NTNU

Norwegian University of
Science and Technology

A comparison of two parallel solution schemes for the advection-diffusion equation

Velocity guided random walk of paticles vs
finite volume discretization on Cartesian grid

Fredrik Pedersen

Master of Science in Physics and Mathematics
Submission date: July 2017
Supervisor: Jon Andreas Støvneng, IFY
Co-supervisor: Tor Nordam, SINTEF

Norwegian University of Science and Technology
Department of Physics

Abstract

In this thesis two methods for solving the diffusion-advection equation was implemented using MPI. The two methods was a velocity guided random walk of particles and solving the advection-diffusion equation on a grid. Both systems were solved for a oscillating double vortex velocity system with reflective boundaries. Both implementations were shown to give similar solutions of the advection-diffusion equation, up to numerical resolution. The computation time of the model using velocity guided random walk of particles is highly affected by the mapping method from particles to grid, as the mapping influence the number of particles needed to get a sufficient picture. Using a "boxcount" method for mapping the particles, the total number of particles needed becomes so large that computational time for the particle model is significantly larger than the grid model. An alternative approach, where each particle was mapped onto the grid as a truncated Gaussian was also investigated. The code were tested on Vilje, the supercomputer located at NTNU. The implementation of both methods show reasonable scaling as the number of the number of ranks increase up to 512 and 1024. The grid model exhibit super linear speedup, probably due to better usage of cache as the number of ranks increase.

Sammendrag

I denne mastergraden ble det implementert to metoder for å løse adveksjon-diffusjons-ligningen ved hjelp av MPI. De to metodene var partisk virrevandring av partikler og å løse adveksjon-diffusjons ligningen på et grid. Begge implementasjoner ble løst for et hastighetsfelt som inneholdt en oscillerende dobbelt virvel strøm med reflektive grense-betingelser. Begge implementasjonene viste seg å gi tilsvarende løsninger på adveksjon-diffusjons-ligingen gitt det overnevnte hastighetsfeltet, opp til numerisk nyaktighet. Beregningstiden til metoden som benyttet seg av partisk virrevandring viste seg å være sterkt påvirket av avbildningen til griddet. Grunnen er at avbildningen direkte påvirker hvor mange partikler man trenger for å få et tilstrekkelig korrekt bilde. Kode ble testet på Vilje, superdatamaskinen til NTNU. Begge implementasjoner viser rimelig skalering mht at antall kjerner ble økt opp til 512 og 1024. Grid modellen utviser super lineær hastighetsøkning, sannsynligvis grunnet bedre utnyttelse av cache ettersom antallet kjerner øker.

Preface

> I don't know anything, but I do know that everything is interesting if you go into it deeply enough.
>
> *Richard Feynman, "The Smartest Man in the World" (1979)*

The work done in this thesis has been as a part of the endeavor of the master program in Applied physics and mathematics at NTNU, one of several master programs that culminate in the "sivilingeniør" title. The master thesis results in 30 ETCS credits and has been executed during the spring of 2017.

I would like to direct my gratitude towards Tor Nordam. His ever enlightening guidance, knowledge and humor, as a supervisor, has been much appreciated.

The "lunch group" of additional master students at Applied physics and mathematic, also deserves acknowledgement. It has been one of the highlights in an otherwise secluded experience to have the possibility to air challenges, accomplishments, jokes, and random fun-facts with fellow students throughout this spring.

Also during my studies of this particular transport phenomenon, I realized that this is a highly interesting subject. Hopefully this is not my last encounter with such an exciting theme.

Fredrik Pedersen

Table of Contents

Chapter 1

Introduction

Science is the poetry of reality.

Richard Dawkins, The Enemies of Reason, ”Slaves to Superstition”

Transport phenomena are involved in several physical and biological events that has implications on our lives. Given that there is a blow-out at an offshore installations in the North Sea or a tanker sinks, were would the resulting oil spill from the accident travel? If there is a high local resurgence of algae, where is the sudden bloom of microorganisms likely to be in a few days or weeks? The transportation of ash in the atmosphere after a volcano eruption affects the aerial transportation sector. The spread of salmon lice between fish farms is a transport phenomenon since the lice is mainly moved by the currents in the fjords.

Each one of these cases is an example of small particles or a localized concentration that moves and spreads out due to the behavior of the medium which the concentration inhabits. An illustrating concept is the behavior of a droplet of ink in a body of water, for instance a river. The ink molecules are advected due to the movement of the fluid, the river. At the same time the ink molecules are diffusing, due to the collision between the water molecules and the ink molecules.

These types of systems are typically large and sensitive to initial conditions. Hence simulations for these kind of systems are computationally demanding, and therefore the simulations are suitable to be run on high performance computers, such as supercomputers. The question arises to what is the most beneficial approach in terms of the required computational time, which depends on several factors. This will be a theme subjected to examination and discussion during the scope of this thesis.

In this section the the governing motivation for undertaking the work of this thesis, along with the main purpose will be presented. Also there will be an introduction to the advection-

diffusion equation. The following section will tackle parallel architecture, as the simulations in this thesis was executed on a supercomputer and a basic introduction is thus beneficial.

1.1 Purpose and the work done

The advection-diffusion equation is applicable to several applications, as the equation is present in both the industry and in scientific research. Such there are several approaches to solve this equation. Therefore it is interesting to study how two fundamental different approaches to solve the same problem perform, in terms of high performance computing, efficiency and complexity.

The work done in this thesis consists of solving the advection-diffusion equation for an analytical velocity field on a multi-core system using MPI. The advection-diffusion problem was solved using two methods: solving the advection-diffusion equation a grid using a finite volume method (see section 3.2) and utilizing velocity guided random walk of particles (see section 3.3). The scalability of both implementations were examined accordingly and also compared.

1.2 The Advection-Diffusion equation

The Advection-diffusion equation is

$$\frac{\partial C}{\partial t} = \nabla \cdot (D \nabla C) - \nabla \cdot (\mathbf{v} C)$$

[19], where C is a conserved quantity, D is the diffusion parameter and $\mathbf{v}$ is the velocity of the medium which the concentration inhibits. The advection-diffusion equation is essential for understanding and modeling physical transport behavior and describes how a physical quantity is transported inside a system by the effects of both diffusion and advection. The term $\nabla \cdot (D \nabla C)$ describes diffusion, a process that is the result of concentration difference, as the physical quantity will diffuse to areas where the concentration of is low. The term $\nabla \cdot (\mathbf{v} C)$ describes advection, which is transport of the physical quantity as a result of the flow or macroscopic motion of the system.

1.2.1 Different types of diffusion

When discussing diffusion in this thesis we talk about molecular diffusion. Molecular diffusion has the origin of the microscopic molecular motion of the molecules of a system. Turbulent diffusion has the origin in turbulent motion as non-laminar flow will lead to faster mixing of substances, compared to the case when the flow is laminar[17]. Turbulent motion is described by a nonlinearity and randomness of the flow, thus turbulent motion is neither fully understood nor fully mathematical describable. It should be noted that

turbulent diffusion will not be the focus of these thesis as only non-turbulent flow will be discussed and analyzed.

1.3 Parallell architeture

1.3.1 Von Neumann architeture

The standard template for a classical computer is the von Neuman architecture, which consists of a central processing unit (CPU), that does the computations, main memory that stores data and instructions, and a interconnect between the memory and the CPU. An analogy to explain the different parts is that the CPU is a factory and the memory is the storage facility for the factory. The interconnect is the highway that connects the two. Due to steadily increasing technological advances the von Neumann architecture is reaching its technological threshold in terms of computational performance. Some of the challenges of the von Neumann architecture are:

- the memory wall
- the power wall

The memory wall is the difference in operating speed of the CPU and memory due to the von Neumann bottleneck, that is how fast one can read from main memory is determined by the limitations of the interconnect [6], [20].

The decrease in size of the transistor has resulted in an advance in the transistor switching speed. However as the transistor performance increase in switching speed, the power consumption increases accordingly. With an increase in power consumption the resulting dissipation of heat increases, leading to higher temperatures. As temperatures increase it makes the transistor unstable. As this limits how many transistors a CPU can contain, this ”cascade” effect is called the power wall of the von Neumann architecture. For further reading and insights on the topic see [11], [16].

Improvements to the von Neumann architecture

It should be noted that even though the von Neumann architecture is the basic blueprint for computer architecture there are several improvements made to target the von Neumann bottleneck. Here the concept of cache will be presented as this is the improvement that is highly likely to influence the performance of the implementations of the gridded model of the advection-diffusion equation method in section 3.2, and the guided random walk of particles in section 3.3.

Cache is generally a collection of memory that is faster to access than some other memory location. In this setting when we discuss cache we talk about CPU cache. CPU cache is faster to access for the CPU than main memory. The cache can be located either on the same chip as the CPU, or on a separate chip that is faster to access than an ordinary memory

chip. The point is that the cache is physically closer to the CPU than main memory, and hence has a shorter access time. The concept of a cache block is essential to understand cache. We will start by explaining data locality, as data locality is an important aspect which the concept of a cache block is based upon, and the reason for implementing cache. Data locality is that a program typically uses data and instructions that are physical close in memory to recently used data and instructions[12]. Thus the cache reads a block of memory values into cache, where this block of memory is called the cache block. This usage of cache blocks is clever, since due to temporal and spatial locality, many of the values in the cache block is probably accessed by the CPU in the near future.

Conceptually the cache is divided into levels where the lowest level (L1) is the smallest and the fastest, whereas the higher levels (L2,L3,..) are slower and larger in size. According to Pacheco [12], as by 2010, most systems has at least two levels of cache, and three levels are common. As such when the CPU needs to access an instruction or data it starts with the lowest level of cache (L1) and works its way up to higher levels. If the instruction or data is not in the cache, the CPU accesses main memory to find what it is looking for. If the instruction or data is in the cache it is called a cache hit. Conversely if the data or instruction if not in cache and has to be fetched from main memory it is called a cache miss.

Another common improvement of of the von Neumann architecture is what is known as pipelining. Pipelining can be thought as a conveyor belt in terms of the number of instructions the CPU can do at the same time. Rather than processing each instruction sequentially, each instruction is split into a series of dependent steps so that different steps can be executed in parallel and instruction can be executed concurrently as a instruction is starting before the previous are finished.

1.3.2 Multi-processor systems

Multi-processor systems are computer systems with several processing units (processors) or cores, as the terms are used interchangeably. The systems based on a multi-processor architecture have the capability to bypass the memory wall and the power wall of the single processor systems based on the von Neumann architecture [12]. And hence these systems has the possibility to solve increasingly demanding computational problems. However multi-processor systems convey new challenges in form of communication between the processors, load balance and synchronization [12]. In the recent years the hardware architecture of desktop computers produce CPUs with several cores, often combined with multi-threading technology, such as the Intel Core i3, i5 or i7-series. Also mobile devices and phones sport multi-processor systems [1]. This demonstrates that multi-processor systems is becoming an important tool for solving computationally demanding problems and is already available as off-the-shelf commerce. Additional it should be noted that Graphics Processing Units (GPUs) are also parallel architecture, but GPUs are outside the scope of this thesis.

1.3.3 Shared memory architecture

In a shared memory architecture for multiple multi-core processors, all processors have a link via an interconnect to main memory or each processor has a direct link to a block of main memory, and the processors can access each others memory through special hardware [12]. One differentiates between uniform memory access (UMA) and nonuniform memory access (NUMA). In UMA systems, the time it takes to reach all memory locations will be the same for all cores. For the case of NUMA systems the memory locations that are directly connected to the core can be accessed more quickly than memory locations that must be accessed trough another chip.

1.3.4 Distributed memory architecture

In distributed memory architecture the individual computational units, called nodes, are joined together by a communications network which let the different nodes communicate with each other [12]. Individually each node typically has a shared memory architecture. In general such systems are heterogeneous as each node may be built from different types of hardware.

1.3.5 Flynn's taxonomy

Flynn's taxonomy is a classification for describing different hardware architectures. It classifies a system according to how many data streams and how many instruction streams the system can manage simultaneously [12]. Further it is worthy to mention that the classical von Neumann architecture is a single instruction stream, single data stream (SISD), wereas the Single Instruction Multiple Data (SIMD) systems are parallel systems. The SIMD applies the same instruction to multiple data items, and hence can be very efficient on large data parallel problems, but typically do not do well on other types of parallel problems. Another notable classification is the Single Program Multiple Data (SPMD). SPMD programs consists of a single executable that can behave as if it were multiple different programs through the use of conditional branches. SPDM can implement task-parallelism as it has the possibility to divide tasks among different processes.

1.3.6 Parallel programming for shared memory systems

OpenMP is a application programming interface (API) normally used for shared memory parallelization. MP stands for multiprocessing. The design is based on multi-threading, were a master thread forks a specified number of "slave threads" which joins at a later time [12]. This is governed by compiler directives and hence has a low amount of code overhead. Due to OpenMP beeing governed by the compiler, the developer of the code has therefore a limited control over memory access and communication when compared to developing programs for distributed memory.

1.3.7 Parallel programming for distributed memory systems

The standardized approach when programming for distributed memory systems, is to utilize Message Passing Interface (MPI). This is a specialization for a standard library for message-passing, that can be utilized from many common programming languages, such as C, C++, Fortran, Python etc. This enables communication between processors in a distributed system. MPI was initially developed with special consideration for library writers and application writers by the MPI Forum[1]. MPI lets the program developer have full control of all memory access and all communication between processors. As a result the code will have more overhead than its shared memory implementation equivalent. For further info see https://www.open-mpi.org/ or [12].

[1] http://mpi-forum.org/

Chapter 2

Theory

In this chapter we will first introduce the advection-diffusion equation and how this equation is discretized on a grid such that the equation can be solved numerically. Then the transition from a random walk model to a advection-diffusion equation will be demonstrated as validation that a model on biased random walk of particles will solve the same problem. Concepts of how to measure performance on parallel systems will also be presented, as these concepts are needed to compare the performance of the implementation for both models.

For this thesis the numerical schemes to approximate the advection-diffusion equation was chosen as explicit. Meaning that a general differential equation $\dot{\mathbf{x}} = f(t, \mathbf{x})$ is approximated on a grid as:

$$\dot{\mathbf{x}}_{n+1} = f(t, \mathbf{x}_n) + \mathbf{x}_n \tag{2.1}$$

instead of the implicit approximation:

$$\dot{\mathbf{x}}_{n+1} = f(t, \mathbf{x}_{n+1}) + \mathbf{x}_n \tag{2.2}$$

when the differential equation is spatially discretized, here $\mathbf{x}_n = n \cdot \Delta x$ where $\Delta \mathbf{x}$ is the grid spacing of the discretized variable $\mathbf{x}$.

Implications of the choice of scheme

Forward time difference as opposed to implicit scheme were one are needed to solve a system of equations, but as a return is unconditionally stable. With an an implicit scheme there are more floating point operations per time-step and thus are more computational heavy. The reason for choosing a explicit scheme is for straight forward implementation. Solving an implicit numerical scheme on a multi-dimensional grid is typical computationally demanding as one has one has to solve a system of equations. Often one has to make use of solvers and thus unforeseen problems can arise. The main focus in this thesis was

on results and implementation more than method. As a part of the work done during a master thesis is to limit and define the planned endeavor, explicit schemes where chosen to emphasize a doable implementation.

2.1 FVM for discretization of the advection-diffusion equation

Here the finite volume method (FVM) approach for discretization of the advection-diffusion equation on cartesian grid will be presented. As stated the Advection-Diffusion equation for a conserved quantity can be written as [19]

$$\frac{\partial C}{\partial t} + \nabla \cdot (\mathbf{v}C) = \nabla \cdot (D \nabla C) \tag{2.3}$$

where the D, C and $\mathbf{v}$ are the diffusion parameter, the conserved quantity and the velocity of the conserved quantity respectively. Here only cases for constant diffusion parameter will be discussed. The equation can be easily discretized over a two-dimensional Cartesian grid by the Finite Volume Method. When using the Finite Volume approach the conservation is enforced from the fact that flux out of a cell is flux in for a neighboring cell. Thus the content in every cell in the discretized domain must be conserved. As this conservation of flux is highly desirable when modeling the behavior of a physical system and one of the reasons for choosing FVM over i.e. a Finite Element Method.

Here the finite volume method and the finite difference method is equivalent. However both methods will be presented as it is a useful and instructive exercise, and FVM has some useful properties, so it is good to be aware of both.

The finite volume method is to integrate the equation over the discrete lattice volume defined by the discretization of the volume of interest.

The advection-diffusion equation integrated over the volume of interest, V, is:

$$\int \frac{\partial C}{\partial t} dA = \int_{\partial V} \nabla \cdot (D \nabla C) dA - \int_{\partial V} \nabla \cdot (\mathbf{v}C) dA \tag{2.4}$$

To solve equation 2.4 we utilize he divergence theorem. Thus:

$$\int \frac{\partial C}{\partial t} dV = \int \mathbf{n} \cdot (D \nabla C) dV - \int \mathbf{n} \cdot (\mathbf{v}C) dV \tag{2.5}$$

When evaluating

$$\int \mathbf{n} \cdot \mathbf{f}(c) dV \tag{2.6}$$

where

$$\mathbf{f}(c) = D \nabla C = (f_1(c), f_2(c)), \tag{2.7}$$

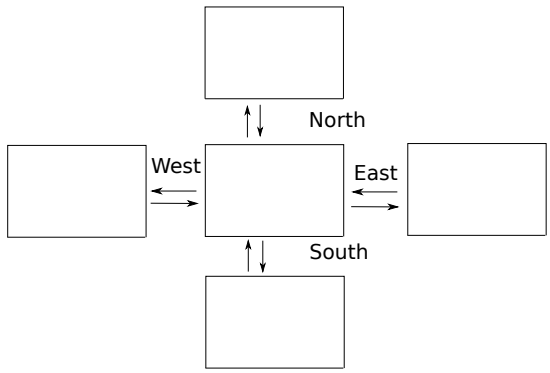


Figure 2.1: Show what gridcells that influence the next timestep fo the cell in the middle.

where $f_i(c)$ is the flux of the conserved quantity c in direction i.

We can take advantage of the fact that we can divide the total volume V into several smaller volumes. Then if eq. (2.5) is solved for all the cells, then eq. (2.3) is solved. The total volume is discretized into N_x cells in the x-direction and N_y cells in the y-direction. With the size of each cell as $\Delta x \cdot \Delta y$, were Δx is the cell-width in the x-direction and Δy is the cell-width in the y-direction. With the discrete variables, $x_i = i \cdot \Delta x, y_i = i \cdot \Delta y$ which in turn yields that $x_{N_x} = N_x \cdot \Delta x = x_{max}$ and $y_{N_y} = N_y \cdot \Delta y = y_{max}$, with the same for $t_k = k \cdot \Delta t$.This approximation for solving equation (2.5) is valid for when the total number of grid-cells goes toward infinity and the resulting size of each cell goes toward zero.

When evaluating the integral in eq (2.6) for a particular grid-cell, realizing that it is a flux integral gives a reason for wanting to find the solution at the boundaries of the grid-cell number i in x-direction and number j in y-direction.

Evaluating the flux integral at the east, north, west and south boundary (as seen in figure 2.1) accordingly:

$$\int_{\Gamma_e} \mathbf{n} \cdot \mathbf{f}(c) = \int_{y_{j-\frac{1}{2}}}^{y_{j+\frac{1}{2}}} f_1(x_{i+\frac{1}{2}}, y, t)dy \qquad \approx f_{1,i+\frac{1}{2},j}\Delta y \tag{2.8}$$

$$\int_{\Gamma_n} \mathbf{n} \cdot \mathbf{f}(c) = -\int_{x_{i-\frac{1}{2}}}^{x_{i+\frac{1}{2}}} -f_2(x, y_{j-\frac{1}{2}}, t)dx \qquad \approx -f_{2,i,j-\frac{1}{2}}\Delta x \tag{2.9}$$

$$\int_{\Gamma_w} \mathbf{n} \cdot \mathbf{f}(c) = -\int_{y_{j-\frac{1}{2}}}^{y_{j+\frac{1}{2}}} f_1(x_{i+\frac{1}{2}}, y, t)dy \qquad \approx -f_{1,i-\frac{1}{2},j}\Delta y \tag{2.10}$$

$$\int_{\Gamma_s} \mathbf{n} \cdot \mathbf{f}(c) = \int_{x_{i-\frac{1}{2}}}^{x_{i+\frac{1}{2}}} f_2(x, y_{j+\frac{1}{2}}, t)dx \qquad \approx f_{2,i,j+\frac{1}{2}}\Delta x \tag{2.11}$$

Here the number 1 and 2 denotes which of the elements, from the flux function in eq (2.7), in question. Further to evaluate the functions in eq (2.8) to eq (2.11) above we use the forward derivative scheme :

$$f_{1i+\frac{1}{2},j} = f_1(c_{i,j}, c_{i+1,j}) = D\frac{c_{i+1,j} - c_{i,j}}{\Delta x} \tag{2.12}$$

$$f_{1i-\frac{1}{2},j} = f_1(c_{i-1,j}, c_{i,j}) = D\frac{c_{i,j} - c_{i-1,j}}{\Delta x} \tag{2.13}$$

$$f_{2i,j+\frac{1}{2}} = f_2(c_{i,j}, c_{i,j+1}) = D\frac{c_{i,j+1} - c_{i,j}}{\Delta y} \tag{2.14}$$

$$f_{2i,j-\frac{1}{2}} = f_2(c_{i,j-1}, c_{i,j}) = D\frac{c_{i,j} - c_{i,j-1}}{\Delta y} \tag{2.15}$$

The second term in the equation 2.5 is derived by similar argument.

$$\int_{\Gamma_e} \mathbf{n} \cdot \mathbf{g}(c) \approx g_{1i+\frac{1}{2}.j}\Delta y \tag{2.16}$$

$$\int_{\Gamma_n} \mathbf{n} \cdot \mathbf{g}(c) \approx -g_{2i,j-\frac{1}{2}}\Delta x \tag{2.17}$$

$$\int_{\Gamma_w} \mathbf{n} \cdot \mathbf{g}(c) \approx -g_{1i-\frac{1}{2},j}\Delta y \tag{2.18}$$

$$\int_{\Gamma_s} \mathbf{n} \cdot \mathbf{g}(c) \approx g_{2i,j+\frac{1}{2}}\Delta x \tag{2.19}$$

$$\tag{2.20}$$

where $\mathbf{g}(\mathbf{c}) = \mathbf{v} \cdot C = (g_1(c), g_2(c))$. The following equations are found by taking the average of c at the boundaries.

$$g_{1i+\frac{1}{2},j} = g_1(c_{i,j}, c_{i+1,j}) = \frac{v_{i,j}(c_{i,j} + c_{i+1,j})}{2} \tag{2.21}$$

$$g_{1i-\frac{1}{2},j} = g_1(c_{i-1,j}, c_{i,j}) = \frac{v_{i,j}(c_{i-1,j} + c_{i,j})}{2} \tag{2.22}$$

$$g_{2i,j+\frac{1}{2}} = g_2(c_{i,j}, c_{i,j+1}) = \frac{v_{i,j}(c_{i,j} + c_{i,j+1})}{2} \tag{2.23}$$

$$g_{2i,j-\frac{1}{2}} = g_2(c_{i,j-1}, c_{i,j}) = \frac{v_{i,j}(c_{i,j-1} + c_{i+1,j})}{2} \tag{2.24}$$

When incorporating the forward time difference, the resulting scheme for the advection-diffusion equation is:

$$\begin{aligned} c_{i,j}^{n+1} &= c_{i,j}^{n} + r_1(c_{i+1,j} - 2c_{i,j} + c_{i-1,j}) \\ &+ r_2(c_{i,j+1} - 2c_{i,j} + c_{i,j-1}) - \frac{u_{1i,j}}{2}(c_{i+1,j} - c_{i-1,j}) - \frac{u_{2i,j}}{2}(c_{i,j+1} - c_{i,j-1}) \end{aligned} \tag{2.25}$$

where $r_1 = \frac{D\Delta t}{\Delta x^2}$, $r_2 = \frac{D\Delta t}{\Delta y^2}$ and $u_{1i,j} = \frac{\Delta t v_{1i,j}}{\Delta x}$, $u_{2i,j} = \frac{\Delta t v_{2i,j}}{\Delta y}$.

This scheme has the numerical stability criteria of [9].

$$0 < r_1 + r_2 \leq \frac{1}{2}$$

and

$$\frac{u_{1i,j}^2}{r_1} + \frac{u_{2i,j}^2}{r_2} \leq 2 \tag{2.26}$$

These dimensionless numbers relate the speed with which changes in c propagate, to the spatial and temporal resolution. Thus demanding that information do not jump over a grid-cell. We see the scheme is consistent as the scheme approaches the partial differential equation as $\Delta x, \Delta y, \Delta t \to 0$.

2.1.1 Finite difference scheme

There are other ways method for making partial differential equations solvable on a grid. One is the finite difference method. The finite difference method utilizes that for a function of a variable U defined on a grid, and by Taylor expansion of $U_{i,j}^{n+1}$ and $U_{i,j}^n$ while combining the previous expansions with the definitions of spacial and temporal derivative; $\frac{d}{dx_k} f(x_k) = \lim_{\Delta x_k \to 0} \frac{f(x_k + \Delta x_k) - f(x_k)}{\Delta x_k}$ and $\frac{d}{dt} f(t) = \lim_{\Delta t \to 0} \frac{f(t+\Delta t) - f(t)}{\Delta t}$ [4]. Then for a variable U the forward-discretization of the time-difference operator is

$$\frac{\partial U}{\partial t} = \frac{U_{i,j}^{n+1} - U_{i,j}^n}{\Delta t} \tag{2.27}$$

Corresponding the spatial derivative operator in the x-direction is

$$\frac{\partial U}{\partial x} = \frac{U_{i+1,j}^n - U_{i,j}^n}{\Delta x} \tag{2.28}$$

Similar the spatial derivation operator in the y-direction is

$$\frac{\partial U}{\partial y} = \frac{U_{i,j+1}^n - U_{i,j}^n}{\Delta y} \tag{2.29}$$

Applying the equation above on the equation 2.3 we get

$$\begin{aligned} c_{i,j}^{n+1} &= c_{i,j}^n + r_1(c_{i+1,j} - 2c_{i,j} + c_{i-1,j}) \\ &+ r_2(c_{i,j+1} - 2c_{i,j} + c_{i,j-1}) - \frac{u_{1i,j}}{2}(c_{i+1,j} - c_{i-1,j}) - \frac{u_{2i,j}}{2}(c_{i,j+1} - c_{i,j-1}) \end{aligned} \tag{2.30}$$

Also since the scheme for second order derivative is second order accurate and the scheme for first order derivative is first order accurate, the first order derivative is approximated by

the centered space method, as it is second order accurate. The centered space derivative method for a variable U in y direction is:

$$\frac{\partial U}{\partial y} = \frac{c_{i,j+1} - c_{i,j-1}}{2\Delta y} \tag{2.31}$$

Note that in the two-dimensional case, on a regular rectangular grid, the finite volume solution for the advection-diffuson equation is equal to the finite difference solution of the advection-diffusion equation.

2.1.2 Numerical stability

To explain numerical stability we shall utilize the theory developed for finite difference methods (FDM). Numerical stability of FDM is mainly studied using three approaches; Fourier analysis[1], the energy method and the normal mode analysis[2]. The energy method and the normal mode analysis are general techniques in contrast to the Fourier method, as it requires linear FDM with constant coefficients for periodic problems. Here only the Fourier method is considered for a one-dimensional case, as it motivates the understanding of numerical stability, however the derivation is easily expanded to higher dimensions. Consider the interval $[0, 1]$, with grid-points at $x_j = j \cdot \Delta x, j \in [0, NJ], \Delta x = \frac{1}{NJ}$ The values in a grid-cell can be expressed as a finite sum of Fourier modes:

$$C_j = \sum_{l=0}^{l=NJ-1} \hat{C}_l e^{2\pi l x_j}, j \in [0, NJ] \tag{2.32}$$

Granted there is assumed periodicity of 1, that is $T_0 = T_{NJ}$. The wavenumber is $k = 2\pi l$ and the index change from l to k. Thus:

$$C_j = \sum_k \hat{C}_k e^{kx_j}, \text{for } j \in [0, NJ] \text{ and } k \in [0, 2\pi, .., 2\pi(NJ-1)] \tag{2.33}$$

Inserting equation 2.33 into a FDM that is a one-step method, that the FDM contains only time-step n and $n+1$, combined with the fact that the vectors $[e^{kx_0}, ..., e^{kx_{NJ-1}}]^T$ are linearly independent for $k \in [0, 2\pi(NJ-1)]$ the following relation holds:

$$\hat{C}_k^{n+1} = g(k\Delta x, \Delta x, \Delta t)\hat{C}_k^n \tag{2.34}$$

Were $g(k\Delta x, \Delta x, \Delta t)$ is the amplification factor. The relation 2.34 implies:

$$\hat{C}_k^n = (g(k\Delta x, \Delta x, \Delta t))^n \hat{C}_k^0 \tag{2.35}$$

were the n on the right hand side is an exponent. The last relation 2.35 motivates the von Neumann condition: "A scalar one step FDM with constant coefficients and periodic boundary conditions is stable, if and only if there is a constant K such that

$$|g(k\Delta x, \Delta x, \Delta t)| \leq 1 + K\Delta t$$

[1] Which is also known as von Neumann stability analysis

[2] Also called GKS analysis, after Gustafsson, Kreis and Sundstrm [5]

for

$$\forall k, 0 < \Delta x \leq \Delta x_0,$$

and

$$0 < \Delta t \leq \Delta t_0 \tag{2.36}$$

If the amplitude factor is independent of Δx and Δt, the von Neumann condition is reduced to

$$|g(k\Delta x)| \leq 1 \tag{2.37}$$

The von Neumann condition is necessary and sufficient for numerical stability and lets there be stable schemes for exponentially growing functions [15].

2.2 From a random walk model to the advection-diffusion equation

Here there will be a transition from a random walk model to the advection-diffusion equation . For a collective of non-interacting particles that exhibit random walk, the individual particle trajectory can be simulated as:

$$\boldsymbol{x}_{i+1} = \boldsymbol{x}_i + \boldsymbol{r}_i\sqrt{2nD\delta t} \tag{2.38}$$

were D is the diffusivity, $\boldsymbol{x}_i = \boldsymbol{x}(t = i \cdot \delta t)$, δt is a short time-interval, $\boldsymbol{r}_i$, is a random vector in n-space such that the expectation value of the distribution moments is $\langle \boldsymbol{r}_i \rangle = 0$, $\langle \boldsymbol{r}_i \cdot \boldsymbol{r}_i \rangle = 1$ and $\langle \boldsymbol{r}_i \cdot \boldsymbol{r}_j \rangle = 0$ for $i \neq j$. [19]. It can be shown that [19]

$$\langle (\boldsymbol{x}_m - \boldsymbol{x}_0)^2 \rangle = 2nDt \tag{2.39}$$

where m an integer defining the time-step, such that $t = m\delta t$. This increasing width of the distribution variance, is a characteristic of diffusive processes.

For the case of velocity driven random walk; were the random walk happens in a medium that has a velocity field associated with itself, the position is also influenced by the velocity. The position of a particle in n-dimensions is at time $t = t_{i+1}$ given :

$$\boldsymbol{x}_{i+1} = \boldsymbol{x}_i + \boldsymbol{u}(\boldsymbol{x}_i)\delta t + \boldsymbol{r}_i\sqrt{2nD(\boldsymbol{x}_i)\delta t} \tag{2.40}$$

It can be shown, for timescales where $t \gg \delta t$, that the time evolution of the probability density function for the concentration is described by [19]

$$\frac{\partial c}{\partial t} = -\nabla \cdot (\boldsymbol{u}c) + \nabla^2(Dc) \tag{2.41}$$

where c is the probability density function for the concentration and $\boldsymbol{u}$ is the velocity vector-function exhibited by the particles. For neutrally buoyant particles this velocity

will be the same as the velocity for the medium.

It is important to note that for the general case, of D not being constant, this is not the Advection-Diffusion equation. This is shown by writing out the equation 2.41. Hence:

$$\frac{\partial c}{\partial t} = -\nabla \cdot (\boldsymbol{u}c) + \nabla \cdot (c\nabla D) + \nabla D(\nabla c) \tag{2.42}$$

The equation 2.42 contains an additional advective term $\nabla \cdot (c\nabla D)$ when compared to equation 2.3. This extra advective term transports particles down gradients of diffusivity. For physical diffusion this is deemed non-physical and hence a corrective term must be introduced. This is however not an issue for the case of constant diffusivity, as in this case the extra advective term becomes $\nabla \cdot (c\nabla D) = 0$. The existence of the extra advective term is also not a case when the velocity is independent of the diffusive constant, which will be the focus of this thesis. For further reading relating the corrective term see [19].

2.3 Performance measuring on parallel systems

2.3.1 Scalability of multiprocessor systems

The serial runtime, the execution time of a parallel program using one core, is defined as $T_{serial} = T(1)$. Whereas the parallel run-time is the execution-time using p cores, is defined as $T_{parallel} = T(p)$. The best parallel runtime is $T(p) = T(1)/p$. When this happens it is known as linear speed-up.

Speedup of a parallel program is defined as [12]

$$S = \frac{T(1)}{T(p)}. \tag{2.43}$$

When there is linear speedup we have $S = p$, and as such the term "linear speedup" make sense.

The efficiency of a parallel program is defined as

$$E = \frac{S}{p} = \frac{\frac{T_{serial}}{T_{parallel}}}{p} = \frac{T_{serial}}{p \cdot T_{parallel}} \tag{2.44}$$

According to Peter Pacheco [12] a program is strongly scalable if there is a rate at which the problem size can be increased so that, as the number of processors is increased, the efficiency remains constant. In contrast, the program is weakly scalable when the problem size needs to be increased at the same rate as the number of processes to keep constant efficiency.

Since parallel code is different from serial code, and needs additional program-overhead when compared to serial code, the parallel execution time of a program can be described as

$$T(p) = T_p = \frac{T_{parallel}}{p} + T_{overhead} \tag{2.45}$$

were $T_{parallel}$ is the part of the program that is parallelisable and hence that part will have its runtime affected by the number of processors, p, allocated to the execution of the program. $T_{overhead}$ is the part of the program that is needed to write parallel code or unaffected by the number of cores allocated. As the program size increases, the time used on code overhead becomes smaller compared to the time used on the parallel part of the program. Therefore $T_p \approx \frac{T_{parallel}}{p}$ as the problem-size becomes sufficiently large.

2.3.2 Amdahl's Law

Amdahl's Law [3] gives the theoretical speedup of the execution of a program with a fixed workload given that the resources (or number of cores) increase. The total runtime of the program can be written as

$$T = T_s + T_p \tag{2.46}$$

where T_s is the time of the serial part of the program and T_p the time of the parallel part of the program. And when f is the fraction of the program that is parallelizable, the equation (2.46) can be written as.

$$T = (1 - f)T + fT \tag{2.47}$$

and when allocating p number of cores the total runtime of the program yields

$$T(p) = (1 - f)T + \frac{f}{p}T \tag{2.48}$$

The resulting speedup is hence calculated by

$$S_A(p) = \frac{T}{T(p)} = \frac{1}{(1 - f) + \frac{f}{p}} \tag{2.49}$$

$S_A(p)$ is the theoretical speedup of the execution of the whole task. p can also be regarded as the speedup of the part of the task that benefits from improved resources. We see that as f becomes large, Amdahl's law becomes $S_A(p) \approx p$.

2.3.3 Gustafson's Law

Gustafson's Law gives the theoretical speedup of a program while having constant execution time, given that the resources (or cores) improve [8]. If the s_t and f_t represent accordingly the serial and the parallel time fraction of the time spent on *parallel* system such that $s_t + f_t = 1$. Then the time required doing the same computation on a serial system would be $s_t + f_t \cdot p$, were p was the total number of cores used in the parallel execution. The resulting scaled speedup is thus: [8]

$$S_G(p, s_t) = p + (1 - p) \cdot s_t \tag{2.50}$$

Gustafson law addresses the shortcomings of Amdahl's law. Since Amdahl's law only consider a fixed amount of work. But as resource increase the amount of work one wants

to do also increases. An another way to view Gustafson's law is, if the problem is scaled by a factor, by which factor do the resources need to be scaled, to keep the execution time constant.

If the we rewrite equation (2.50) to $S_G(p, s_t) = p + (1 - p) \cdot (1 - f_t)$, we see that when scaling the problem sufficiently large, the parallel time fraction is most significant i.e. $1 - f \approx 0$. Then Gustafson's law becomes $S_G(p, s_t) \approx p$.

2.3.4 Karp-Flat Metric

The Karp-Flat metric is a measure of to which extent the code is parallelized on a parallel processor system [10]. Hence the Karp-Flatt metric gives the empirical serial part of the program. Combining the expression for Amdahl's law in its simplest form:

$$T(p) = T_s + \frac{T_p}{p} \tag{2.51}$$

with the serial time fraction $s_t = \frac{T_s}{T(1)}$ of total time on a parallel system. Then equation (2.51) can be expressed as:

$$T(p) = T(1)s_t + \frac{T(1)(1 - s_t)}{p} \tag{2.52}$$

using the definition of speedup from (2.43)

$$\frac{1}{S} = s_t + \frac{1 - s_t}{p} \tag{2.53}$$

solving for the serial fraction gives the resulting equation

$$s_t = \frac{\frac{1}{S} - \frac{1}{p}}{1 - \frac{1}{p}} \tag{2.54}$$

s_t is the experimental serial fraction and as s_t becomes smaller, the more the code is parallelisable.

2.3.5 Optimal allocation of resources

There are two goals when doing large scale parallel simulations:

- The reason for using parallel programming is to get the results of the simulation as fast as possible.
- The simulations should utilize the smallest amount of resources when doing the computation. Hence being cost efficient.

The resulting best case scenario is when the serial fraction is 0. This is what is known as perfect scaling. Here the CPU-time is independent of the number of CPUs used for the simulation.

In practice the serial fraction is larger than 0. This means that as the number of cores, p, gets sufficiently large, we se from eq (2.49) that for a problem of fixed size the speedup becomes constant: $\lim_{p\to\infty} S_A(p) = \frac{1}{1-f}$. Also by taking the derivative of eq (2.49):

$$\frac{d}{dp}S_A(p) = \frac{f}{((1-f)p+f)^2} \tag{2.55}$$

Here we see that the slope of the speedup is steadily decreasing. As such there exist a point were allocating more recourses is not necessarily advantageous, even tough there will be an increase in speedup. An explanation could be to think in the terms of ”economic” motivation, at some point the cost of allocating new resources outweighs the gain from faster execution time of the program.

Chapter 3

Method

This chapter will entail the methods implemented for solving the diffusion equation presented in section 2.1 and 2.2. First the velocity field utilized in this thesis will be presented and discussed, as the properties of the velocity field will have certain consequences for the resulting implementations for solving the advection-diffusion equation. Then the chosen method for implementation of solving the advection-diffusion on a grid will be presented along with how to maintain the boundary conditions for this implementation. Further, the chosen method for implementing the velocity guided random walk for solving the advection-diffusion equation, will be presented. Also here the boundary condition will be discussed, as this method requires a different way of enforcing the boundaries. Then there follows a discussion of the differences between the domain-decomposition in the two models. The method for comparing the result in section 3.2 and 3.3 will be presented accordingly. The end of this chapter will entail a presentation of the system used for testing performance the implementations.

It should be noted that in this chapter the term *rank* refers to a single core or thread (also called process) in the parallel computing system.

Also the source code for implementing both the grid model method and the particle model method can be found in the appendix A

3.1 System geometry

The analytical velocity field used in the simulations consist of two counter-rotating gyres, where the centerline between them oscillates back and forth. The analytical expression for

the velocity field, taken from [13], is defined as:

$$\begin{aligned} v_x &= -\pi A \sin(\pi f(x,t)) \cos(\pi y) \\ v_y &= -\pi A \cos(\pi f(x,t)) \sin(\pi y) \frac{\partial f(x,t)}{\partial x} \end{aligned} \tag{3.1}$$

where $x \in [0, 2.0], y \in [0, 1.0]$ and

$$\begin{aligned} f(x,t) &= a(t)x^2 + b(t)x \\ a(t) &= \epsilon sin(\omega t) \\ b(t) &= 1 - 2\epsilon \sin(\omega t) \end{aligned} \tag{3.2}$$

The A determines the amplitude of the velocity field, ω is how fast the centerline, between the two gyres, oscillates back and forth. As seen in the vector plot of the velocity field in eq. (3.1) and (3.2) in figure 3.1, for different points in time. ϵ is a factor that determines the width of the oscillations belonging to the centerline. In figure 3.1 and the implementations the values were set to $A = 0.1, \omega = 1.0, \epsilon = 0.25$. The advantage of this particular analytical expression is that the velocity is always parallel to the boundaries. Hence there will not occur any advection across the boundary. As such the only possible way to cross the boundary is by diffusion.

In order to make the system conserve mass, reflective boundary conditions was enforced. This means that our system can be compared to a closed box, were an initially concentrated substance is distributed by advection and diffusion.

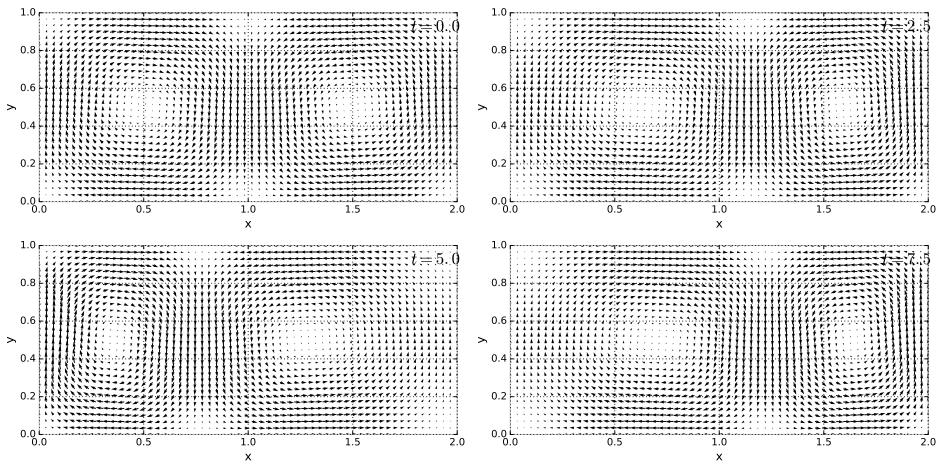


Figure 3.1: Plot of the velocity field specified in eq (3.1) and (3.2) for different points in time.

3.2 The grid model

As shown in section 3.2 the advection-diffusion equation in eq. (2.3) can be numerically approximated as eq. (2.25) and solved on a grid. As mentioned previously the system was solved using parallel computations. When doing computations in parallel, the primary objective has to be subdivided into tasks that can be computed in parallel, ideally independent of how many ranks are available to the program. For this model the total domain of the grid was divided into several sub-domains. One sub-domain for each rank and as such each rank was solving the eq. (2.25) for its own sub-domain. As each rank solved the equation for a different region of the domain this is an example of a SPDM system [12].

To make the communication between ranks as straightforward as possible, the ranks was organized as a Cartesian grid that overlapped the domain. MPI has built in support for this through what is known as a Cartesian Communicator. This is done by utilizing a MPI functions called `MPI_Dims_create` and `MPI_Cart_create`. As MPI has functionally to communicate between distributed memory systems, MPI has what is known as a *communicator* that is used to let a group of ranks communicate if they belong to the same communicator. `MPI_Cart_create` takes the arguments of how many ranks that are available along with a number specifying the dimensional configuration and returns a "dimension array", an array that contains the number of ranks in each dimension. `MPI_Cart_create` takes the dimension array as an argument and returns a Cartesian communicator. This Cartesian communicator is used to keep track of the neighbors of each rank in the north, south, east and west direction, as seen in figure 3.2.

Thus each rank had the computational responsibility for a square of the grid, with a size depending of how many ranks that was involved in the computation. By looking at eq (2.25), we see that $c_{i,j}^{t+1}$ depends on its neighboring grid-cells, as seen in figure 3.3. Therefore we run into a calamity when solving the eq (2.25) on the border cells of the sub-domain, since these are belonging to a different rank. To resolve this a "halo" of grid-cells was added around the sub-domain grid of each rank. The halo consists of the neighboring ranks boundaries, as seen in figure 3.2. For each rank, with neighboring ranks in all four directions, the halo consisted of two bordering columns and two border rows. For the ranks on the "edges" of the domain, with no neighboring rank in a certain direction, this part of the halo was used to enforce the boundary conditions. After each timestep, each rank exchanges updated values for the halo cells with its neighbors.

3.2.1 Initial conditions

The system initial condition were instantiated by the master rank and were a two-dimensional Gaussian distribution of concentration;

$$p(x, y) = A \exp\left(-\left(\frac{(x - x_0)^2}{2\sigma_x^2} + \frac{(y - y_0)^2}{2\sigma_y^2}\right)\right) \tag{3.3}$$

where $x \in [0, N_x], y \in [0, N_y], x_0 = \frac{N_x}{2}, y_0 = \frac{N_y}{2}, \sigma_x = \frac{N_x}{16}, \sigma_y = \frac{N_y}{8}$. The total size of the system is hence $N_y \cdot N_x$. A is a scaling factor. The initial condition were chosen to

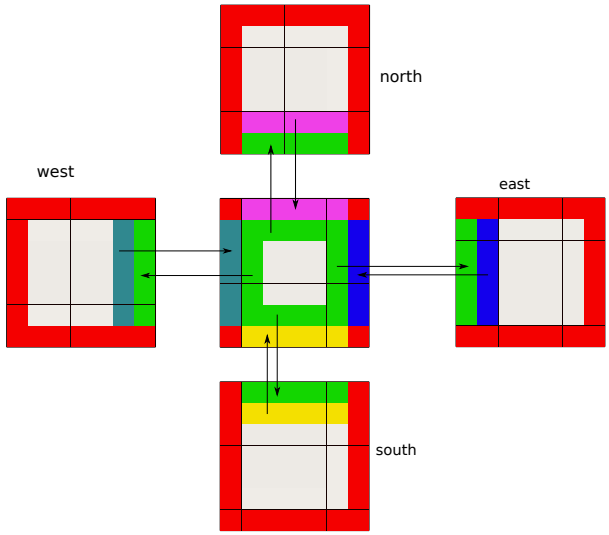


Figure 3.2: Shows border communication between processors in the domain decomposition of the diffution advection equation

be Gaussian as any discontinuities in the concentration are not beneficial for the numerical scheme in (2.25). As a result the initial condition for the particle model had to be similar.

3.2.2 Computational recipe

The following recipe was utilized when running a computation on p ranks;

- Divide the domain between the ranks and set up the Cartesian communicator.
- Initialize the concentration distribution on the master rank.
- Distribute the initial concentration between the p ranks.
- The next steps are performed on each rank for a number of N_{steps} time steps:
  - update the velocity field.
  - communicate between ranks by transferring the halos.
  - compute the concentration in each grid cell for the new time step by applying eq. (2.25).
  - At given time intervals the concentration is gathered at the master rank and written to file.

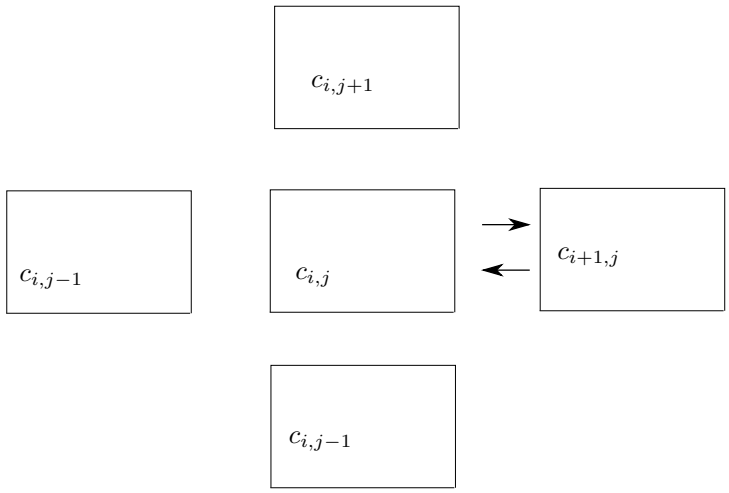


Figure 3.3: Illustration of the gridcell influencing a new timestep, for the scheme in the grid model. As seen in equation (2.25)

3.2.3 Enforcing boundary conditions

When assuring that the boundary conditions are met the value of the concentration at the boundaries were simply set to be the same values as the ones next to the boundaries. To be strictly formal the reflective boundaries is a Neumann boundary condition. The flux in the advection-diffuion equation is given by:

$$j = j_A + j_D = \mathbf{v} \cdot C - D\nabla C, \tag{3.4}$$

were j_A and j_D are the advective flux and the diffusive flux respectively. When $\frac{\partial C}{\partial x} = 0$ the diffusive flux, j_D, in x-direction is zero. The reason for this is that the flux out of a grid-cell is proportional with the concentration in the cell. If $c_{i,j} = c_{i,j+1}$ in figure 3.3, then the flux is equal in both directions, so that total flux is zero. Such is the spatial derivative of the concentration at the boundary is zero. Thus by demanding that:

$$\begin{aligned}
\frac{c_{N_x+1,j} - c_{N_x,j}}{\Delta x} &= 0, \ \forall\, j \in [0, N_y] \\
\frac{c_{0,j} - c_{-1,j}}{\Delta x} &= 0, \ \forall\, j \in [0, N_y] \\
\frac{c_{i,N_y+1} - c_{i,N_y}}{\Delta y} &= 0, \ \forall\, i \in [0, N_x] \\
\frac{c_{i,0} - c_{i,-1}}{\Delta y} &= 0, \ \forall\, i \in [0, N_x]
\end{aligned} \tag{3.5}$$

we see that the stencil in eq (2.25) on the edges of the domain becomes:

$$\begin{aligned} c^{n+1}_{N_x,j} &= c^n_{N_x,j} + r_1(c_{N_x,j} - 2c_{N_x,j} + c_{N_x-1,j}) + r_2(c_{N_x,j+1} - 2c_{N_x,j} + c_{N_x,j-1}) \\ &\quad - \frac{u_{1N_x,j}}{2}(c_{N_x,j} - c_{N_x-1,j}) - \frac{u_{2N_x,j}}{2}(c_{N_x,j+1} - c_{N_x,j-1}) \\ &\quad \forall j \in [0, N_y] \end{aligned} \tag{3.6}$$

$$\begin{aligned} c^{n+1}_{0,j} &= c^n_{0,j} + r_1(c_{1,j} - 2c_{0,j} + c_{0,j}) + r_2(c_{0,j+1} - 2c_{0,j} + c_{0,j-1}) - \frac{u_{10,j}}{2}(c_{1,j} - c_{0,j}) \\ &\quad - \frac{u_{20,j}}{2}(c_{0,j+1} - c_{0,j-1}) \\ &\quad \forall i \in [0, N_x] \end{aligned} \tag{3.7}$$

$$\begin{aligned} c^{n+1}_{i,j_{max}} &= c^n_{i,N_y} + r_1(c_{i+1,N_y} - 2c_{i,N_y} + c_{i-1,N_y}) + r_2(c_{i,N_y} - 2c_{i,N_y} + c_{i,N_y-1}) \\ &\quad - \frac{u_{1i,N_y}}{2}(c_{i+1,N_y} - c_{i-1,N_y}) - \frac{u_{2i,N_y}}{2}(c_{i,N_y} - c_{i,N_y-1}) \\ &\quad \forall j \in [0, N_y] \end{aligned} \tag{3.8}$$

$$\begin{aligned} c^{n+1}_{i,0} &= c^n_{i,0} + r_1(c_{i+1,0} - 2c_{i,0} + c_{i-1,0}) + r_2(c_{i,1} - 2c_{i,0} + c_{i,0}) - \frac{u_{1i,0}}{2}(c_{i+1,0} - c_{i-1,0}) \\ &\quad - \frac{u_{2i,0}}{2}(c_{i,1} - c_{i,0}) \\ &\quad \forall i \in [0, N_x] \end{aligned} \tag{3.9}$$

As seen some of the terms cancel eachother out. As such, the stencil in equations (3.6) - (3.9) show that to enforce the boundary conditions and thereby to change the computational stencil of eq (2.25) at the boundaries, is the same as setting the values at the boundaries equal to the values next to the boundaries at each timestep, without changing the stencil of eq (2.25). Here it possible to not change the stencil as each rank has its own halo.

3.3 Method of velocity guided random walk of particles

The particle model relies on the fact that a sufficiently large amount of particles that exhibit random walk in a velocity field will behave collectively as an advective-diffusive process, as covered in section 2.2. The following procedure was utilized on each rank when running a computation with N particles on p ranks;

- N/p particles were initiated on each rank. The initial conditions was chosen to be as similar as possible to that for the gridded equations: Each particle was located at a random position drawn from a Gaussian distribution with $\mu_x = 1.0$, $\mu_y = 0.5$, $\sigma_x = \sigma_y = 1/8$. Such that that the expected value is at the center of the domain.

- The next steps are performed on each rank for a number of N_{steps} time steps:
  - The particles were advected forward in time using the Runge-Kutta 4th-order method as the trajectory of each particle is given as $\dot{\mathbf{x}} = \mathbf{v}(\mathbf{x}, t)$, where $\mathbf{v}$ is the velocity of the particle given by eqs. (3.1) and (3.2).
  - After each advective step the particles were given a random displacement. This displacement fulfills the required conditions for the random vector described in section 2.2; and used in the equation $\boldsymbol{x}_{i+1} = \boldsymbol{x}_i + \boldsymbol{u}(\boldsymbol{x}_i)\delta t + \boldsymbol{r}_i\sqrt{2nD(\boldsymbol{x}_i)\delta t}$
- At given time intervals the position of the particles were sent to rank 0 and written to file.

3.3.1 Enforcing the boundary conditions

Due to the nature of the Gaussian distribution, some initial conditions will likely be outside the system domain. Therefore, the initial conditions of every particle was checked to be inside the system, if this was not the case the position was rejected, resulting in a new pick of position, and following an assertion that the new position is valid. This procedure was repeated for each particle until all the particles had a valid initial condition.

If a particle crossed the boundary of the domain due to the random walk, the position had to be corrected to enforce the reflective boundary condition. Thus the total distance traveled by the particle from the boundary was calculated, in the x-direction: $d_r = x - x_{max}$. The new position is then:

$$x \to x - 2 \cdot (x - x_{max}). \tag{3.10}$$

The "corrected" position is thus the distance d_r from the boundary, in the opposite direction, into the domain, as seen in figure 3.4. Hence it was as the particle had a elastic collision with a wall.

3.4 Domain decomposition vs particle decomposition

The main difference between how parallelization was utilized in the two models is how the overlying task is decomposed between the ranks available. For the grid-model when decomposing the simulated domain between the ranks and letting each rank solve one spatial part of the domain of interest, the strength of this approach is that each ranks does not need to know the velocity field over the entire system. Another advantage is that in the domain decomposition, each rank always know the local concentration. On the other hand this method of domain decomposition advocates a substantial amount of communication.

In the particle based method, each rank keep track of a fraction of the conserved quantity. Here the advantage lies on the fact that the number of times needed to communicate between the ranks is greatly reduced, when compared to the domain decomposition. A disadvantage is that particles in a given physical location can be distributed across many ranks. This means concentration cannot be calculated without communication. A hybrid

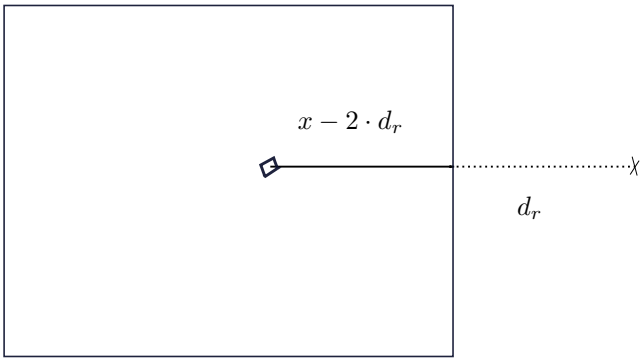


Figure 3.4: Shows how particles are moved into the domain if they cross the domain boundaries due to diffusive movement. Here it is only shown in the x-direction. The cross indicates the position of the particle initial position outside the domain. The box indicates the corrected position were the particle is moved inside the domain.

model, where particles are assigned to ranks according to their spatial position, is also possible. This has been investigated by Nordam et al [18].

3.5 Comparing the results

To compare the two models there were computed several plots. Concentration plot for the grid-model, at a given time, were made from a file containing the concentration and normalized before plotting and then presented.

When producing the concentration plot for the particle model, the particles positions that were stored to file were mapped to grid using the grid-spacing determined by the grid-model, hence the choice of $h = 1/(512 - 1)$, and the resulting plot was normalized and presented. The mapping was a ”boxcount” method, where if the a grid-cell contained a particle the value of this cell was incremented. As such each box contains a count of how many particles there are in each box. From this ”boxcount” a resulting heatmap was constructed. Also there were produced concentration plots from the particle model using a different mapping, a truncated Gaussian. Were each particle is mapped using a truncated Gaussian function of a given size, hence smoothing the plot.

To view the resulting differences between the plots a difference plot were constructed. The normalized concentration made from the grid-model were subtracted from the normalized

concentration made from the particle-model and the resulting ”difference” concentration were plotted.

Also computational time for the particle method as function of the number of particles was demonstrated. Timings of the wall-time for both implementations were presented along with resulting plots of speed-up, the Karp-Flat metric, CPU-time and efficiency, generated as a result of the number of ranks.

3.6 Performance testing

This two programs were implemented in C, and tested on Vilje [7], using the openMPI implementation of MPI. Vilje is a SGI Altix ICE X system procured by NTNU together with met.no and UNINETT Sigma, with SUSE Linux Enterprise Server 11 operating system and Mellanox FDR infiniband and Enhanced Hypercube Topology interconnect. Vilje has a total of 1404 nodes. Each node had 2 eight-core processors (Intel Xeon E5-2670) with a processor speed of 2.6 GHz. 8 cores share a L3 Cache of 20 MB. This gives a total of $1404 \cdot 16 = 22464$ cores for the complete system. The program were tested on up to 512 and 1024 ranks (64 nodes).

Chapter 4

Performance, results and discussion

First, we present an overview of the results in this chapter, as the result gained from the simulations done for both models will be compared and discussed. Then two different ways of mapping particles onto grid are considered, boxcount and truncated Gaussian kernel. We will investigate different ways of calculating a concentration grid from particle positions, and address the question of what is a suitable number of particles. Finally, performance characteristics of the models, as a result of the number of cores available, will be shown and discussed.

4.1 Overview of results

In figure 4.1, results from the grid model implementation are shown from 10 different timesteps, to give an idea of how the concentration field evolves trough time. Note that the scale of the colorbar is kept constant.

Later certain output timesteps will be examined and compared to the results from the particle model. We will refer to timesteps by number, where timestep 0 is the initial condition (Upper left panel of equation figure 4.1). It should be noted that *timestep* here means how many times particle position or concentration is been written to file, i.e. the *output timestep*. The internal timestep δt, (see eq. (2.25)) is much shorter.

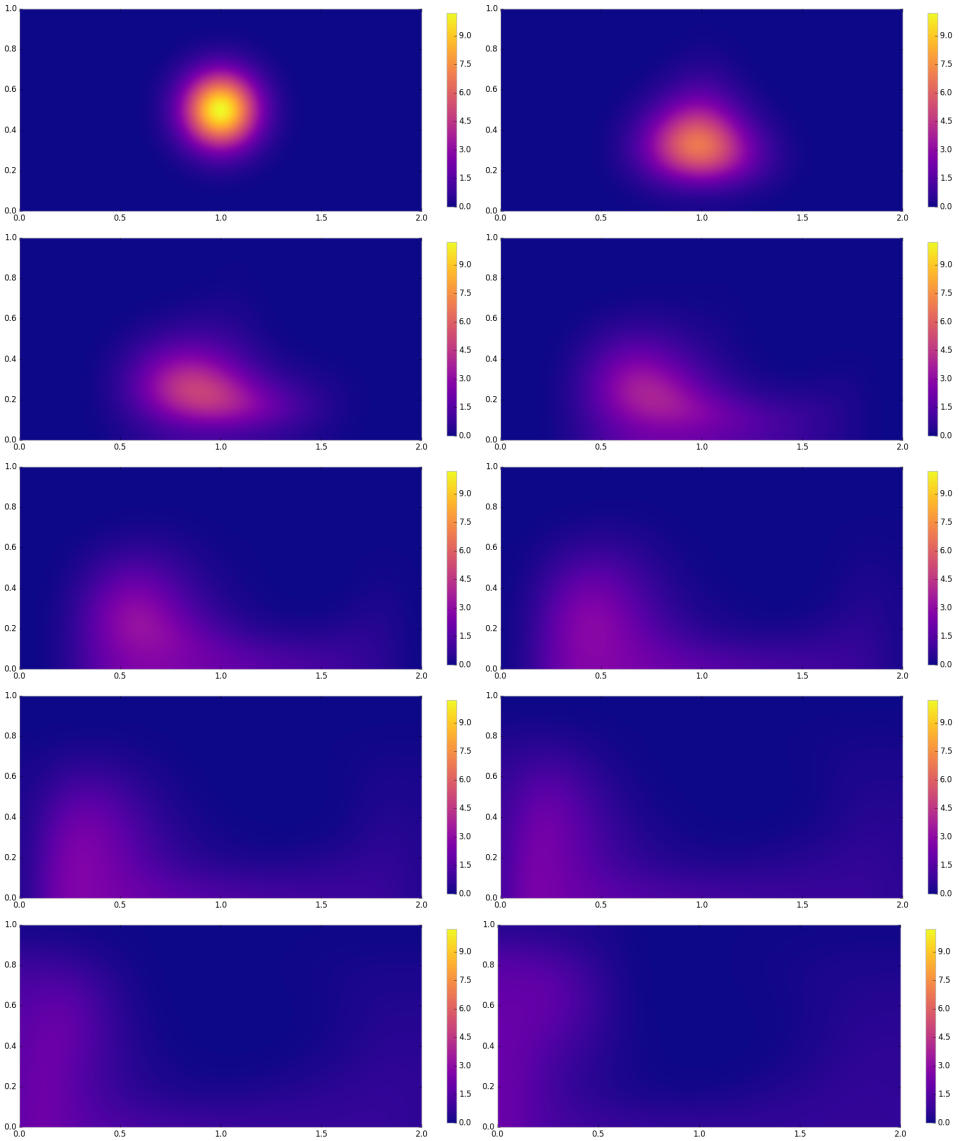


Figure 4.1: Results from the gridded model with $(Nx, Ny) = (1024, 2048)$. This plot show the time evolution of the concentration, as given by the advection-diffusion equation, solved over using the grid method. The initial condition was as described in eq. (3.3) ini a certain time period. Timestep 0 to 9.

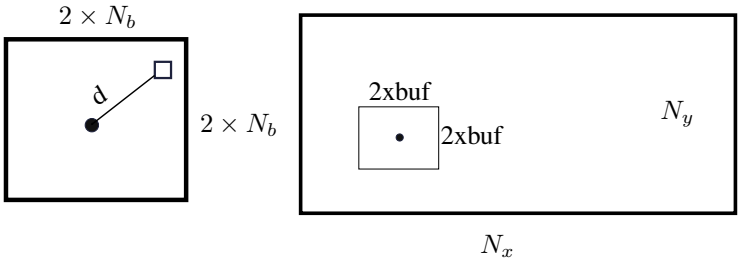


Figure 4.2: The domain of influence from each particle, when mapping the position of the particle to a grid by using eq (4.1) and eq (4.2).

4.2 Mapping particle onto grid

The following are a recipe for how each particle is counted, when mapping the particles to grid, using a truncated Gaussian, such that the results from two models can be compared.

- We first define a grid that covers the area $0 \leq x \leq 2, \leq y \leq 1$, with constant cell size. The number of cells are $N_x \times N_y$, where $N_x = 2048$ and $N_y = 1024$. The goal i to calculate the concentration in each cell, such that the particle model can be compared directly with the grid model.
- We then define a smaller grid, G, to represent the contribution from each particle. Each cell in the grid G is assigned a value $G(i, j)$:

$$G(i,j) = \frac{1}{\sqrt{2\pi}\sigma} \cdot e^{-\left(\frac{d^2}{2\cdot\sigma^2}\right)} \tag{4.1}$$

  where

$$d^2 = (i - N_b)^2 + (j - N_b)^2 \tag{4.2}$$

- Finally we loop over all the particles. For each particle we find the cell that is the center of the particle is located. Then G will be superimposed onto the concentration grid, such that the center of G is located a the center of the particle, and the contribution in each cell in G is added to the concentration grid.
- To assert the particles whose distance from the edge of the concentration grid than the distance σ (note that σ is given in number of cells), the contribution from the

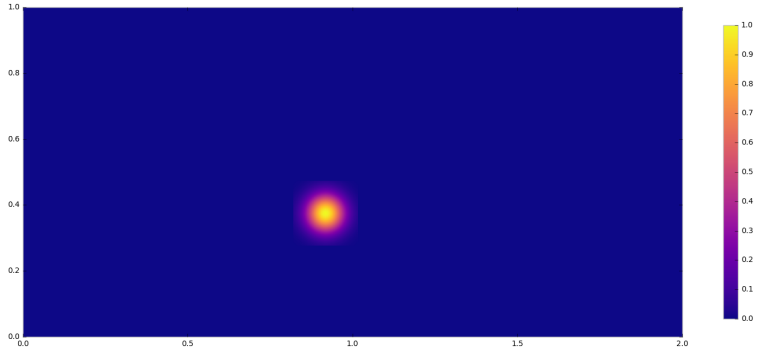


Figure 4.3: Illustration of the added Gaussian function for one particle.

cells that end up on the outside gets reflected such that they end up inside the concentration grid. This is necessary to ensure the mass in the concentration grid.

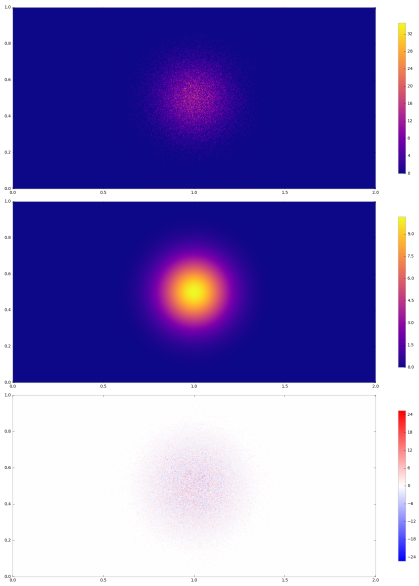
Note that $\sigma = (1, 20, 40)$ for the pictures (a,b,c) in figure 4.4 to 4.6 respectively. The concept of the truncated Gaussian kernel above is illustrated in figure 4.2 and the mapping of one particle can be seen in figure 4.3

A comment relating the execution time of the griding of particles is that when the grid-projection was implemented in python, using the Numba just-in-time compiler, the computational time to calculate the grid projection was insignificant compared to the time it took to run the particle model.

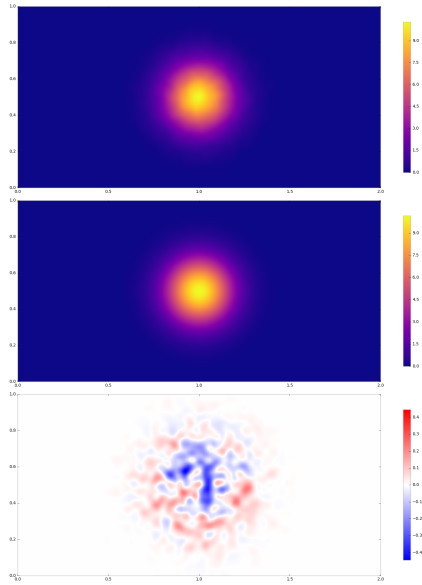
4.3 Visualizing model differences

As seen in figure 4.4 to 4.6 the behavior of the grid model and the particle model are similar. Here the plot for particle model is made in three different ways. In picture a) the particles are counted as a delta function, picture b) and c) maps each particle as a truncated Gaussian. Naming the method in a) "boxcount" is due to the fact that one counts the number of particles in each gridcell, and afterwards normalizes the total concentration plot. The difference between the particle plots in picture b) and c) is the σ in equation (4.1), were $\sigma = 20$ and $\sigma = 40$ in b) and c) respectively. Also the Gaussian function was the same for each particle.

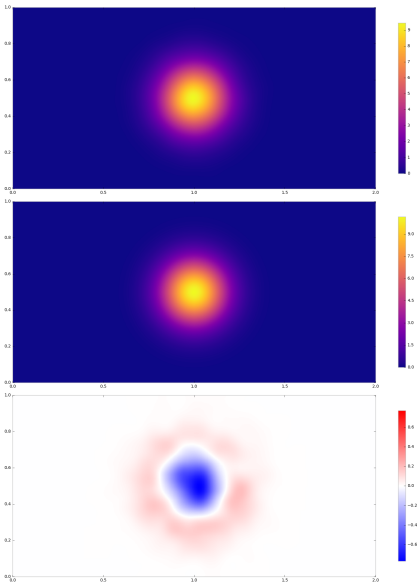
As seen in figure 4.7 the concentration when using the grid method witness some numerical diffusion. As this is unavoidable due to round of effects, it is well within acceptable limits since the loss of information is less than 0.25% during the entire simulation period. As for the particle method it is perfect conservative as it is highly unlikely that the loss of an entire data variable will occur. Since each particle in the implementation is assigned an element in an array, it would require the assigned memory of the data variable in question to be overwritten.



(a) Particle plotted using a boxcount method.

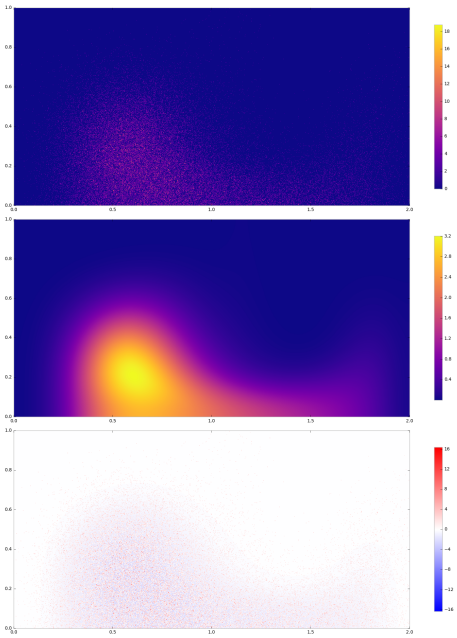


(b) Particles plotted using a Gaussian smoothing with $\sigma = 20$ (see eq (4.1) and eq (4.2)).

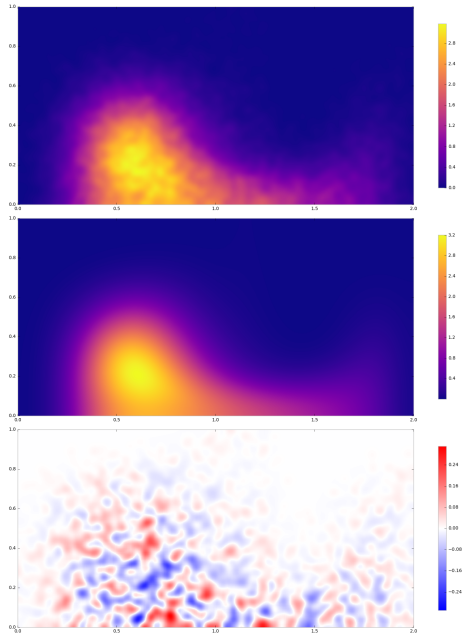


(c) Particles plotted using a Gaussian smoothing with $\sigma = 40$ (see eq (4.1) and eq (4.2)).

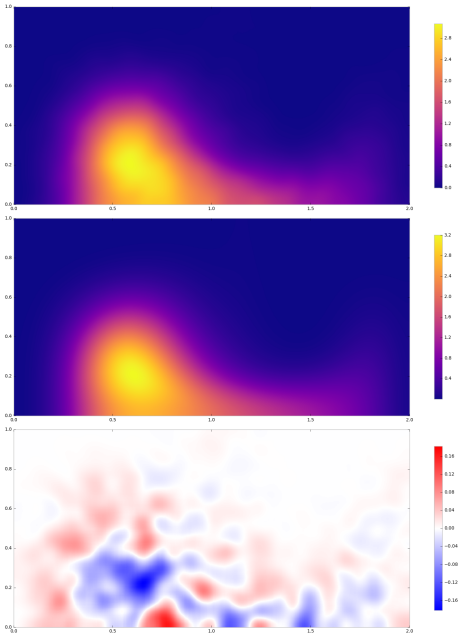
Figure 4.4: The top panel shows the normalized concentration plot made from the particle model using 100000 particles, the mapping of boxcount or truncated Gaussian. The middle panel show the normalized plot of the concentration from the grid model. The bottom panel shows the difference between the two plots. All plots were from timestep 0.



(a) Particle plotted using a boxcount method.

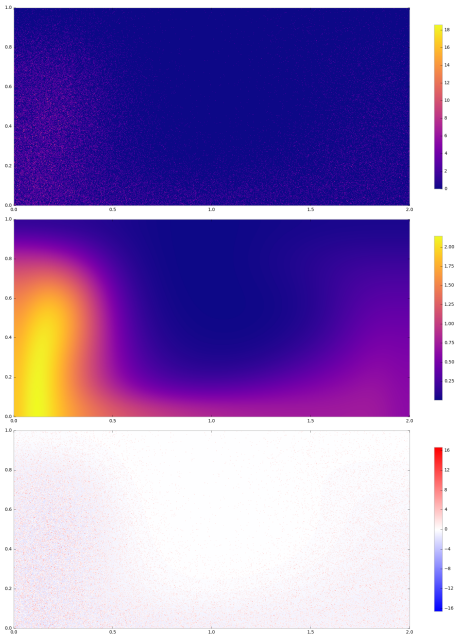


(b) Particles plotted using a Gaussian smoothing with $\sigma = 20$ (see eq (4.1) and eq (4.2)).

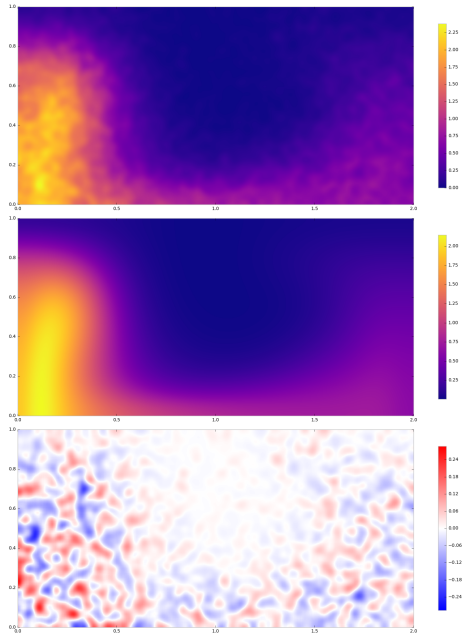


(c) Particles plotted using a Gaussian smoothing with $\sigma = 40$ (see eq (4.1) and eq (4.2)).

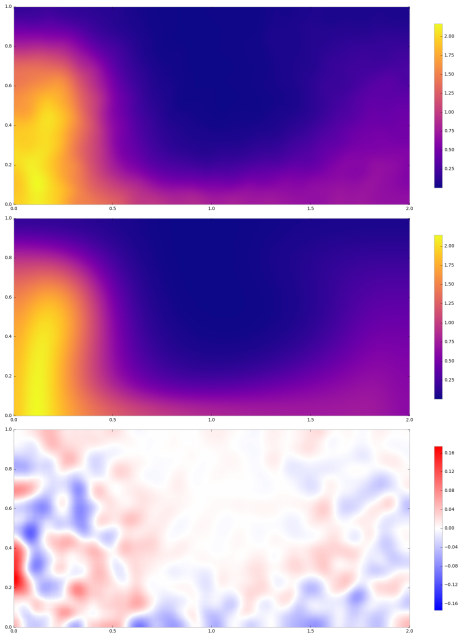
Figure 4.5: Comparison of the two models from timestep 4, where the panels are as in figure 4.4



(a) Particle plotted using a boxcount method.



(b) Particles plotted using a Gaussian smoothing with $\sigma = 20$ (see eq (4.1) and eq (4.2)).



(c) Particles plotted using a Gaussian smoothing with $\sigma = 40$ (see eq (4.1) and eq (4.2)).

Figure 4.6: Comparison of the two models from timestep 8, where the panels are as in figure 4.4.

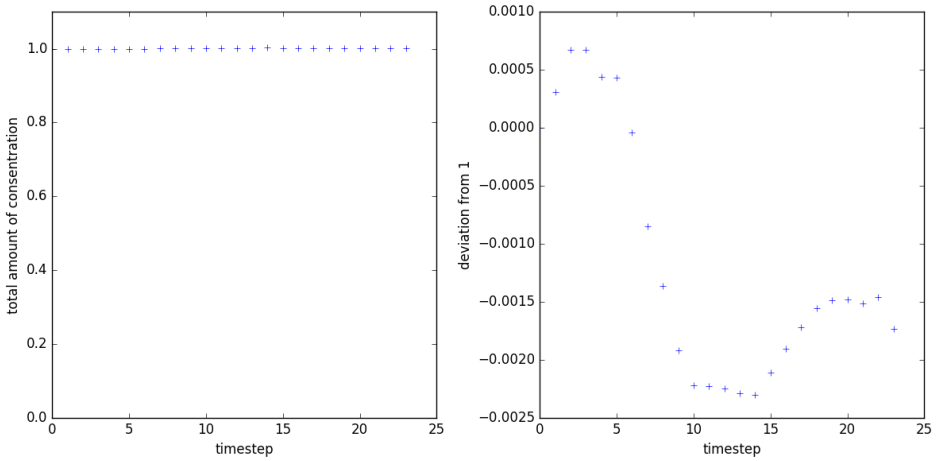


Figure 4.7: Plot over normalized concentration present in the total simulation domain for the double gyre velocity field as a function of time. Low diffusion effect and reflective boundary conditions.

It should be noted that if the boxcount method for mapping the particles to a grid where utilized, the number of particles has to be extremely high, if the resulting plot is compared to the plot made from the grid model. Figure 4.11 and 4.12 shows how the boxcount method for plotting concentration approaches the concentration plot made from the grid method as the number of particles increase. As the number of particles increase the execution time increases accordingly.

One of the reasons for the boxcount method requiring a a large amount of particles, is since the amount of particle, to get a sufficient image, has to be large enough. Large enough means that the standard deviation of the number of particles for a particular bin-cell, imposed by the mapping function, has to be sufficiently smaller than the expected number of particles inside a bin.

Further, the choice of mapping function, or counting method for the particles in the particle model, dictates how many particles needed to get a similar picture as the grid model. As seen in the figure 4.4, 4.5 and 4.6 the number of particles required here are and order of 2^{15} fewer than the number of particles required in 4.11 and 4.12. Thus being much less computational demanding. I.e the mapping of the particles to grid plays a significant part in the required time of the particle model, as the boxcount method will lead to the same results as the grid method, for the limit where the number of particles goes towards infinity, as seen in section 2.2. The grid model when running with 64 ranks uses just under 1000 sec, running the $N_{steps} = 5000$ number of iteration timesteps with a $dt = 2.5 \cdot 10^{-5}$. As a comparison, in figure 4.13, the execution time of the particle method is plotted as function of the number of particles, NP, with $dt = 2.5 \cdot 10^{-3}$, $N_{steps} = 5000$, $p = 64$. Note that the dt in the grid model are 100 times smaller than the dt used in figure 4.13.

The initial idea with the particle model is to model a physical system using a set of assumptions. By using a particle based model the advection part is the relatively simple as

it is solved by numerical integration. The diffusion part of the system is here modeled by random walk. It is worth noting that the "particles" in the simulation does not have to be physical particles. Thus a computational effective method that do not yield any no-physical result is acceptable. Example of models that are based on similar concepts as the particle model but has other inherit abilities are Lattice Boltzmans method, which is highly suitable for numerical fluid dynamics [2] [14], or cylinders particles with increasing radius. Cylinder particles are particles that are cylinders that are defined to have a height and radius. As the position of the cylinders are advected and diffused the radius also increases with time. This increasing radius results in an increased smoothing when calculating the concentration.

Due to the particles being non interacting and the random initial condition the particle method can be run as x different executions of the same program with N particles. This will yield the same collective particle mapping as running one execution of the program with $N \cdot x$ particles.

It could be argued that as the initial conditions or velocity field have never been altered, the difference between the models end behavior has not been examined. However this was not the intended scope of this thesis.

The scheme in the particle model can be viewed as:

$$\boldsymbol{x}_{i+1} = \text{RK-4}(\boldsymbol{x}_i, \boldsymbol{u}(\boldsymbol{x}_i), \delta t) + \boldsymbol{r}_i\sqrt{4D\delta t} \tag{4.3}$$

where RK-4 means the Runge-Kutta 4th order method. From this we see that the accuracy in time for the particle model is of 4th order for the advection part. When it comes to the diffusive part, section 2.2 show that the variance of the diffusive term increases linear with time. Variance in this case, can be thought of as the *diffusive transport length*. A detailed analysis of the accuracy of the diffusion part of the particle method demands the utilization of the theory of stochastic differential equations, as this is highly extensive it is outside the scope of this thesis.

The Runge-Kutta 4th order scheme is a well known scheme and proven workhorse when doing numerical integration. Combined with the fact that there is nothing in the velocity field that changes so fast that one would expect the Runge-Kutta 4th order to not be a sufficient integrator scheme. As the results from the particle method and the grid method look similar and the implementation of the RK-4 is straightforward, it advocates that the RK-4 is well suited for this purpose.

There has been no consideration for which time-steps is reasonable for the particle model, as the focus of this thesis has been on implementing and comparing differences between the two implementations. For the particle model the largest velocity in the velocity field multiplied with the time-step should not be larger than the width of a grid-cell in the resulting mapping. As the largest velocity in the velocity field was found by numerical investigation to be $v_{max} = 0.47$. This yields the largest step, when the $\delta t = 2.5 \cdot 10^{-5}$, as $\delta t \cdot v_{max} = 1.175 \cdot 10^{-5}$. Compared this to the gridsize of $\Delta x = 1/(1024 - 1) = 9.78 \cdot 10^{-4}$, we see that the distance, a particle with the largest possible speed will travel, is a distance shorter than a grid cell. Hence the time-step are within reasonable limits.

For the grid model any time-step and grid size is reasonable as long as the stability criterion of eq 2.1 are upheld. With $\delta t = 2.5 \cdot 10^{-5}$, $D = 0.00619$ and $\Delta x = 1/(512-1) = 1.96 \cdot 10^{-3}, \Delta y = 1/(512-1) = 1.96 \cdot 10^{-3}$ and setting $v_{1i,j}$ and $v_{2i,j}$ in eq (2.1) equal to v_{max}, the stability parameters becomes $r_1 = r_2 = 0.0404$ and $\frac{u_{max}^2}{r_1} = \frac{u_{max}^2}{r_1} = 8.923 \cdot 10^{-4}$. Here the scheme are well within the stability domain and as such could have increased the length of the time-step in order to make the computations faster. As as such for a given gridsize of h in both directions, and with $v_{1i,j} = v_{2i,j} = v_{max}$ the two stability demands can be expressed as i) $\Delta t \leq \frac{h^2}{4 \cdot D}$ and ii) $\frac{v_{max}^2}{D} \leq \Delta t$ meaning that $\delta t \leq 5.04 \cdot 10^{-4}$ and $\delta t \leq 0.028$. Thus the time-step could have been increased by a factor 10, for the system in this thesis, without violating the stability criterion.

4.3.1 A note on gridsize parameters used in timing runs

A note should be made regarding grid-size, the keen-eyed reader would have observed that the grid-size in the previous consideration is not $1/(1024-1)$ which it should be in relation to the number of grid-cells. This is due to simple human error, as the grid-size has to be set independently of the number of grid-cells in the code. The reason for this error inducing choice of implementation is a consequence of having the grid-size variable declared as *static* in the code. Meaning that it is easy available in cache as the variable is accessed often throughout the execution of the program, resulting in a faster execution of the program. As this error in grid-size makes no sense physically it does not imply any error for the execution of the program. This error is only present for the performance tests of the grid-model and in addition as the number of floating point operations are the same. Also the stability criterion would still be upheld if the Δx will be doubled. Thus the speed-test is deemed valid and as an indicator for the scalability for the grid-model.

4.4 Model performance

As seen in figure 4.8 the grid-model experience superlinear scaling. Meaning that the speedup is more than double if the number if ranks are doubled. This is probably explained by better use of cache. As more cores allow for a larger portion of the total problem to be kept in the lower parts of cache, and therefore are closer to the CPU. Hence more of the problem fit into cache and therefore the speedup is better than linear. As a result the efficiency is higher than 1.0 some places. Note that the highest amount of efficiency is seen at $p = 2^7$, as such the CPU-time is at its lowest.

The particle model has a more linear scaling as seen in figure 4.9. This should be expected as the implementation of this model emphasize low of communication between the ranks as communication occur only as a need for returning data and the calculations in it self are independent of parallelization.

A performance plot for the particle model with a larger timestep can be seen in figure 4.10. Here the serial part of the program is larger as the speedup starts flattening out for large

number of ranks can be seen more clearly.

It should be noted that the reason for stopping the speedup tests at $p = 512$ for the particle model with the large stepsize, was since the simulations suddenly took a much larger wall-time while not producing any output, thus probably indicating that the program is hanging. Node crashes were also experienced. The reasons for this are not fully understood, as if the speedup threshold have been reached the wall-time should not increase, it should merely be the same. Accordingly maybe the time spent on communication has increased to such lengths that the resulting total execution time take a significant longer time. For some reason when using the Cartesian communicator it requires a certain amount of grid-cells for not stalling the program. One would be led to believe that when not using the Cartesian communicator this should not be an issue. However this seems not to be the case. It turns out that Vilje has two implementations of the MPI-library one is openMPI, the other is "mpt" which is SGI Message Passage Toolkit [1]. Mpt tends to lead to a slower performance of the program than openMPI, but does not crash when number of ranks increase to over 1024. Since all of the other performance tests was done using openMPI and performance trend is clearly visible from figures 4.8, 4.9 and 4.10, the performance tests was not redone while utilizing the mpt library.

4.5 Inquiries relating performance testing

4.5.1 Gridded model

Given that the system sizes can be written as $i_{max} = 2^m$ and $j_{max} = 2^n$ were $m, n \in \mathbb{N}$, some defined parameters are worthy of examination. Since the total number of processors can be written as $p = 2^k$, the total number of grid-cells pr core in the grid-model is

$$F(m,k) = \frac{2^m \cdot 2^n}{p} = 2^{m+n-k} = 2^{2m-k-1} \tag{4.4}$$

since by definition $n = m - 1$ (as $x \in [0,2], y \in [0,1]$). Given the Cartesian MPI communicator used in the code for the grid-model, the number of processors can be expressed as $p = 2^h \cdot 2^b$, the number of processors in receptively height and width of the Cartesian structure.

Due to the boundary conditions of the system being non-periodic, there exist a number of grid-cells that will not be sent. The reason for this is that there exist no neighbour-processor in the direction of a boundary for the processor in question. Hence the total number of grid-cells that is not sent is $2 \cdot 2^m + 2 \cdot 2^n = 3 \cdot 2^m$.

Further the total number of grid-cells to communicate is:

$$\begin{aligned} C(k(h,b),m) &= p \cdot \left(2 \cdot \frac{2^m}{2^b} + 2 \cdot \frac{2^n}{2^h}\right) - 3 \cdot 2^m \\ &= 2^m \cdot (2^{h+1} + 2^b - 3) \end{aligned} \tag{4.5}$$

[1] https://support1-sgi.custhelp.com/app/home

Thus the number of grid-cells to communicate per core is:

$$CP(m, k(h, b)) = \frac{C(h, b, m)}{p} = \frac{2^{m+1+h} + 2^{m+b} + 3 \cdot 2^m}{2^h \cdot 2^b} = 2^{m+1-b} + 2^{m-h} - 3 \cdot 2^{m-h-b} \tag{4.6}$$

Then it is reasonable to define the "area vs circumference" of a processor. That is how large is the domain of the processor compared to halo (which has to be communicated). Thus:

$$\begin{aligned} L(m, h, b) &= \frac{F(m, k(h, b))}{CP(m, k(h, b))} = \frac{2^{2m-k-1}}{\frac{2^{m+1+h}+2^{m+b}-3\cdot 2^m}{2^h \cdot 2^b}} \\ &= \frac{2^{m-1}}{2^{h+1} + 2^b - 3} \end{aligned} \tag{4.7}$$

4.5.2 The guided random walk model

When the total number of particles is 2^N, and the total number of cores are 2^k. The number of particles for each core is:

$$P_c(N, k) = \frac{2^N}{2^k} \tag{4.8}$$

Thus the total number of send operations are 1 with 2^{N-k} elements. Number of particles per total area of the grid model are thus:

$$P_a(N, m) = \frac{2^N}{2^{2m-1}} \tag{4.9}$$

Meaning that when using the boxcount method for mapping the particles to a grid the number of particles particles has to be comparable to or higher than the number of bins in the resulting picture. N_p must be much larger than the number of cells in the area with interesting concentration for the boxcount method to work. Such that random fluctuations originating from the random walk does not influence the resulting concentration plot to such an extend that it differs from the plot generated by the grid-model.

The plots of L and $P_C c$ in figure 4.14a and 4.14b should be correlated to speedup for not to large amount of ranks. As the L and P_c is a indication of the amount of work done on calculation vs the work done on sending.

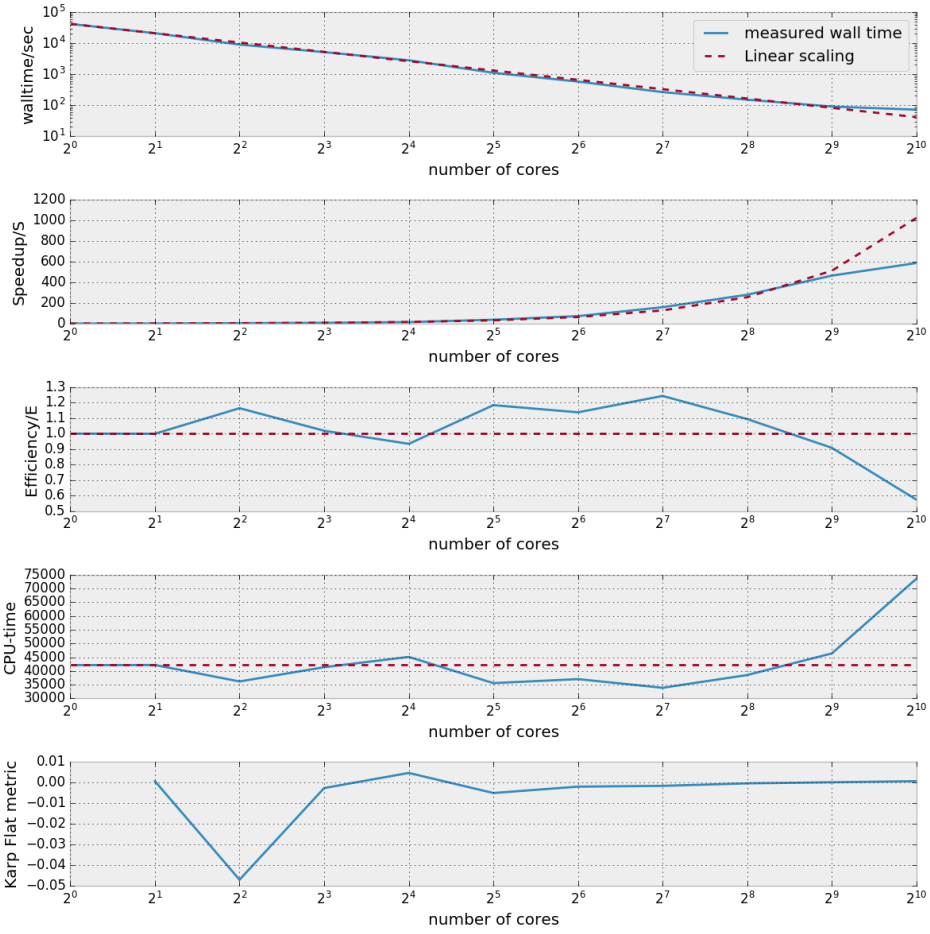


Figure 4.8: Plott of wall-time, speedup, efficiency, total CPU time and Karp-Flat metric for the grid model with system size (Nx,Ny)=(2048,1024) and $N_{steps} = 500000$

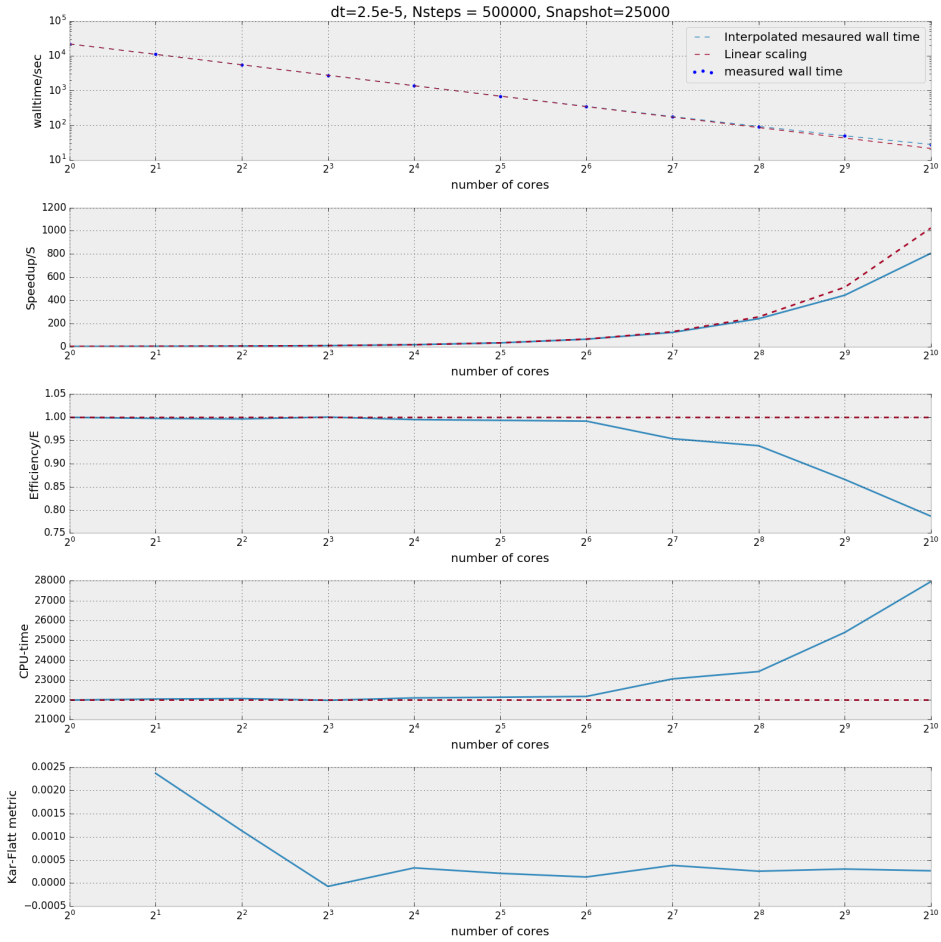


Figure 4.9: Plott of wall-time, speedup, efficiency, total CPU time and Karp-Flat metric for the particle model with 2^{17} number of particles and $N_{steps} = 500000$, with $\delta t = 2.5 \cdot 10^{-5}$

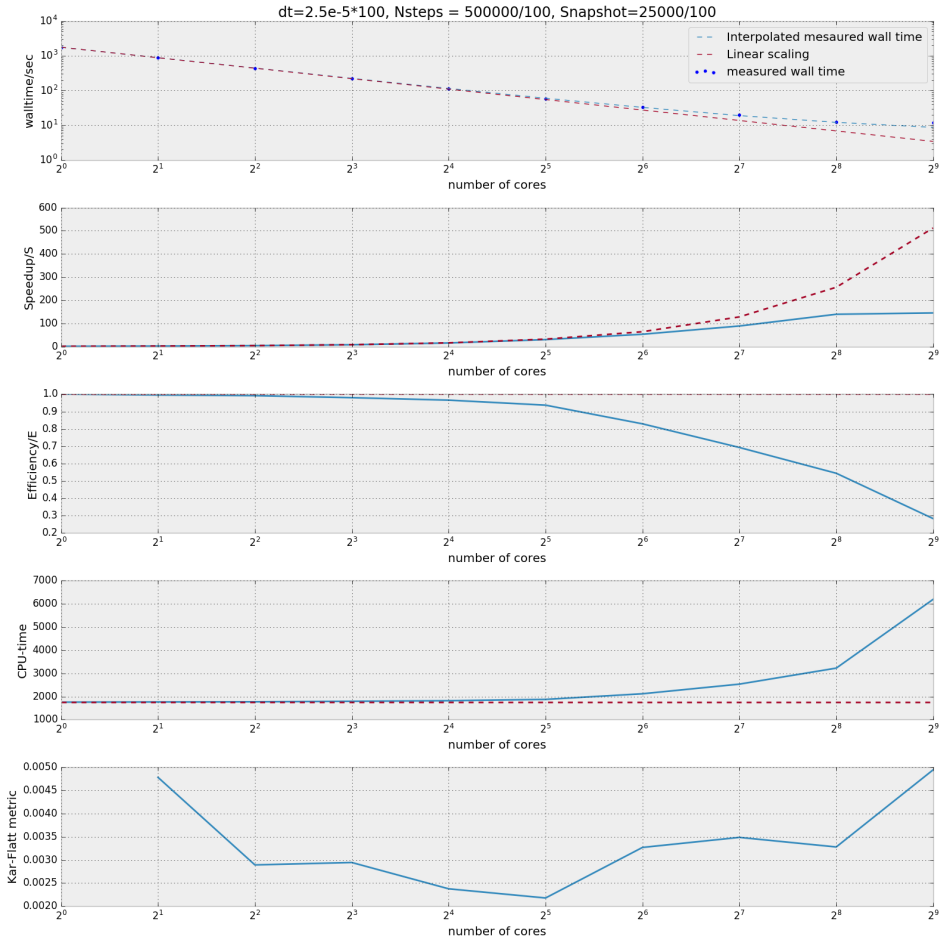


Figure 4.10: Plott of wall-time, speedup, efficiency, total CPU time and Karp-Flat metric for the particle model with 2^{20} number of particles and $N_{steps} = 5000$, with $\delta t = 2.5 \cdot 10^{-3}$

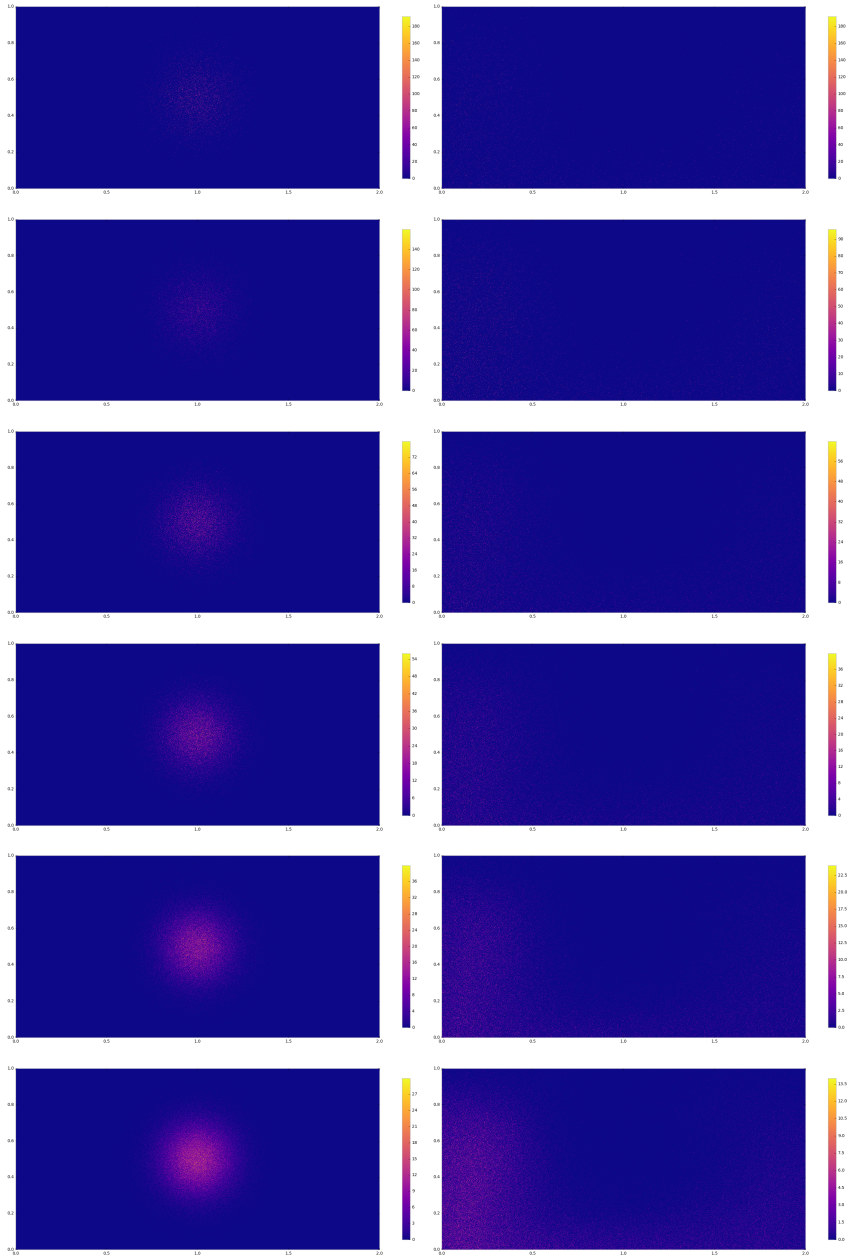


Figure 4.11: Shows how the pictures generated using the boxcount method of the particle implementation, goes towards the plots from the grid method seen in figure 4.1.The pictures in each line the left are of output timestep 0 and the right picture are of output picture 8. To generate the plots in the top line 2^{20} particles are used. For each line the number of particles are doubled. To generate the plots in the bottom line 2^{25} particles are used. Here the gridspacing $h = 2/1023 = 1.955 \cdot 10^{-3}$

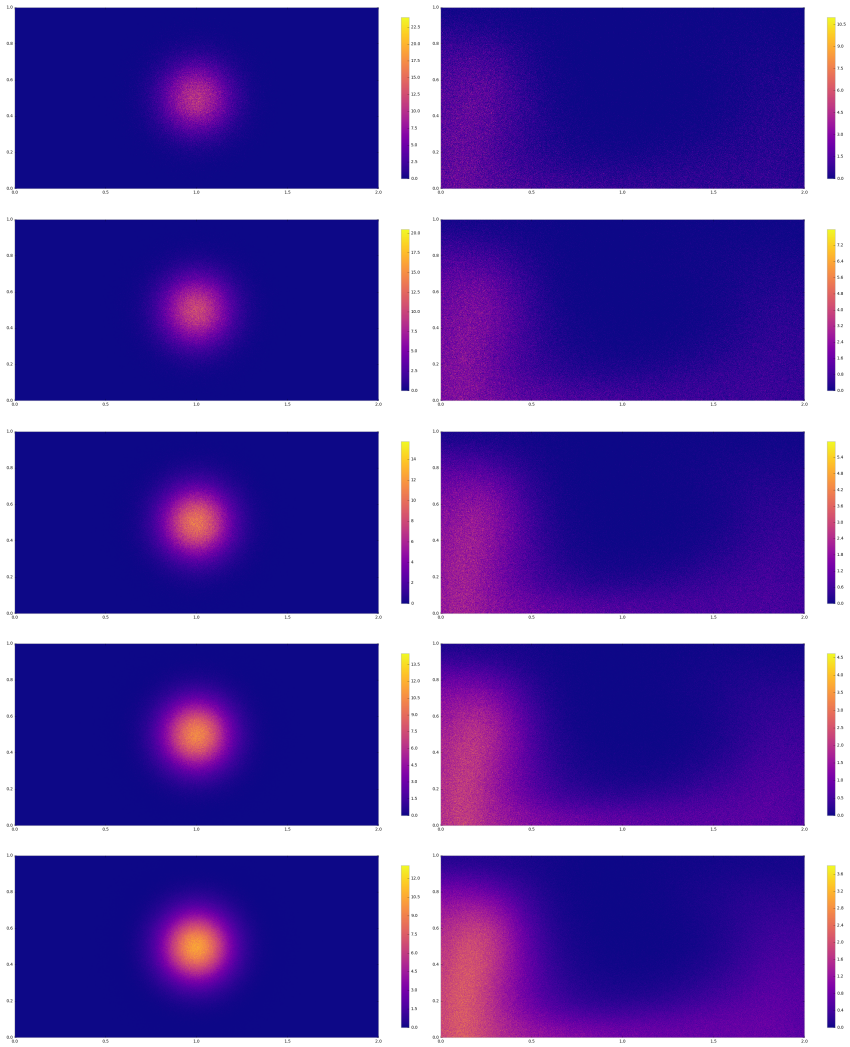


Figure 4.12: Shows how the pictures generated using the boxcount method of the particle implementation, goes towards the plots from the grid method seen in figure 4.1.The pictures in each line the left are of output timestep 0 and the right picture are of output timestep 8. To generate the plots in the top line 2^{26} particles are used. For each line the number of particles are doubled. To generate the plots in the bottom line 2^{30} particles are used. Here the gridspacing $h = 2/1023 = 1.955 \cdot 10^{-3}$

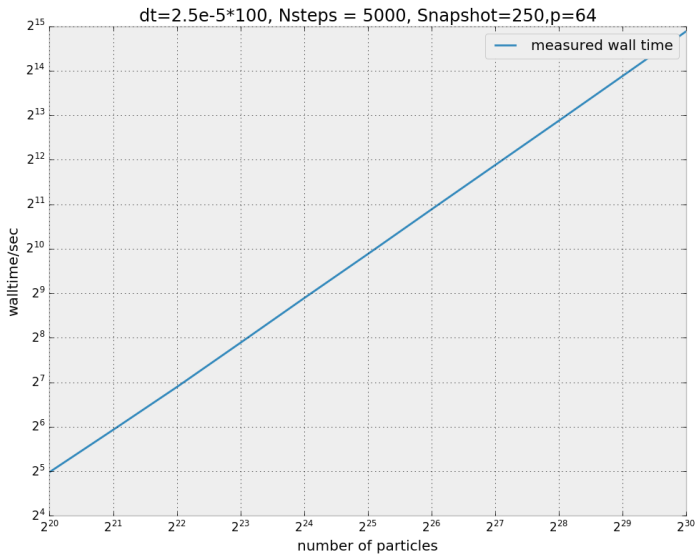
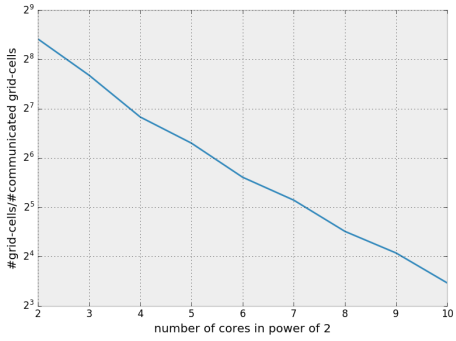
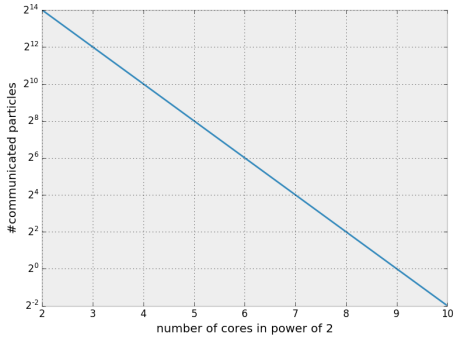


Figure 4.13: Shows how the computational time of the particle implementation scales as a function of the number of particles, when using number of cores $p = 64$.



(a) Plot of equation (4.7), with, $m = 11, n = 10$ as a function of k, where h is defined as $b - 1$ if k is not an equal number.



(b) Plot of equation (4.8) as it is the total number of particles each rank has to send. Here $N = 17$

Figure 4.14: A visualisation of total amount of sending operations versus computational operations.

Chapter 5

Conclusion

It was shown that both method scales rather well for increasing number of processors. The computational time for the particle method is highly dependent of the number of particles. As the mapping of particles to grid highly influences the number of required particles to make a similar concentration plot as the grid model. Using a ”boxcount” method for mapping the particles to grid demands such a large number of particles that the particle model will be outperformed by the grid model. The grid model exhibit super linear scaling, due to better usage of cache. Implementations of both models are highly affected by the implementation of MPI on the test facilities. Both models produce very similar results, up to numerical resolution. Note that ”numerical resolution” is related to number of particles, and that a certain amount of random noise is unavoidable with this formulation. For the box count method, the amount of noise is also related to the grid resolution used when ”box counting” the particles. From what we have seen, it appears the box count method is unsuitable for applications where concentration must be known, due to the extremely large number of particles required to get smooth result. As seen in figure 5.1 even for 2^{30} (1073741824) particles, the box-count method yields a visibly noisy result when projected onto a grid of 2048×1024 cells. Considering that this gives an average of 512 particles per cell (in practice more, since the concentration is essentially zero in parts of the grid), this is perhaps a somewhat surprising result.

5.1 Sugestions for further work or inquiries

Further work could relate to examine if the results from this thesis is equivalent if the dimensionality of the problem is increased. Does there exist a point were one method are highly favorable over the other as a result of the dimensionality? The equation could also be expanded from advection-diffusion equation to advection-diffusion-reaction equation.

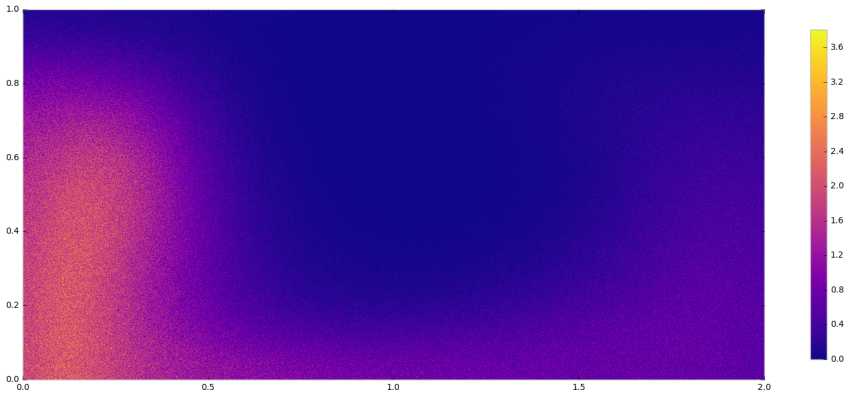


Figure 5.1: Shows how the picture made from the particle implementation of timestep 8 generated using the boxcount method. Here 2^{30} particles are used and the gridspacing is $h = 2/1023 = 1.955 \cdot 10^{-3}$

Bibliography

[1] Strategy analytics: Multi-core processors dominate smartphones in q3 2011. *Entertainment Close - Up*, Dec 22 2011. Name - Texas Instruments Inc; Copyright - Copyright 2011 Close-Up Media, Inc. All Rights Reserved; Last updated - 2015-04-12.

[2] Davide Alemani, Bastien Chopard, Josep Galceran, and Jacques Buffle. Lbgk method coupled to time splitting technique for solving reaction-diffusion processes in complex systems. *Physical chemistry chemical physics : PCCP*, 7(18), September 2005.

[3] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities, reprinted from the afips conference proceedings, vol. 30. *IEEE Solid-State Circuits Society Newsletter*, 12(3):19–20, Summer 2007.

[4] Dale A Anderson. *Computational fluid mechanics and heat transfer*. Series in computational methods in mechanics and thermal sciences. Hemisphere Publ, Washington, 1984.

[5] Arne Sundstrm Bertil Gustafsson, Heinz Otto Kreiss.

[6] Gene Cooperman and Victor Grinberg. Scalable parallel coset enumeration: Bulk definition and the memory wall. *Journal of Symbolic Computation*, 33(5):563 – 585, 2002.

[7] NTNU HPC GROUP. Vilje hpc wiki, 2017.

[8] John L. Gustafson. Reevaluating amdahl's law. *Commun. ACM*, 31(5):532–533, May 1988.

[9] P. M. ; Griffiths D. F. Hindmarsh, A. C. ; Gresho. The stability of explicit euler time-integration for certain finite difference approximations of the multi-dimensional advectiondiffusion equation. *International Journal for Numerical Methods in Fluids, September 1984, Vol.4(9), pp.853-897*, 1984.

[10] Alan H. Karp and Horace P. Flatt. Measuring parallel processor performance. *Commun. ACM*, 33(5), may 1990.

[11] A. Leva, F. Terraneo, I. Giacomello, and W. Fornaciari. Event-based power/performance-aware thermal management for high-density microprocessors. *IEEE Transactions on Control Systems Technology*, PP(99):1–16, 2017.

[12] Peter S. Pacheco. *An Introduction to Parallel Programming*. Morgan Kaufmann, 2011.

[13] Shawn C. Shadden, Francois Lekien, and Jerrold E. Marsden. Definition and properties of lagrangian coherent structures from finite-time lyapunov exponents in two-dimensional aperiodicflows. *Physica D: Nonlinear Phenomena*, 212(3):271 – 304, 2005.

[14] Feifei Song, Wei Wang, and Jinghai Li. A lattice boltzmann method for particle-fluid two-phase flow. *Chemical Engineering Science*, 102:442 – 450, 2013.

[15] John C. Strikwerda. Finite difference schemes and partial differential equations. *Mathematics of Computation*, 55(192):869–870, 1990.

[16] Arthur Suszko and Mohamed S. El-Genk. Thermally anisotropic composite heat spreaders for enhanced thermal management of high-performance microprocessors. *International Journal of Thermal Sciences*, 100:213 – 228, 2016.

[17] H. Tennekes and J. L. Lumley. *A first course in turbulence*. MIT Press, 1972.

[18] Thomas Janke Petter Rnningen Tor Nordam, Raymon Nepstad. Parallel particle transport, 2016.

[19] Andre W. Visser. Lagrangian modelling of plankton motion: From deceptively simple random walks to fokkerplanck and back again. *Journal of Marine Systems*, 70(34):287 – 299, 2008. Impact of Small-scale Physics on Marine BiologySelected papers from the 2nd Warnemnde Turbulence Days2nd Warnemnde Turbulence Days.

[20] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: Implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, March 1995.

Appendices

Appendix A

Source code

Here the source code for the models presented in section 3.2 and 3.3. First the code for the gridded model will be presented and after the code for the particle model will be presented.

Listing A.1: Grid model implementation

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <math.h>

#include <mpi.h>

/* Prototypes */
void ftcs_solver ( int step );
void border_exchange ( int step );
void gather_temp( int step );
void scatter_temp();
void scatter_material();
void scatter_velocity();
void commit_vector_types ();
void external_heat ( int step );
void write_temp ( int step );
void print_local_temps(int step);
void init_temp_material();
void init_local_temp();
//void init_velocity_material();
void init_local_velocity_material(double time);
void enforce_reflective_boundaries(int step);
double double_gyre_x_direc( double x, double y, double t, double A, double
    e, double w);
double double_gyre_y_direc( double x, double y, double t, double A, double
    e, double w);
void conservation(int step);
void generate_position_file(int number);
```

```
void write_conservation_to_file(char* filename);
/*velocity field properties*/
const float AMPLITUDE = 0.1;//0.1;
const float OMEGA     = 1.0;
const float EPSILON   = 0.25;
/*material diffusive parameter*/
const float MERCURY   = 0.00619;

const float two       = 2.0;
const int   one       = 1.0;
#define  M_PI acos(-1.0)

/* Size of the computational grid */
const int GRID_SIZE[2] ={2048,1024};

/* Parameters of the simulation: how many steps */
const int NSTEPS = 500000;


/* How often to dump state to file (steps).  */
const int SNAPSHOT = 25000;

/* Border thickness */
const int BORDER = 1;

/* Arrays for the simulation data */
float
    *material,            // Global material constants, on rank 0
    *temperature,         // Global temperature field, on rank 0
    *velocity[2],         // global velocity field in x and y direction,
        on rank 0
    *local_material,      // Local part of the material constants
    *local_temp[2],       // Local part of the temperature (2 buffers)
    *local_velocity[2],   // Local part of the velocity field
    *total_conservation;  // Array containing the total amount of
        consentraton/heat

/* Discretization: 5cm square cells, 2.5ms time intervals */
const float
    h  = 1.0/(512-1),
    dt = 2.5e-5;

/* Local state */
int
    size, rank,                     // World size, my rank
    dims[2],                        // Size of the cartesian
    periods[2] = { false, false },  // Periodicity of the cartesian
    coords[2],                      // My coordinates in the cartesian
    north, south, east, west,       // Neighbors in the cartesian
    local_grid_size[2],             // Size of local subdomain
    local_origin[2];                // World coordinates of (0,0) local

// Cartesian communicator
MPI_Comm cart;


// MPI datatypes for gather/scater/border exchange
```

```
MPI_Datatype
    border_row, border_col, grid_block, receive_block,
        receive_block_material;

/* Indexing functions, returns linear index for x and y coordinates,
    compensating for the border */

// temperature
int ti(int x, int y){
    return y*GRID_SIZE[0] + x;
}

// material
int mi(int x, int y){
    return ((y+(BORDER-1))*(GRID_SIZE[0]+2*(BORDER-1)) + x + (BORDER-1));
}

// local_material
int lmi(int x, int y){
    return ((y+(BORDER-1))*(local_grid_size[0]+2*(BORDER-1)) + x + (BORDER
        -1));
}

// local_temp
int lti(int x, int y){/* this is with the border */
      return ((y+BORDER)*(local_grid_size[0]+2*BORDER) + x + BORDER);
}

int inside(int x, int y){//this checks if you are inside local area
    return x >= local_origin[0] &&
    x < local_origin[0] + local_grid_size[0] &&
    y >= local_origin[1] &&
    y < local_origin[1] + local_grid_size[1];
}



void ftcs_solver( int step ){//#
     for(int i = 0; i < local_grid_size[0]; i++){

      for(int j = 0; j < local_grid_size[1]; j++){
          local_temp[(step+1)%2][lti(i,j)] = local_temp[step%2][lti(i,j)]
              + local_material[lmi(i,j)]*( local_temp[step%2][lti(i+1,j)]
              + local_temp[step%2][lti(i-1,j)] + local_temp[step%2][lti(i,
              j+1)] + local_temp[step%2][lti(i,j-1)] - 4.0*local_temp[step
              %2][lti(i,j)]) + local_velocity[0][lmi(i,j)]*( local_temp[
              step%2][lti(i+1,j)] -  local_temp[step%2][lti(i-1,j)]) +
              local_velocity[1][lmi(i,j)]*( local_temp[step%2][lti(i,j+1)]
               -  local_temp[step%2][lti(i,j-1)]);
          //implementing stability crit
          if  (local_material[lmi(i,j)]>=0.25){
              perror("Stability criteria violated! Diffusion parameter to
                  large\n");
          }
          if (local_material[lmi(i,j)]<0.0){
```

```
                perror("Stability criteria violated! Diffusion parameter to
                    small\n");
            }
            if (( local_velocity[0][lmi(i,j)]*local_velocity[0][lmi(i,j)]/
                local_material[lmi(i,j)] +  local_velocity[0][lmi(i,j)]*
                local_velocity[0][lmi(i,j)]/local_material[lmi(i,j)]) >2.0 )
                {
                perror("Stability criteria violated! Advection parameter to
                    large\n");
            }

    }
  }
}


void commit_vector_types ( void ){//#
    /* TODO: Create and commit the types for the border exchange and
        collecting the subdomains */
    MPI_Type_vector( local_grid_size[1], 1, local_grid_size[0]+2*BORDER,
        MPI_FLOAT, &border_col );
    MPI_Type_commit( &border_col);


    MPI_Type_vector( local_grid_size[0], 1, 1, MPI_FLOAT, &border_row );
    MPI_Type_commit( &border_row);

    MPI_Type_vector( local_grid_size[1], local_grid_size[0], GRID_SIZE[0],
         MPI_FLOAT, &grid_block);
    MPI_Type_commit( &grid_block);

    MPI_Type_vector( local_grid_size[1], local_grid_size[0],
        local_grid_size[0]+2*BORDER, MPI_FLOAT, &receive_block);
    MPI_Type_commit( &receive_block);

    MPI_Type_vector( local_grid_size[1], local_grid_size[0],
        local_grid_size[0], MPI_FLOAT, &receive_block_material);
    MPI_Type_commit(&receive_block_material);
}


void border_exchange ( int step ){//#
    /* TODO: Implement the border exchange */

    float* newest_local_temp = local_temp[(step)%2];

    MPI_Sendrecv(&newest_local_temp[ lti(0,0)], 1, border_col, west, 1, &
        newest_local_temp[lti(local_grid_size[0],0)], 1, border_col, east,
         1, cart, MPI_STATUS_IGNORE);

    MPI_Sendrecv(&newest_local_temp[ lti(local_grid_size[0]-1,0)], 1,
        border_col, east, 1, &newest_local_temp[lti(-BORDER,0)], 1,
        border_col, west, 1, cart, MPI_STATUS_IGNORE);

    MPI_Sendrecv(&newest_local_temp[lti(0,0)],1,border_row, north, 1, &
        newest_local_temp[lti(0,local_grid_size[1])], 1, border_row, south
        , 1, cart, MPI_STATUS_IGNORE);
```

```
    MPI_Sendrecv(&newest_local_temp[lti(0,local_grid_size[1]-1)],1,
        border_row, south, 1, &newest_local_temp[lti(0,-BORDER)], 1,
        border_row, north, 1, cart, MPI_STATUS_IGNORE);

}


void gather_temp( int step){//#
    /* TODO: Collect all the local subdomains in the temperature array at
        rank 0 */

    MPI_Barrier(cart);
    if(rank!=0){

      MPI_Send(&local_temp[0][lti(0,0)],1,receive_block,0,3,MPI_COMM_WORLD
          );
    }
    if (rank==0){
      for (int i=1; i<size; i++){
        MPI_Cart_coords( cart, i, 2, coords);

        MPI_Recv(&temperature[ti(coords[0]*local_grid_size[0],coords[1]*
            local_grid_size[1])],1,grid_block,i,3,MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);
    }

        MPI_Cart_coords( cart, 0, 2, coords);
        MPI_Sendrecv(&local_temp[0][lti(0,0)],1,receive_block,0,1,&
            temperature[ti(coords[0]*local_grid_size[0],coords[1]*
            local_grid_size[1])],1,grid_block,0,1,cart,MPI_STATUS_IGNORE);
   }
    MPI_Barrier(cart);
}


void scatter_temp(){//#
    /* TODO: Distribute the temperature array at rank 0 to all other ranks
         */
    MPI_Barrier(cart);
    if (rank==0){
        MPI_Cart_coords( cart, 0, 2, coords);
        MPI_Sendrecv(&temperature[ti(coords[0]*local_grid_size[0],coords
            [1]*local_grid_size[1])],1,grid_block,0,1,&local_temp[0][lti
            (0,0)],1,receive_block,0,1,cart,MPI_STATUS_IGNORE);

      for (int i=1; i<size; i++){
        MPI_Cart_coords( cart, i, 2, coords);
        MPI_Send(&temperature[ti(coords[0]*local_grid_size[0],coords[1]*
            local_grid_size[1])],1,grid_block,i,2,MPI_COMM_WORLD);
      }
   }
    if (rank!=0){
      MPI_Recv(&local_temp[0][lti(0,0)],1,receive_block,0,2,MPI_COMM_WORLD
          ,MPI_STATUS_IGNORE);
    }
    MPI_Barrier(cart);
```

```
}

void scatter_velocity(){//this could be deleted
/*have a deadlok here, bad code and not used anyways*/
    if(rank==0){
      for (int i=0; i<size; i++){
        MPI_Cart_coords(cart, i, 2, coords);

        MPI_Send(&velocity[0][mi(coords[0]*local_grid_size[0],coords[1]*
            local_grid_size[1])],1,grid_block,i,1,MPI_COMM_WORLD); //
            sending x-velocities
        MPI_Send(&velocity[1][mi(coords[0]*local_grid_size[0],coords[1]*
            local_grid_size[1])],1,grid_block,i,2,MPI_COMM_WORLD); //
            sending y-velocities
    }
  }
  MPI_Recv(&local_velocity[0][lmi(0,0)],1,receive_block_material,0,1,
      MPI_COMM_WORLD,MPI_STATUS_IGNORE);
  MPI_Recv(&local_velocity[1][lmi(0,0)],1,receive_block_material,0,2,
      MPI_COMM_WORLD,MPI_STATUS_IGNORE);
}



void scatter_material(){//#
        //this function is only valid for border=1, since border-1=0
        /* Distribute the material array at rank 0 to all other ranks */
  MPI_Barrier(cart);
    if (rank==0){
        MPI_Cart_coords( cart, 0, 2, coords);
        MPI_Sendrecv(&material[mi(coords[0]*local_grid_size[0],coords[1]*
            local_grid_size[1])],1,grid_block,0,1,&local_material[lmi(0,0)
            ],1,receive_block_material,0,1,cart,MPI_STATUS_IGNORE);
      for (int i=1; i<size; i++){
        MPI_Cart_coords( cart, i, 2, coords);
        MPI_Send(&material[mi(coords[0]*local_grid_size[0],coords[1]*
            local_grid_size[1])],1,grid_block,i,4,MPI_COMM_WORLD);
    }
  }
  if (rank!=0){
    MPI_Recv(&local_material[lmi(0,0)],1,receive_block_material,0,4,
        MPI_COMM_WORLD,MPI_STATUS_IGNORE);
  }
  MPI_Barrier(cart);
}



int main ( int argc, char **argv ){
    MPI_Init ( &argc, &argv );
    MPI_Comm_size ( MPI_COMM_WORLD, &size );
    MPI_Comm_rank ( MPI_COMM_WORLD, &rank );

    MPI_Dims_create( size, 2, dims );
```

```
    MPI_Cart_create( MPI_COMM_WORLD, 2, dims, periods, 0, &cart );
    MPI_Cart_coords( cart, rank, 2, coords );

    MPI_Cart_shift( cart, 1, 1, &north, &south );
    MPI_Cart_shift( cart, 0, 1, &west, &east );

    local_grid_size[0] = GRID_SIZE[0] / dims[0];
    local_grid_size[1] = GRID_SIZE[1] / dims[1];
    local_origin[0] = coords[0]*local_grid_size[0];
    local_origin[1] = coords[1]*local_grid_size[1];

    commit_vector_types ();

    if(rank == 0){
        size_t temperature_size = GRID_SIZE[0]*GRID_SIZE[1];
        temperature = calloc(temperature_size, sizeof(float));
        size_t material_size    = (GRID_SIZE[0]+2*(BORDER-1))*(GRID_SIZE
            [1]+2*(BORDER-1));
        material    = calloc(material_size, sizeof(float));
        velocity[0] = calloc(material_size, sizeof(float));
        velocity[1] = calloc(material_size, sizeof(float));
        total_conservation = calloc(NSTEPS/SNAPSHOT,sizeof(float));
        init_temp_material();


      }
    size_t lsize_borders = (local_grid_size[0]+2*BORDER)*(local_grid_size
        [1]+2*BORDER);
    size_t lsize = (local_grid_size[0]+2*(BORDER-1))*(local_grid_size
        [1]+2*(BORDER-1));//why is BORDER-1? THIS WILL MAKE ZERO
    local_material = calloc( lsize , sizeof(float) );
    local_temp[0] = calloc( lsize_borders , sizeof(float) );
    local_temp[1] = calloc( lsize_borders , sizeof(float) );
    local_velocity[0] = calloc( lsize ,sizeof(float));
    local_velocity[1] = calloc( lsize, sizeof(float));

    init_local_temp();



    scatter_material();

    scatter_temp();

    for( int step=0; step<NSTEPS; step += 1 ){
        border_exchange( step );

        init_local_velocity_material((double) dt* step);
        enforce_reflective_boundaries(step);

        ftcs_solver( step );

        if((step % SNAPSHOT) == 0){
            gather_temp ( step );
            if(rank == 0){
                write_temp(step);
                //conservation(step);
```

```
                generate_position_file(step);
            }
        }
    }
    if(rank == 0){
        //write_conservation_to_file("
            concentration_dump_low_not_reflective_bc.txt");
        free (temperature);
        free (material);
        free (velocity[0]);
        free (velocity[1]);
        free (total_conservation);
    }

    free(local_material);
    free(local_temp[0]);
    free (local_temp[1]);
    free (local_velocity[0]);
    free (local_velocity[1]);

    MPI_Finalize();
    exit ( EXIT_SUCCESS );
}

void write_conservation_to_file(char* filename){

  FILE *f = fopen(filename, "w");
  if (f == NULL)
  {
    printf("Error opening file!\n");
    exit(1);
  }
  /* rather unsure of this fprintf  */
  for(int i=0; i<NSTEPS/SNAPSHOT; i++){
    fprintf(f,"%f\n",total_conservation[i]);
  }
  fclose(f);
}

void conservation(int step){
      total_conservation[step/SNAPSHOT] = 0.0;
      for (int i=0; i<GRID_SIZE[0]*GRID_SIZE[1]; i++){
          total_conservation[step/SNAPSHOT] += temperature[i];
      }
}









void generate_position_file(int number){//#
        char filename[28];
```

```
        sprintf (filename, "data/concentration%.4d.bin", number/SNAPSHOT )
            ;
        FILE *f = fopen(filename,"w");
        if (f == NULL)
        {
            printf("Error opening file!\n");
            exit(1);
          }
                fwrite( temperature,4,GRID_SIZE[0]*GRID_SIZE[1],f); /*
                    since the temperature array are of type float*/

        fclose(f);
}



void enforce_reflective_boundaries(int step){//#

    if (east==-2){
      //west bc
      for(int i = 0; i < local_grid_size[1]; i++ ){//this only works for
          border=1!
          local_temp[step%2][lti(local_grid_size[0],i)] = local_temp[step
              %2][lti(local_grid_size[0]-1,i)];
      }
    }

    if (west==-2){
      //east bc
      for(int i = 0; i < local_grid_size[1]; i++ ){
      local_temp[step%2][lti(-1,i)] = local_temp[step%2][lti(0,i)];
      }
    }

    if (north==-2){
      //west bc
      for(int i = 0; i < local_grid_size[0]; i++ ){
        local_temp[step%2][lti(i,-1)] = local_temp[step%2][lti(i,0)];
      }
    }
    if (south==-2){
      //east bc
      for(int i = 0; i < local_grid_size[0]; i++ ){
        local_temp[step%2][lti(i,local_grid_size[1])] = local_temp[step
            %2][lti(i,local_grid_size[1]-1)];
      }
    }
}



void init_local_temp(void){//#

    for(int x=- BORDER; x<local_grid_size[0] + BORDER; x++ ){
        for(int y= - BORDER; y<local_grid_size[1] + BORDER; y++ ){
            local_temp[1][lti(x,y)] = 0.0;
```

```
            local_temp[0][lti(x,y)] = 0.0;
        }
    }
}

double double_gyre_x_direc( double x, double y, double t, double A, double
     e, double w){
    double a = e * sin(w*t);
    double b = 1.0 - 2.0*e*sin(w*t);
    double f = a*x*x + b*x;

    return -1.0*M_PI*A*sin(M_PI*f) * cos(M_PI*y);              // x
        component of velocity


}

double double_gyre_y_direc( double x, double y, double t, double A, double
     e, double w){
    double a = e * sin(w*t);
    double b = 1.0 - 2.0*e*sin(w*t);
    double f = a*x*x + b*x;

    return M_PI*A*cos(M_PI*f) * sin(M_PI*y) * (2.0*a*x + b); // y
        component of velocity

}

void init_local_velocity_material(double time){
    for( int x = 0; x < local_grid_size[0]; x++){
        for(int y = 0; y < local_grid_size[1]; y++){

          local_velocity[0][lmi(x,y)] =
                  -(dt/(2.0*h))*double_gyre_x_direc((double) 2.0*(
                      local_origin[0] + x)/(GRID_SIZE[0]-1), (double)(
                      local_origin[1]+ y)/(GRID_SIZE[1]-1), time,
                      AMPLITUDE, EPSILON, OMEGA);

          local_velocity[1][lmi(x,y)] =
                  -(dt/(2.0*h))*double_gyre_y_direc((double) 2.0*(
                      local_origin[0] + x)/(GRID_SIZE[0]-1), (double)(
                      local_origin[1]+ y)/(GRID_SIZE[1]-1), time,
                      AMPLITUDE, EPSILON, OMEGA);
        }
      }



}




void init_temp_material(){//#
    double stn_dev_x = GRID_SIZE[0]/16.0;
    double stn_dev_y = GRID_SIZE[1]/8.0;
```

```
    double x_0 = GRID_SIZE[0]/2.0;
    double y_0 = GRID_SIZE[1]/2.0;
    double A = 100.0;


    for(int x = 0; x < GRID_SIZE[0]; x++){
        for(int y = 0; y < GRID_SIZE[1]; y++){
            temperature[ti(x,y)] = A*exp(-(  (x-x_0)*(x-x_0)/(2.0*
                stn_dev_x*stn_dev_x) +  (y-y_0)*(y-y_0)/(2.0*stn_dev_y*
                stn_dev_y) ));
            material[mi(x,y)] = MERCURY * (dt/(h*h));
        }
    }
}


void print_local_temps(int step){

    MPI_Barrier(cart);
    for(int i = 0; i < size; i++){
        if(rank == i){
            printf("Rank %d step %d\n", i, step);
            for(int y = -BORDER; y < local_grid_size[1] + BORDER; y++){
                for(int x = -BORDER; x < local_grid_size[0] + BORDER; x++)
                    {
                    printf("%5.1f ", local_temp[step%2][lti(x,y)]);
                }
                printf("\n");
            }
            printf ("\n");
        }
        fflush(stdout);
        MPI_Barrier(cart);
    }
}

/* Save 24 - bits bmp file, buffer must be in bmp format: upside - down */
void savebmp(char *name, unsigned char *buffer, int x, int y) {
  FILE *f = fopen(name, "wb");
  if (!f) {
    printf("Error writing image to disk.\n");
    return;
  }
  unsigned int size = x * y * 3 + 54;
  unsigned char header[54] = {'B', 'M',
                      size&255,
                      (size >> 8)&255,
                      (size >> 16)&255,
                      size >> 24,
                      0, 0, 0, 0, 54, 0, 0, 0, 40, 0, 0, 0, x&255, x >> 8,
                           0,
                      0, y&255, y >> 8, 0, 0, 1, 0, 24, 0, 0, 0, 0, 0, 0,
                           0,
                      0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                           0};
  fwrite(header, 1, 54, f);
  fwrite(buffer, 1, GRID_SIZE[0] * GRID_SIZE[1] * 3, f);
```

```
  fclose(f);
}

/* Given iteration number, set a colour */
void fancycolour(unsigned char *p, float temp) {
    float r = (temp/101) * 255;

    if(temp <= 25){
        p[2] = 0;
        p[1] = (unsigned char)((temp/25)*255);
        p[0] = 255;
    }
    else if (temp <= 50){
        p[2] = 0;
        p[1] = 255;
        p[0] = 255 - (unsigned char)(((temp-25)/25) * 255);
    }
    else if (temp <= 75){

        p[2] = (unsigned char)(255* (temp-50)/25);
        p[1] = 255;
        p[0] = 0;
    }
    else{
        p[2] = 255;
        p[1] = 255 -(unsigned char)(255* (temp-75)/25) ;
        p[0] = 0;
    }
}

/* Create nice image from iteration counts. take care to create it upside
    down (bmp format) */
void output(char* filename){
    unsigned char *buffer = calloc(GRID_SIZE[0] * GRID_SIZE[1]* 3, 1);
    for (int i = 0; i < GRID_SIZE[0]; i++) {
      for (int j = 0; j < GRID_SIZE[1]; j++) {
        int p = ((GRID_SIZE[1] - j - 1) * GRID_SIZE[0] + i) * 3;
        fancycolour(buffer + p, temperature[(i + GRID_SIZE[0] * j)]);
      }
    }
    /* write image to disk */
    savebmp(filename, buffer, GRID_SIZE[0], GRID_SIZE[1]);
    free(buffer);
}


void write_temp ( int step ){//#
    char filename[15];
    sprintf ( filename, "data/%.4d.bmp", step/SNAPSHOT );

    output ( filename );
    printf ( "Snapshot at step %d\n", step );
}
```

Listing A.2: Particle model implementation

```
#include <stdio.h>
```

```
#include <stdlib.h>
#include <stdbool.h>
#include <math.h>
#include <time.h>
#include <mpi.h>
#include <float.h>
/*velocity field properties*/
const double AMPLITUDE = 0.1;//0.1;
const double OMEGA     = 1.0;
const double EPSILON   = 0.25;
/*material constants*/
const double MERCURY   = 0.00619;//6.19;// 0.0619;
const double two       = 2.0;
const int    one       = 1.0;
#define  M_PI acos(-1.0)

/* Size of the computational grid   */
const int GRID_SIZE[2] ={256,128};

/* Parameters of the simulation: how many steps */
const int NSTEPS = 500000;

const int NPARTICLES = 131072;
/*The reason for not using pow is: if NPARTICLES is declared using pow,
    the number is declared as const. not available on stack.*/

/*
Values for initiating particles. Remember this is the positions not index
    */
double stn_dev_x = 2.0/16.0;  //GRID_SIZE[0]/16.0;
double stn_dev_y = 1.0/8.0;   //GRID_SIZE[1]/8.0;
double x_0 =       1.0;       //GRID_SIZE[0]/2.0;
double y_0 =       0.5;       //GRID_SIZE[1]/2.0;
double A = 1;
double u_1 = 0.0;
double u_2 = 0.0;

const int SNAPSHOT = 25000;

/* Border thickness */
const int BORDER = 1;

/* Arrays for the simulation data */
double
    *material,              // Global material constants, on rank 0
    *temperature,           // Global temperature field, on rank 0
    *velocity[2],           // global velocity field in x and y direction,
         on rank 0
    *local_material,        // Local part of the material constants
    *local_temp[2],         // Local part of the temperature (2 buffers)
    *local_velocity[2],     // Local part of the velocity field
    *total_conservation;    // Array containing the total amount of
        consentraton/heat

/*Datatype for holding the information of the particle*/
struct Particle {
    double x_position;
```

```
    double y_position;
} test_particle;

typedef struct Particle particle;

struct Particle  *total_particle_ensemble;// Array containing the total
    number of particles
struct Particle  *particle_ensemble;      // the number of local particles
//MPI_Particle **particle_ensmeble;

double  *send_array_y,
        *send_array_x;
/* Discretization: 5cm square cells, 2.5ms time intervals */
const double
    h  = 1.0/(128-1),
    dt = 2.5e-5;


/* Local state */
int
    size, rank,                     // World size, my rank
    dims[2],                        // Size of the cartesian
    periods[2] = { false, false },  // Periodicity of the cartesian
    coords[2],                      // My coordinates in the cartesian
    north, south, east, west,       // Neighbors in the cartesian
    local_grid_size[2],             // Size of local subdomain
    local_origin[2];                // World coordinates of (0,0) local

/* Datatypes to be used in making my own datatype for MPI*/
    int blen[3];                        //3
    MPI_Aint array_of_displacements[3];//3
    MPI_Datatype array_of_types[3];     //3
    MPI_Datatype oldtypes[3];           //3
    MPI_Datatype MPI_Particle;


/* Prototypes for functions found at the end of this file */
void solver ( int step);
//void border_exchange ( int step );
//void gather_temp( int step );
//void scatter_temp();
//void scatter_material();
//void scatter_velocity();
void commit_vector_types(void);


//void external_heat ( int step );
void write_temp ( int step );
//void print_local_temps(int step);
void initiate_particle_ensemble(void);

//void enforce_reflective_boundaries(int step);
double double_gyre_x_direc( double x, double y, double t, double A, double
     e, double w);
double double_gyre_y_direc( double x, double y, double t, double A, double
     e, double w);
void conservation(int step);
```

```
void write_conservation_to_file(char* filename);
double random_val(void);
void runge_kutta_solver(float time, int index);
void reset_temperature(void);
void generate_position_file(int number);
// Cartesian communicator
MPI_Comm cart;

// MPI datatypes for gather/scater/border exchange
MPI_Datatype
    border_row, border_col, grid_block, receive_block,
        receive_block_material;

/* Indexing functions, returns linear index for x and y coordinates,
    compensating for the border */

// temperature
int ti(int x, int y){
    return y*GRID_SIZE[0] + x;
}

// material
int mi(int x, int y){
    return ((y+(BORDER-1))*(GRID_SIZE[0]+2*(BORDER-1)) + x + (BORDER-1));
}

// local_material
int lmi(int x, int y){
    return ((y+(BORDER-1))*(local_grid_size[0]+2*(BORDER-1)) + x + (BORDER
        -1));
}

// local_temp
int lti(int x, int y){
      return ((y+BORDER)*(local_grid_size[0]+2*BORDER) + x + BORDER);
}
//this checks if you are inside local area
int inside(int x, int y){
    return x >= local_origin[0] &&
    x < local_origin[0] + local_grid_size[0] &&
    y >= local_origin[1] &&
    y < local_origin[1] + local_grid_size[1];
}

void commit_vector_types ( void ){//#
    /* TODO: Create and commit the types for the border exchange and
        collecting the subdomains */
    blen[0] = 1; array_of_displacements[0] = 0;
        oldtypes[0] = MPI_DOUBLE;
    blen[1] = 1; array_of_displacements[1] = sizeof(double);
        oldtypes[1] = MPI_DOUBLE;
    blen[2] = 1; array_of_displacements[2] = sizeof(test_particle);
        oldtypes[2] = MPI_UB;
    MPI_Type_create_struct( 3, blen, array_of_displacements, oldtypes, &
        MPI_Particle );
    MPI_Type_commit(&MPI_Particle);
```

```
    MPI_Barrier(cart);
    MPI_Bcast(&test_particle, 1, MPI_Particle, 0, cart);
    MPI_Barrier(cart);
   }


void solver( int step ){//#
    /* TODO: Implement solver */
    for(int i = 0; i < NPARTICLES/size; i++){
          runge_kutta_solver((step)*dt, i);
  }

}


void gather_particles(void){//#
    /* TODO: Collect all the local subdomains in the particle array at
        rank 0 */
    MPI_Barrier(cart);
    MPI_Gather(&particle_ensemble[0],NPARTICLES/size,MPI_Particle,&
        total_particle_ensemble[0],NPARTICLES/size,MPI_Particle,0,cart);
    MPI_Barrier(cart);
}


void runge_kutta_solver(float time, int index){//#
      double k_1_x,
             k_1_y,
             k_2_x,
             k_2_y,
             k_3_x,
             k_3_y,
             k_4_x,
             k_4_y;

      double x_val = particle_ensemble[index].x_position;
      double y_val = particle_ensemble[index].y_position;
      double val1 = sqrt(4.0*MERCURY*dt);
      double temp;
      k_1_x = double_gyre_x_direc(x_val, y_val, time, AMPLITUDE, EPSILON,
          OMEGA);
      k_1_y = double_gyre_y_direc(x_val, y_val, time, AMPLITUDE, EPSILON,
          OMEGA);

      k_2_x = double_gyre_x_direc(x_val + dt/2*k_1_x, y_val + dt/2*k_1_y,
          time + dt/2, AMPLITUDE, EPSILON, OMEGA);
      k_2_y = double_gyre_y_direc(x_val + dt/2*k_1_x, y_val + dt/2*k_1_y,
          time + dt/2, AMPLITUDE, EPSILON, OMEGA);

      k_3_x = double_gyre_x_direc(x_val + dt/2*k_2_x, y_val + dt/2*k_2_y,
          time + dt/2, AMPLITUDE, EPSILON, OMEGA);
      k_3_y = double_gyre_y_direc(x_val + dt/2*k_2_x, y_val + dt/2*k_2_y,
          time + dt/2, AMPLITUDE, EPSILON, OMEGA);

      k_4_x = double_gyre_x_direc(x_val + dt*k_3_x, y_val + dt*k_3_y, time
           + dt, AMPLITUDE, EPSILON, OMEGA);
      k_4_y = double_gyre_y_direc(x_val + dt*k_3_x, y_val + dt*k_3_y, time
```

```
            + dt, AMPLITUDE, EPSILON, OMEGA);

      particle_ensemble[index].x_position = x_val + (dt/6.0)*(k_1_x + 2.0*
          k_2_x + 2.0*k_3_x + k_4_x) + random_val()*val1;

      particle_ensemble[index].y_position = y_val + (dt/6.0)*(k_1_y + 2.0*
          k_2_y + 2.0*k_3_y + k_4_y) + random_val()*val1;

      /*Enforcing BC*/
      if (particle_ensemble[index].x_position < DBL_EPSILON){
          temp = particle_ensemble[index].x_position;
          particle_ensemble[index].x_position = 0.0 - temp;
      }
      else if (particle_ensemble[index].x_position > 2.0){
          temp = particle_ensemble[index].x_position;
          particle_ensemble[index].x_position = 2.0 - (temp - 2.0);
      }
      if(particle_ensemble[index].y_position < DBL_EPSILON){
          temp = particle_ensemble[index].y_position;
          particle_ensemble[index].y_position = 0.0 - temp;
      }
      else if(particle_ensemble[index].y_position > 1.0){
          temp = particle_ensemble[index].y_position;
          particle_ensemble[index].y_position = 1.0 - (temp - 1.0);
      }
}

double random_val(void){ //#
      return (rand()*(1.0/RAND_MAX)*2.0*sqrt(3.0) - sqrt(3.0))*(1.0/sqrt
          (2.0));
}

int main ( int argc, char **argv ){
    MPI_Init ( &argc, &argv );
    MPI_Comm_size ( MPI_COMM_WORLD, &size );
    MPI_Comm_rank ( MPI_COMM_WORLD, &rank );

    MPI_Dims_create( size, 2, dims );
    MPI_Cart_create( MPI_COMM_WORLD, 2, dims, periods, 0, &cart );
    MPI_Cart_coords( cart, rank, 2, coords );

    MPI_Cart_shift( cart, 1, 1, &north, &south );
    MPI_Cart_shift( cart, 0, 1, &west, &east );


    commit_vector_types();

    /*Heating up randomnumber generator. */
    time_t t;
    srand((unsigned) time(&t));


    if(rank == 0){//rank 0 initializes the global temperature grid
        size_t temperature_size = GRID_SIZE[0]*GRID_SIZE[1];
        temperature = calloc(temperature_size, sizeof(double));
        total_particle_ensemble = calloc(NPARTICLES, sizeof(particle));
      }
```

```
    particle_ensemble = calloc( NPARTICLES/size, sizeof(particle));

    initiate_particle_ensemble();
    // Main integration loop: NSTEPS iterations, impose external heat
    for( int step=0; step<NSTEPS; step += 1 ){

              solver(step);
              if((step % SNAPSHOT) == 0){

                gather_particles();

              printf("after gather of particles \n");
                if(rank == 0){
                  generate_position_file(step);
                //  conservation(step);
                printf("after heatmap \n");
                  reset_temperature();

                printf("after reset temp \n");
                }
              }

    }

    if(rank == 0){
        //write_conservation_to_file("
            concentration_dump_low_not_reflective_bc.txt");
        free (temperature);
        free (total_particle_ensemble);
        free (total_conservation);
    }

   free (particle_ensemble);
   MPI_Type_free(&MPI_Particle);
   MPI_Finalize();
   exit ( EXIT_SUCCESS );
}



/*void generate_heatmap(int number){
  int i,j;
  for (int k=0; k < NPARTICLES; k++){
        i = total_particle_ensemble[k].x_position/h;
        j =  total_particle_ensemble[k].y_position/h;
        if(i<0){
        i=0;
        printf("negative i idex \n");
        }
        if(j<0){
        j=0;
        printf("negative j idex \n");
        }
        temperature[ti( (int)i, (int)j)]+= A;
  }
    write_temp(number);
}
```

```
*/
void generate_position_file(int number){//#

        char filenamex[16];
        char filenamey[16];

        sprintf (filenamex, "data/x%.4d.bin", number/SNAPSHOT );
        sprintf (filenamey, "data/y%.4d.bin", number/SNAPSHOT );
        FILE *fx = fopen(filenamex,"w");
        FILE *fy = fopen(filenamey,"w");
        if (fx == NULL)
        {
            printf("Error opening file!\n");
            exit(1);
          }
        if (fy == NULL)
        {
            printf("Error opening file!\n");
            exit(1);
          }
         for (int k=0; k < NPARTICLES; k++){
                fwrite( &total_particle_ensemble[k].x_position,8,1,fx);
                fwrite( &total_particle_ensemble[k].y_position,8,1,fy);

        }
        fclose(fx);
        fclose(fy);
}



void reset_temperature(void){//#
      for (int k=0; k < GRID_SIZE[0]*GRID_SIZE[1]; k++){
          temperature[k]=0.0;
      }
}


void write_conservation_to_file(char* filename){

  FILE *f = fopen(filename, "w");
  if (f == NULL)
  {
    printf("Error opening file!\n");
    exit(1);
  }
  /* rather unsure of this fprintf  */
  for(int i=0; i<NSTEPS/SNAPSHOT; i++){
    fprintf(f,"%f\n",total_conservation[i]);
  }
  fclose(f);
}

void conservation(int step){
      total_conservation[step/SNAPSHOT] = 0.0;
      for (int i=0; i<GRID_SIZE[0]*GRID_SIZE[1]; i++){
```

```
            total_conservation[step/SNAPSHOT] += temperature[i];
        }
}



double double_gyre_x_direc( double x, double y, double t, double A, double
     e, double w){
    double a = e * sin(w*t);
    double b = 1 - 2.0*e*sin(w*t);
    double f = a*x*x + b*x;

    return (-1.0*M_PI*A*sin(M_PI*f) * cos(M_PI*y));              // x
        component of velocity
}

double double_gyre_y_direc( double x, double y, double t, double A, double
     e, double w){
    double a = e * sin(w*t);
    double b = 1 - 2.0*e*sin(w*t);
    double f = a*x*x + b*x;

    return (M_PI*A*cos(M_PI*f) * sin(M_PI*y) * (2.0*a*x + b)); // y
        component of velocity
}



void initiate_particle_ensemble(void){//#

    //creates uniformly distributed numbers
    bool condition;

    for(int i=0; i<NPARTICLES/size; i++){
        condition = true;
        while(condition){
          condition = false;

          u_1 = rand()*(1.0/RAND_MAX);
          u_2 = rand()*(1.0/RAND_MAX);
          /*First part calulate the normaly distributed numbers multiplied
               with stndev and added mean*/
          particle_ensemble[i].x_position = sqrt(-2.0*log(u_1))*cos(2.0*
              M_PI*u_2)*stn_dev_x + x_0;
          particle_ensemble[i].y_position = sqrt(-2.0*log(u_1))*sin(2.0*
              M_PI*u_2)*stn_dev_y + y_0;

          if (particle_ensemble[i].x_position > 2.0){
              printf("Out of bonds setting value in x-direction to max.
                  Makes a new sample\n");
              condition = true;
          }
          if (particle_ensemble[i].x_position < DBL_EPSILON){
              printf("Out of bonds setting value in x-direction to min.
                  Makes a new sample\n");
              condition = true;
          }
```

```
            if (particle_ensemble[i].y_position > 1.0){
                printf("Out of bonds setting value in y-direction to max.
                    Makes a new sample\n");
                condition = true;
            }
            if (particle_ensemble[i].y_position < DBL_EPSILON){
                condition = true;
                printf("Out of bonds setting value in y-direction to min.
                    Makes a new sample\n");
            }
        }
    }
}


/* Save 24 - bits bmp file, buffer must be in bmp format: upside - down */
void savebmp(char *name, unsigned char *buffer, int x, int y) {
  FILE *f = fopen(name, "wb");
  if (!f) {
    printf("Error writing image to disk.\n");
    return;
  }
  unsigned int size = x * y * 3 + 54;
  unsigned char header[54] = {'B', 'M',
                      size&255,
                      (size >> 8)&255,
                      (size >> 16)&255,
                      size >> 24,
                      0, 0, 0, 0, 54, 0, 0, 0, 40, 0, 0, 0, x&255, x >> 8,
                           0,
                      0, y&255, y >> 8, 0, 0, 1, 0, 24, 0, 0, 0, 0, 0, 0,
                           0,
                      0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                          0};
  fwrite(header, 1, 54, f);
  fwrite(buffer, 1, GRID_SIZE[0] * GRID_SIZE[1] * 3, f);
  fclose(f);
}

/* Given iteration number, set a colour */
void fancycolour(unsigned char *p, float temp) {
    float r = (temp/101) * 255;

    if(temp <= 25){
        p[2] = 0;
        p[1] = (unsigned char)((temp/25)*255);
        p[0] = 255;
    }
    else if (temp <= 50){
        p[2] = 0;
        p[1] = 255;
        p[0] = 255 - (unsigned char)(((temp-25)/25) * 255);
    }
    else if (temp <= 75){

        p[2] = (unsigned char)(255* (temp-50)/25);
        p[1] = 255;
```

```
        p[0] = 0;
    }
    else{
        p[2] = 255;
        p[1] = 255 -(unsigned char)(255* (temp-75)/25) ;
        p[0] = 0;
    }
}

/* Create nice image from iteration counts. take care to create it upside
    down (bmp format) */
void output(char* filename){
    unsigned char *buffer = calloc(GRID_SIZE[0] * GRID_SIZE[1]* 3, 1);
    for (int i = 0; i < GRID_SIZE[0]; i++) {
      for (int j = 0; j < GRID_SIZE[1]; j++) {
        int p = ((GRID_SIZE[1] - j - 1) * GRID_SIZE[0] + i) * 3;
        fancycolour(buffer + p, temperature[(i + GRID_SIZE[0] * j)]);
      }
    }
    /* write image to disk */
    savebmp(filename, buffer, GRID_SIZE[0], GRID_SIZE[1]);
    free(buffer);
}


void write_temp ( int step ){
    char filename[15];
    sprintf ( filename, "data/%.4d.bmp", step/SNAPSHOT );

    output ( filename );
    printf ( "Snapshot at step %d\n", step );
}
```