file_extension)))
44
45 def plot_from_files(nPart, time, mpi_size, dt):
```

```
46      fig = plt.figure(figsize = (figsize_x, figsize_y))
47      colors = 'brcykgmk'
48      for rank in range (mpi_size):
49          # load particles
50          id, active, XY = IO.load_grid_of_particles_output(nPart, rank
                , time, mpi_size)
51          plt.scatter(XY[0,active], XY[1,active], marker = '.', s = 6,
                label = 'rank = %s' % rank, color = colors[rank])#label =
                'rank = %s' % rank)#, linewidth = 1)# s = size
52
53          #plt.text(iter(['rank 0', 'rank 1', 'rank 2', 'rank 3']))
54          #plt.text(1.65, 0.9, 'rank = %s\n$t = %s$' % (rank, time),
                size = 36)
55          # print('rank', rank)
56          # print('x', XY[0,active])
57          # print('\n')
58          # print('y', XY[1,active])
59
60      plt.xlim(x_min, x_max)
61      plt.ylim(y_min, y_max)
62      plt.legend(scatterpoints = 1)
63      #plt.show()
64      plt.savefig(os.path.join('plots', 'allranks_particles_time%s_dt%s
            %s' % (time, dt, file_extension)))
65
66  ## FUNCTIONS end
67
68  if __name__ == '__main__':
69      import IO
70      from parameters import nParticles
71      nParticles = int(nParticles)
72
73      dt= 0.5
74      file_extension = '.png'
75      t = 0
76      total_ranks = 4
77      figsize_x = 12
78      figsize_y = 6
79
80      # plot all particles from files and colour each rank
81      plot_from_files(nParticles, t, total_ranks, dt)
```