Massively parallel simulation of dusty protostellar systems

Implementing physically accurate particle simulation for large scale simulations

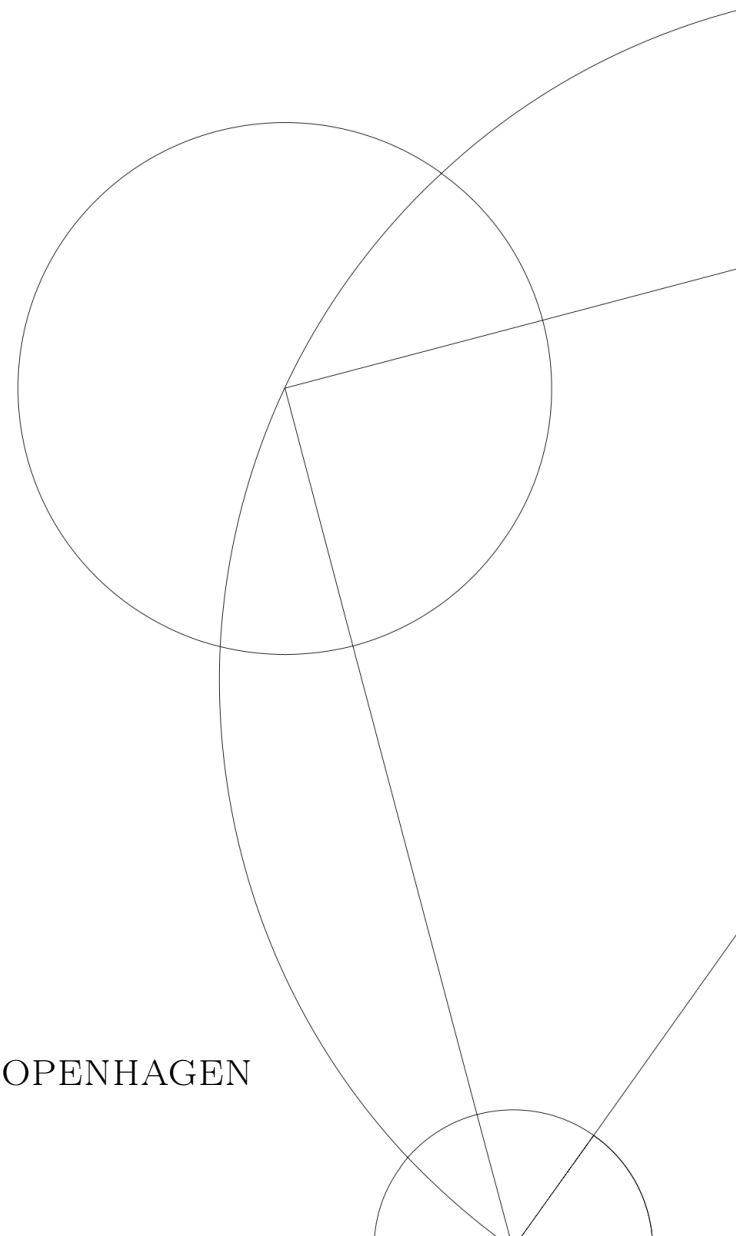
Masters project

Written by *Rasmus Damgaard Nielsen*

September 3, 2023

Supervised by

Troels Haugbølle and Åke Nordlund



University of Copenhagen

UNIVERSITY OF COPENHAGEN

Name of Institute:	NBI
Name of Department:	Department of science
Author(s):	Rasmus Damgaard Nielsen
Email:	xsz511@alumni.ku.dk
Title and subtitle:	Massively parallel simulation of dusty protostellar systems - Implementing physically accurate particle simulation for large scale simulations
Supervisor(s):	Troels Haugbølle and Åke Nordlund
Handed in:	1. September 2023
Defended:	??

Name Rasmus Damgaard Nielsen

Signature Damgaard

Date 03-09-2023

Abstract

The formation of protoplanetary and protostellar systems is a complex and heterogeneous astrophysical phenomenon driven by the interaction between magnetohydrodynamical dynamics, selfgravitational forces, radiation, dust etc., all occurring at temporal and spatial scales spanning many orders of magnitude. These complexities and more make studying the system very difficult, and especially when it relates to the dust behaviour outside of the mid-plane of a homogeneous disk, many open questions are yet to be answered. One such unknown is the degree to which the outflows of the disk can transport away heated dust particles, enriching the surrounding GMC.

In this study, we introduce a high-performance dust simulation incorporated into the state-of-the-art three-dimensional code, DISPATCH[1], integrating dust simulation into a highly detailed model of the magnetohydrodynamics and self-gravity in the system, modelling full dust-gas feedback. This allows realistic physical modelling of these complex environments all the way from initial collapse to disk formation. This approach provides a level of detail in studying the process of planetary formation that would not otherwise be possible, and its usefulness will continue to increase with the rapid development of supercomputer hardware.

With this implementation, we were able to simulate a protoplanetary system discretized into 1.5 million cells of gas and magnetism, as well as 60 million dust particles through more than 10,000 years of early evolution, at a cost of 150ns/particle update. Our findings validate the implementation's accuracy. Furthermore from this simulation, we can conclude that the rate of particles passing near the star and being ejected is highly particle-size dependent but may be up to on order 10% of the total dust budget. This result is validated by analysing gas tracers simulated by the simulation framework RAMSES, although further study at higher resolution is needed.

With this implementation, we have thus taken a step towards enabling further computational studies of the behaviour of gas in the highly complex environment that is planet formation.

Contents

1 Introduction

The formation of planets and stars is a complex and multifaceted problem that has long been studied in astrophysics. One of the key components in this process is dust. Though it may seem insignificant at first glance, it plays a crucial role in these systems, for one because they are the building blocks of planets.

Protoplanetary systems are also quite special, in that they are a uniquely high-density region that varies in temperature from the deeply embedded central disk of only tens of kelvin to the innermost part of the disk which is heated by the star can reach much higher temperatures [2]. This allows complex chemical reactions to occur in these regions [3].

Moreover, a solid understanding has large observational implications, since dust is the main opacity source in these systems. This means that any interpretation of observations of protoplanetary systems will inevitably draw assumptions from dust models to arrive at the physical properties.

However, the temporal and spatial scales, as well as the breath of physical processes involved make comprehensive modeling of these systems a daunting challenge: the molecular cloud in which the protostar is formed has scales of millions of astronomical units (AU) and exists for millions of years, while the planets are formed at scales of AU and have dominant dynamical timescales similar to the orbital timescales of years. Moreover, the environment within these disks is not calm and orderly. Turbulence caused by a variety of factors including the magnetohydrodynamic (MHD) effects of the weakly ionized gas in the disk can cause dust particles to move in unpredictable ways, complicating the process of accretion, and magnetically driven outflows drive outflows of gas and dust moving at immense velocities.

These complexities make the problem of planetary formation a challenging one to study. Observationally it is now possible to observe protoplanetary disks [4] [5], which provides a wealth of insight, but sampling biases, optical thickness, and the inability to explore the three-dimensional structure are natural limitations.

From an analytical perspective, the standard model of accretion disks known as the α disk model [6] simplifies the problem to that of a viscous very thin disk driven by turbulence gravitationally dominated by the central star. Other models apply a semi-analytical approach where the assumptions of a number of symmetries leave only a much simpler one-dimensional model to be computationally solved such as the model for the disk vertical structure by [7]. For a more complete overview of the theory of the formation of planets, see [8]

To augment these observational modeling approaches, astrophysicists have turned to large computer simulations to study this process. However, the vast scales of time and space involved, coupled with the need to simulate the behavior of countless dust particles in a turbulent, magnetized gas with self-gravity, make this a computationally tricky problem.

This is where massively parallel supercomputer models [9][10][11][1] come into play. These models divide the problem into smaller parts, distributing these across thousands of processors working together, distributing the computational resources according to the temporal and spatial scales relevant at each point of the system. This allows for the simulation of complex, three-dimensional environments over long periods of time, making it possible to study the process of planetary formation in a level of detail that would not otherwise be possible. Furthermore, with the rapid development of supercomputer hardware, this approach will only become ever more feasible. The numerical models can give new insights into the processes at play and help in identifying the physics that dominate the evolution, ultimately allowing us to build a better theory for dust dynamics in protoplanetary disks and planet formation.

In this thesis, we describe the implementation of dust simulation in the massively parallel state-of-the-art 3-dimensional magnetohydrodynamical and selfgravitational code DISPATCH [1], allowing highly accurate simulation of planetary formation.

Section 2-5 aims to give a general overview of the physics and computational concerns of the problem, Section 6 describes the specific algorithms developed in this thesis, section 7 presents the results, and finally Section 8 provides conclusion and reflections on future work.

2 The physics of planetary formation

The process of planetary formation is a complex interplay of various physical phenomena, operating over a wide range of time and length scales. To understand how we model this process and why being able to model the dust is so critical from a theoretical and observational perspective, we will in this section present the relevant physical processes and equations that we will use in our model, such as hydrodynamics, magnetism, gravity and drag.

2.1 Formation process

We will first outline the canonical model for stellar formation, with the general physical principles sourced from [13].

The formation of planets is fundamentally the endstage of gravitational instability and collapse in the interstellar medium (ISM). In certain regions near the galactic midplane, a high density of gas and dust collects and can cool down to very low temperature $\sim 20K$ such that the atoms can form molecules, giving these regions the name of "molecular clouds". Turbulence from e.g. shock fronts of supernovae causes the density within these clouds to change, resulting in some regions of higher density[14]. At this point, the self-gravity of some of these over-dense regions might be big enough that pressure can not equalize fast enough to counteract any increase in self-gravity due to contraction, making it gravitational unstable and triggering its collapse. The condition for this instability, leading to contraction under self-gravity, is described by the Jeans criterion, which essentially compares the timescale for sound to propagate across the region with the gravitational free-fall timescale. A sphere of gas with a sound speed c_s and average density ρ will be unstable to density perturbations if its diameter is smaller than

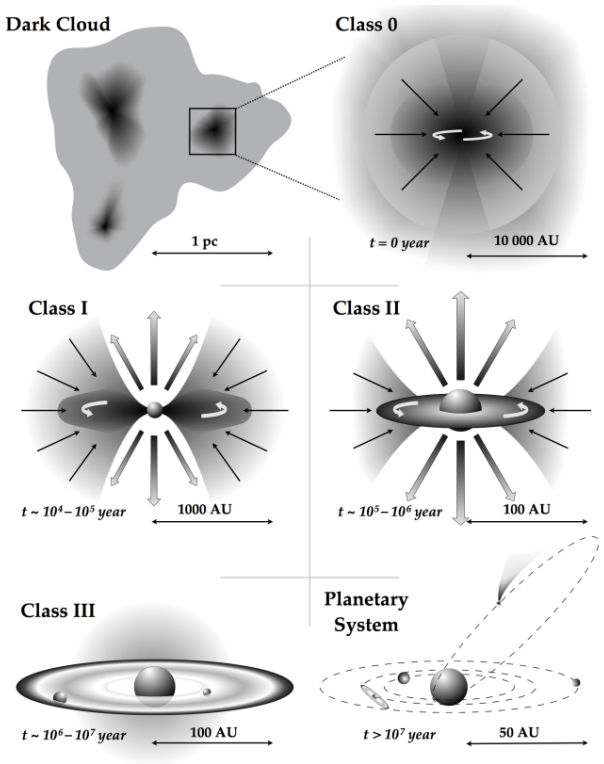


Figure 1: Sketch of stellar and planetary formation involving the relevant scales. For reference, 1pc = 200.000AU. Source: [12]

$$\lambda_J = \sqrt{\frac{\pi c_s^2}{G\rho}} \tag{1}$$

where λ_J is the Jeans length, which typically is of order ~ 0.1pc.

Initially, the core will collapse quickly isothermally, but as the density rises, so does the opacity, and at some point, the contraction will become adiabatic. At this point, heating will drastically slow down the collapse as the core becomes adiabatic and pressure forces counteract the gravitational pull. This object is called the First Larson Core [15]. Slowly shrinking and heating, the center will reach a critical temperature of 2000 K where H2 dissociates. The dissociation of H2 into free hydrogen atoms acts as an energy sink, making the collapse almost isothermal again. This process occurs once again on the free fall timescale, which is much faster than the radiative cooling timescale.

This collapse is halted when all molecular hydrogen is dissociated, and subsequently, hydrogen and helium are ionized, resulting in a new adiabatic core at 100.000 K at the center. The second Larson core, or protostar, is formed. At the same time, conservation of momentum means that the infalling gas

will have any rotational velocity amplified by 2-3 orders of magnitude. Viscosity between the gas will then erase all counteracting rotations, resulting in a protoplanetary disk around the protostar. This disk, partially supported by pressure and partially by centrifugal forces, will stop further collapses and slowly become more and more defined.

From this point on, any flux of gas and dust from the disc will require dissipation of the associated angular momentum to accrete. This dissipation can either happen through viscous diffusion, caused by the differential rotation at different orbits causing shearing turbulence, or it can happen through magneto-rotational instability (MRI) where the disk rotation "winds up" the magnetic lines. These wound-up magnetic lines then couple to the gas through its ionized particles, driving an outflow above and below the disk.

This outflow transports large amounts of gas and dust away, which has been thermally processed by the large temperatures near the star. This could potentially either enrich the chemical composition of the molecular cloud material or, if the material cycles back to the outer disk, enrich the outer regions of the protoplanetary disk with inner-disk material. This is a very difficult process to model analytically, and the vast scales and complexity of driving physics have until recently made full modeling unfeasible computationally. Part of the goal of this project is to open the door to further explore this process computationally.

2.2 Navier Stokes Equations

Fundamentally, any gas is composed of individual particles moving around more or less at random, but luckily for physicists, they do not do so independently. If they are given enough time, any collection of particles bumps into each other and exchanges momentum enough such that they can be well described at any given point simply as a given density having a net velocity $\vec{v}$ and a spread of velocities described by the Boltzmann distribution, parameterized by the temperature field T.

If we take any small box of gas V with a surface ∂V, a few things may happen to this box:

- The gas around the box may push on the box, aka. pressure.
- There may be a differential velocity dragging on the box of gas known as viscosity
- There may be forces acting on the gas in the box, e.g. gravity or magnetism.
- Gas may be accumulating in the box, flowing in through the sides
- The box may move relative to the gas velocity causing advection.

Adding up all of these effects and applying the conservation of momentum and mass, we arrive at the Navier-Stokes equations:

$$\rho \frac{\partial u}{\partial t} = -\vec{u} \cdot \nabla \vec{u} - \nabla p + \nabla \cdot \vec{\tau} + f \tag{2}$$

$$\frac{\partial \rho}{\partial t} = -\nabla \cdot (\vec{u}\rho) \tag{3}$$

The first equation called the Cauchy momentum equation expresses the force balance, and the second equation called the continuum equation expresses mass conservation. τ is called the deviatoric stress tensor, and in all relevant cases here expresses the shearing drag from viscosity.

For numerical simulation, it is advantageous to write directly the time differential of the quantity that needs to be conserved (in this case ρu). Applying the product rule we can rewrite equation 2 to the conservative form:

$$\frac{\partial (\rho u)}{\partial t} = -\nabla \cdot (\rho \vec{u} \vec{u}) - \nabla p + \nabla \cdot \vec{\tau} + f \tag{4}$$

Later we will see how this form allows numerical schemes that conserve e.g. mass and momentum when doing time updates.

As already mentioned, given enough time, any ensemble of particles can be described through a Boltzmann steady-state distribution. The mixing time required to reach the steady state distribution around the mean particle velocity can be estimated as the ratio between the mean free path of the particle l and the typical relative velocity Δv

$$t_{mix} \sim \frac{l}{\Delta v} = \frac{(\sqrt{2}n\sigma)^{-1}}{\Delta v}$$

where σ is the particle cross section and n is the number density. Thus, if we consider free hydrogen atoms with cross-section $\pi \cdot (1\text{Å})^2 \sim 10^{-16}\text{cm}^2$ in molecular clouds where densities are around 100 cm^{-1} the mixing at relative velocities of a few 100 m/s happens on timescales of 100s of years, and class 0 objects with densities 2 orders of magnitude higher has mixing timescales of single years, much shorter than any relevant free-fall or orbital timescales for an object of 10000 AU. Therefore the assumption of homogeneous gas is probably reasonable in stellar and planetary formation theory.

2.3 Thin disk model

Although the Navier-Stokes equations are in general highly nonlinear and complex, under some assumptions they can produce elegant analytical models. In our venture to model dust dynamics in a protoplanetary disk, it is worthwhile to study for comparison the thin disk model of protoplanetary disks[16]. We will look at a very simple model in which we have the following assumptions:

- The system is in approximately hydrostatic equilibrium, such that time differentials can be neglected
- The disk is azimuthally symmetric
- The gas has negligible viscosity
- The disk is thin such that $\sin(\theta) = \theta$
- The shell theorem applies, meaning that gravity is given by $\vec{f} = -G\frac{mM(|r|)}{|r|^2}\vec{r}$ where $|r|$ is the distance from the center and $M(|r|)$ is the total mass within this distance.

Under these assumptions, we can simplify the momentum equations to

$$\rho(\vec{u} \cdot \nabla)\vec{u} = -\nabla p + \rho\vec{g} \tag{5}$$

$$\implies \frac{1}{\rho}\nabla p = -(\vec{u} \cdot \nabla)\vec{u} + \vec{g} \tag{6}$$

We would like to derive the radial and vertical density profiles (the azimuthal being constant), so starting with the radial profile R

$$\frac{1}{\rho}\frac{\partial p}{\partial R} = -\left(u_R\frac{\partial u_R}{\partial R} + \frac{u_\phi}{R}\frac{\partial u_R}{\partial \phi} + u_z\frac{\partial u_R}{\partial z} - \frac{u_\phi^2}{R}\right) - G\frac{M(R)}{R^2} \tag{7}$$

$$= -\frac{\partial u_R^2}{2\partial R} + \frac{u_\phi^2}{R} - G\frac{M}{R^2} \tag{8}$$

$$\implies \frac{u_\phi^2}{R} = G\frac{M}{R^2} + \frac{1}{\rho}\frac{\partial p}{\partial R} + \frac{\partial u_R^2}{2\partial R} \tag{9}$$

We used azimuthal symmetry and mirror symmetry across the disc plane to remove the differentials in ϕ and z, as well $\cos(z/R) = 1$ for the gravitational term. In the typical case, the radial gas velocity will also be completely negligible removing the last term. This means that in the limit of zero pressure gradient, we are left with $u_\phi = \sqrt{\frac{GM}{R}}$, i.e. regular Keplerian rotation. Any non-zero pressure gradient

will be pointing inwards, meaning that the pressure is highest at the center and the gradient with respect to R is negative. Thus the azimuthal (or orbital) speed will be lower than the Keplerian speed. As the disk develops, the balance of these terms changes, going from an initial pressure-supported Class 0 object to a later rotationally supported disk.

Now investigating the vertical profile, we get:

$$\frac{1}{\rho}\frac{\partial p}{\partial z} = -\left(u_R\frac{\partial u_z}{\partial R} + \frac{u_\phi}{R}\frac{\partial u_z}{\partial \phi} + u_z\frac{\partial u_z}{\partial z}\right) - G\frac{M(R)}{R^2}\frac{z}{R}$$

Where we used $\sin(z/R) = z/R$. Furthermore, assuming negligible vertical velocity and velocity gradient with respect to R, we are left with just

$$\frac{1}{\rho}\frac{\partial \rho}{\partial z} = -G\frac{M(R)}{R^3}z$$

Where we recognize the right-hand side to simply be $\Omega_k^2 z$, with Ω_k being the Keplerian rotational frequency. In the thin disk approximation, given that the disk is optically thin and can radiate equally from everywhere, the disk will be vertically isothermal. Under this assumption, we can solve the equation analytically using $p = c_s^2\rho$:

$$\begin{aligned}\frac{c_s^2}{\rho}\frac{\partial \rho}{\partial z} &= -\Omega_k^2 z \\ \implies \rho &= \rho_0 \exp\left(-\frac{\Omega_k^2}{2c_s^2}z^2\right)\end{aligned}$$

Defining the disk scale height $h = c_s/\Omega_k$, we see that the density profile is simply a Gaussian with this width.

The degree to which all of these assumptions apply to the various stages of the protoplanetary disk evolution is debatable, but we can use these equations as first-order approximations of the disk structure, which also gives us some idea of realistic parameters and important structures to capture in our model.

2.4 Self-gravity

Newton's law of gravity might seem innocent enough, but with just 3 interacting bodies, analytically calculating the trajectories is impossible in all but a small selection of cases. For this reason, it shouldn't be surprising that modeling the mutual attraction between any two points of any given density field is completely unfeasible in all but very specific special cases. For planetary models, this means that the best we can do is to treat any mass from the disk or a protoplanet as perturbations to the solar mass. The removal of this assumption by itself can be a big advantage of using numerical simulations.

When simulating self-gravity, it is often useful to not solve for the forces directly, but rather to solve the Poisson equation $\nabla^2\phi = 4\pi G\rho$, and then calculate the gravitational acceleration as $\nabla\phi$.

2.5 Magnetohydrodynamics and its Role in Planetary Formation

One of the most complex but important aspects in understanding planetary formation is the interaction between magnetic fields and gas, the so-called magnetohydrodynamics (MHD). Magnetic fields can play a crucial role in protoplanetary disks due to the low densities and a small ionized fraction in the gas allowing them to build up. Important MHD concepts to model are the transport of magnetic fields, their coupling to the surrounding gas, and the removal of angular momentum from protoplanetary disks. This removal occurs through the magneto-rotational instability (MRI) in the radial direction of the disk [17] and through magneto-centrifugal winds and jets in the vertical direction orthogonal to the disk [18].

2.5.1 The Induction Equation

Magnetic fields are an integral component of many astrophysical systems, including protoplanetary disks. To comprehend how magnetic fields evolve and transport, we turn to the induction equation, a fundamental equation in MHD. The induction equation describes how magnetic fields change over time due to two processes: advection (mass transport) by the flow of a conducting fluid and magnetic diffusion.

Mathematically, the induction equation can be expressed as follows[19]:

$$\frac{\partial \mathbf{B}}{\partial t} = \nabla \times (\mathbf{u} \times \mathbf{B}) - \eta \nabla^2 \mathbf{B} \tag{10}$$

Where $\mathbf{B}$ is the magnetic field vector, $\mathbf{u}$ denotes the velocity vector of the conducting fluid (the gas in our case, if it is at least weakly ionized) and η is the magnetic diffusivity, a measure of how efficiently magnetic fields can diffuse through the gas. This diffusivity is defined as the $\eta = \frac{1}{\mu_0 \sigma}$ with μ_0 being the vaccum permeability and σ being the electrical conductivity.

In equation 10, the first term on the right-hand side represents the advection of the magnetic field by the fluid flow, causing the field lines to be carried along with the motion of the gas. The second term represents the diffusion of the magnetic field, which tends to smooth out and disperse the field lines. The balance between these two processes governs the evolution of magnetic fields in the protoplanetary disks.

An important special case is if the diffusion term is small compared to the advection term. This limiting case is called Ideal MHD, and without dissipation the magnetic field and gas will be "frozen" together, causing the fields to be transported by the gas.

The coupling between the magnetic field and gas can cause magnetic waves to propagate, known as Alfven waves. In the case of Ideal MHD, these have a velocity [20] of

$$v_A = \frac{|\vec{B}|}{\sqrt{\rho \mu}}$$

with μ being the magnetic permeability of the medium. These waves often set the maximal propagation speed of information, which we will see later is very important for the stability of numerical simulation.

2.5.2 Magnetorotational Instability (MRI) and Angular Momentum Transport

The magneto-rotational instability (MRI) [17] is a critical mechanism that governs the removal of angular momentum from protoplanetary disks, allowing material to accrete onto the central star and form planets.

The MRI arises when the gas in the disk rotates at different speeds at various radii causing the magnetic field lines become to wound up, storing magnetic energy. If the gas is sufficiently ionized and the magnetic diffusivity is low enough, the MRI can be triggered.

The MRI is significant only when there is a weak magnetic field and enough free charges (10^{-10} in fractional density) in the disk to cause gas-magnetic field coupling. While the mathematical analysis of the instability is relatively complex, conceptually it is simple. A field line connecting two fluid elements at different radii will be stretched in the angular direction by the differential rotation, due to the frozen-in condition of ideal MHD. The restoring magnetic force will then transfer angular momentum from the inner fluid element to the outer fluid element resulting in the inner fluid element moving inwards and the outer fluid element moving outwards, further stretching the field line and amplifying the field. In the non-linear regime, this creates a turbulent toroidal field which is fed by the energy of the differential rotation.

2.5.3 Magnetic breaking

In the presence of a strong vertical magnetic field during the initial collapse, the magnetic tension force becomes important and the MRI is suppressed [18]. The differential rotation will try to reorient the field into a toroidal geometry and in doing so large amounts of angular momentum of the gas are lost, prompting effective collapse and strong accretion. The angular momentum can be transported away from the system electromagnetically as a Poynting flux related to the winding of the magnetic field lines. Alternatively, it may be removed through the launching of a magneto-centrifugal wind or jet when gas in the upper layers of the disk moving along the field lines reaches an Alfvén velocity larger than Kepler velocity and is thus ejected from the system carrying with itself a large amount of momentum.

2.5.4 Magnetic Pressure

In addition to the magnetic field's role in the MRI and the launching of outflows, magnetic pressure is another important aspect of magnetohydrodynamics (MHD) that significantly influences the dynamics of protoplanetary disks.

Magnetic pressure arises from the energy stored in the magnetic field lines as they become tangled and twisted within the disk. When the magnetic field lines are compressed or stretched due to the motion of the gas, the magnetic pressure increases, exerting a force on the surrounding material.

Mathematically, the magnetic pressure P_{B} can be expressed as [21]:

$$P_{\mathrm{B}} = \frac{|\mathbf{B}|^2}{8\pi} \tag{11}$$

The magnetic pressure competes with the gas pressure within the protoplanetary disk. When the magnetic pressure dominates over the gas pressure, the magnetic field plays a significant role in shaping the disk's structure and dynamics. This situation can lead to various phenomena, including the formation of magnetically supported regions or the launching of powerful outflows.

Furthermore, magnetic pressure can also affect the disk's stability and the formation of structures such as rings and gaps. In regions where magnetic pressure is substantial, the gas may experience magnetic levitation, creating regions of reduced density and influencing the accumulation of material.

Understanding magnetic pressure is crucial for characterizing the complex interactions between magnetic fields and the gas in protoplanetary disks. It has implications for the formation and evolution of planets, as well as for interpreting observational data of young star systems. As research in MHD continues, the role of magnetic pressure in planetary formation remains an exciting and active area of investigation[21].

2.6 Dust-gas interplay

The motion of dust particles within a protoplanetary disk is significantly influenced by drag forces. These forces arise due to the interaction between the dust particles and the gas in the disk. The specific form of the drag force depends on the relative sizes of the dust particles and the gas molecules, as well as their relative velocities [22].

In the Epstein regime, where the dust particles are smaller than the mean free path of the gas molecules, the drag force can be described by the Epstein drag equation

$$\vec{F}_{\mathrm{E}} = -\frac{4}{3}\pi\rho s^2 y_{th}\Delta\vec{v} \tag{12}$$

with $\Delta\vec{v}$ being the gas-dust relative velocity, v_{th} being the gas thermal velocity, ρ being the gas density and v_{th} being the gas thermal velocity.

In the Stokes regime, where the dust particles are larger than the mean free path of the gas molecules, the drag force is instead described by the Stokes drag equation

$$\vec{F}_{\mathrm{S}} = -\frac{1}{2} C_D \pi s^2 \rho |\Delta\vec{v}| \Delta\vec{v} \tag{13}$$

where C_D is the drag coefficient of order unity.

The finite drag on particles means that they will have a non-zero stopping time, i.e. time scale of coming to a stop defined as $\tau_s = m_s/F$. The ratio between this number and the timescales of the relevant dynamics T is defined as the Stokes number $S = \tau_s/T$, which is a relevant quantity to describe the hydrodynamical properties of a particle. The dynamical timescale may be the Keplerian orbital time, or the overturn time of turbulent vortexes. In essence, this number specifies whether the particle is bound to the gas and simply traces it, or whether it moves decoupled from the gas. This will also become important later when discussing how to simulate the equations of motion.

When particles become decoupled from the gas, the dust density might locally become a significant part of the mass. One of the leading explanations for the initial condensation of pebbles of dust into planetesimals is the streaming instability [23]. For this to activate, the dust-to-gas ratio needs to be of order unity. Thus the locations and conditions necessary for dust over-densities are of prime interest to planetary physics.

On the large scales where no significant size selection has occurred, the PDF of dust at different masses takes the shape of power law distribution [24]. In the protoplanetary disk, the density of dust will cause these to condense, fracture, and otherwise evolve. In our project, we will neglect this aspect, but modeling this would be a natural future stem.

There is a priori no reason to believe that any regions would have an over-density of particles would have any given velocity relative to the gas, nor significant variation in the size distribution in different regions. Analysing such shifts from homogeneous distributions is one of the principal applications for a particle solver as part of a larger MHD simulation.

3 Numerical simulation

Numerical simulation is a powerful tool for studying complex physical systems like protoplanetary disks. It allows us to model the behavior of these systems over long timescales and large spatial scales, providing insights that would be difficult to obtain through observation or experiment alone. This section provides an overview of the key concepts and techniques used in numerical simulation.

3.1 Discretization

The first step in doing any computational simulation of a physical system is to perform discretization of the system: trying to simulate the evolution and interaction of virtually infinite numbers of gas and dust particles and the continuous electromagnetic and gravitational fields in continuous time would be a hopeless venture.

The solution to this problem is to approximate the physical quantities (pressure, velocity, magnetic field, gravitational gradient, etc.) with a finite number of variables, which can then be advanced with finite time steps to approximate the time evolution. Such a numerical scheme is called consistent if the error in the approximation vanishes as the number of variables and time steps are increased to infinity.

In theory, given realistic boundary and initial conditions, enough patience, and a powerful enough computer, not much more knowledge is needed to perform a numerical simulation of these systems: do a naive discretion of the equations with an almost infinitely high resolution in both time and space, and the output will be arbitrarily close to the analytical result. But of course, we would like to model the physics as accurately as possible within the given computational constraints, and the rest of section 3 and 4 will explore the techniques used to achieve this goal.

To maximize the "physics for bucks", it is important to consider the scales relevant to the physics under study. If the resolution chosen is far too large, then computational time and memory are wasted, and if it is too coarse, then relevant physics may be smoothed over. In this project we would like to capture the physics relevant to star and planet formation, thus it is important to capture the orbital timescale and the outflow width in the simulation.

3.2 Grid methods vs Particle methods

In numerical simulations, the fundamental elements of our system either have almost infinitely many variables (the gas particles) or even uncountable many, as in the case of the continuous magnetic and gravitational fields. To simulate these systems, it is therefore necessary to approximate the full complexity with a finite representation which can then be stored and evolved in finite space and time. This spatial discretization of the physics can be performed in two main ways: using grid methods, or by describing the system as a collection of meta-particles. These methods have different advantages and disadvantages, which we will go through here.

3.2.1 Grid methods

In grid methods, we require that all of our variables are represented by scalar and vector fields and that our time evolution can be written in the form of partial differential equations on these fields. For the magnetic and gravitational fields, this is fundamentally true, and in section 2.2 we described how to approximate a gas with such a description, yielding the Navier Stokes equations.

Given these fields and PDEs, we can perform the spatial discretization by defining a so-called mesh[1] which splits the domain into a large number of small cells. In each cell, we then approximate each field to be represented by only a single number or vector (or possibly, one per cell surface if using "staggered" or offset grids, typical for vector quantities).

With this finite representation, we can come up with many ways of similarly discretizing the differential equations. We will here present the finite element method but note that other approaches exist, e.g.

[1] Specifics on how to do this discussed in section 3.5.

the Finite Difference Method.

In the finite element method, we integrate the fluxes over the surfaces between the cells and use these to calculate the changes to the variables in each cell. For instance, consider the equation for the density ρ of gas in the Navier-Stokes equation:

$$\frac{d\rho}{dt} = -\nabla \cdot (\rho \vec{u}) \tag{14}$$

Given a cell with volume $\mathcal{V}_i$ and boundary $\partial\mathcal{V}_i$, the change in the mass m_i within a cell i can be found by integrating both sides over the boundary:

$$\begin{aligned}
\frac{dm_i}{dt} &= \frac{d}{dt}\int_{\mathcal{V}_i} \rho dV && && (15)\\
&= \int_{\mathcal{V}_i} \frac{d\rho}{dt} dV && \text{Assuming smooth fields} && (16)\\
&= \int_{\mathcal{V}_i} \nabla \cdot (\rho \vec{v}) dV && \text{Applying Naviers-Stokes} && (17)\\
&= \int_{\partial\mathcal{V}_i} \hat{n} \cdot (\rho \vec{v}) dA && \text{Gauss theorem} && (18)
\end{aligned}$$

And thus we see that the change in the mass is given by the total flux through all the boundaries. An analogous procedure can be performed for the other simulated quantities such as the internal energy, momentum, magnetic field, etc., integrating the respective PDEs over each cell.

The next step is to actually calculate the integral of the field values over the faces of each cell. While the integration of the PDE over the volume of the cell was simply an analytical rewriting with no approximation, this calculation of the flux is where the approximation comes in. Since we only store an approximated field representation, we can never calculate this integral exactly, but many numerical methods exist for doing this with maximal accuracy. For a simple cubic cell mesh of side length Δx we can simplify this to first order as a sum over each face, with $\vec{v}^j$ being the j^{th} component of the velocity, and l_j and u_j subscripts denoting values at the lower and upper face of the cell:

$$\begin{aligned}
\frac{dm_i}{dt} &= \int_{\partial\mathcal{V}_i} \hat{n} \cdot (\rho \vec{v}) dA && && (19)\\
&\approx \sum_{j=1}^{3} -\rho_{l_j} \vec{v_{l_j}^j} \Delta x^2 + \rho_{u_j} \vec{v_{u_j}^j} \Delta x^2 && * && (20)\\
&\approx \Delta x^3 \sum_{j=1}^{3} \frac{d}{dx^j}(\rho \vec{v}^j) && \text{First order taylor expansion} && (21)
\end{aligned}$$

* Since to first order the average of the integrand is simply the central value.

It is worth noting that no matter how precisely we estimate the fluxes, as long as the estimation of the flux through a cell face is done consistently from both sides, we still have mass conservation. This is a central advantage of this approach. For the magnetic field update, it is additionally desirable to conserve $\nabla \cdot B = 0$, and so a special algorithm for calculating the fluxes called "constrained transport" [25] is used.

To first order, one might just assume the flux to be linear between cell centers as in the above example. This might work for some applications, but by fitting polynomials to a whole region of cells, we can approximate the integral to higher accuracy, meaning that a given field can be represented using fewer

grid points. This comes at the cost of making each timestep more computationally costly (as well as possibly more unstable), so the optimal degree of the differentials depends on the specifics of the problem and should be experimentally tuned.

Within this framework, it is natural to simulate gravity by solving the Poisson equation $\nabla^2\phi = 4\pi G\rho$ for the potential field $\phi(x, y, z, t)$. This equation can be solved in multiple ways, notably the following two:

- In the case of a rectilinear grid (i.e. one where all faces are rectangles), this differential equation can be converted to an algebraic equation using Fourier transforms. By performing inverse Fourier transforms to both sides one gets $-|q|^2\mathcal{F}(\phi) = 4\pi G\mathcal{F}(\rho) \implies \phi = 4\pi G\mathcal{F}^{-1}(-|q|^2\mathcal{F}(\rho))$. Furthermore, if the grid has a constant size in each direction, this can be calculated using the Fast Fourier Transform (FFT). The problem with this method is that boundary conditions on the potential or non-cyclic boundaries cannot be solved with this method.
- Alternatively, by discretizing the equation, one can write down the N^3-dimensional problem (N being grid side length) as a linear system to be solved. Although this is far too complex to solve directly through matrix inversion (since matrix inversion is cubic, the total complexity would be $O(N^9)$), the Jacobi method can solve the problem approximately to arbitrary precision by taking a guess for the solution and refining this iteratively. The issue here is that the Jacobi iteration acts as a diffusion operator on the potential, only approximating the local problems in each step, meaning that the information takes N^2 steps to be communicated across the N cells. The issue is fundamental that the global problem is solved as a large number of local problems, only slowly building up the global solution.

  To speed up the convergence of Jacobi iterations, multigrid methods work by exploiting the linear nature of the Laplace equation, solving the gravitational potential of a lower-resolution copy of the mass distribution, then refining this low-resolution approximation to the global potential to also approximate increasingly more local structures[26]. This approach can reduce the error of the solution to the gravitational potential in linear time in the number of cells, which is asymptotically faster than the fast Fourier method, making this approach very desirable.

  Since the time taken by this approach is mainly dependent on the accuracy of the initial guess, the quality of this is important. Conveniently, the gravitational potential changes according to the transport of mass, and the mass distribution in a single timestep only changes the level of a fraction of a cell, therefore the solution to the previous timestep is a nearly perfect guess. This means that the solver only has to solve the difference in the potential at every iteration, a much simpler problem.

3.2.2 Particle mathods

Unfortunately, while the grid methods lend themselves naturally to simulating gasses and MHD, they lend themselves badly to describing dust since the equilibration timescale ($\sim$ the drag timescale) may be arbitrarily long. This may also be a problem even for gasses in some regions, like the outflow where high-temperature, high-velocity gas meets colder stationary gas, or in low-density regimens like the ISM where cold, warm, and hot gas components coexist. With the assumption of a finite number of different populations, these can be modeled using multiple fluid fields [9].

A particle description, on the other hand, models the system using a collection of so-called metaparticles, each representing a population of gas or dust particles with a given position, velocity, and other properties. These individual metaparticles can then be updated using the equation of motion given by drag, gravity, and other potential forces. One such code is Phantom [11]. In this way, multiple coexisting populations can exist at any given position. Additionally, the trajectories can be tracked, which can have scientific advantages. On the other hand, these advantages come with some costs:

- Interactions are more difficult to handle efficiently in this setup. Gravity is an all-to-all force, and drag makes each metaparticle interact with nearby metaparticles, a neighbor relationship

that has to be updated frequently. To avoid having to consider every pair of metaparticles, of which there are quadratically many, it is essential to have some efficient setup for reducing this problem to a sub-quadratic one. The typical approach taken is using spatial tree structures, where the space is represented by a tree structure with each node representing a partitioning of the previous level, with the particles stored in the leaves or nodes of this structure. This makes it possible to query for particles within some sphere of a given (small) radius from a given particle by recursively exploring nodes covering regions intersecting with this sphere. This makes collision detection and handling feasible.

- Gravity can be handled by using the method of softening to the acceleration [27]. Consider the gravitational force from a small group of particles within a distance of ΔL from each other on a particle a distance of L away. If $\Delta L \ll L$, this force is well approximated by the gravitational attraction from their center of mass. The leading term in the relative acceleration error is the dipole correction which is at most of order $\frac{\Delta L}{L}$. Thus for an efficient procedure for evaluating the gravity of all particles, we store the total mass and centre of mass of all particles within each branch of the spatial tree. For each particle, we recurse through the tree until the size of the region results in a maximum relative error in the gravitational acceleration that is smaller than some softening tolerance ϵ (or some other criteria, e.g. based on the contained mass). This method comes with the advantage that the solution of the potential tracks the resolution in the particle distribution. Alternatively, if a certain minimum scale is already known, e.g. in the case where the particle method is coupled to a grid, the mass may be deposited to the grid and the gravitational potential and force can be calculated at the grid scale using the methods outlined above.

- While the fluid description naturally handles density variations across many orders of magnitudes (like the one occurring in the gravitational collapse of a prestellar core) by simply varying the number representing the density, such a density variation in the particle description corresponds to a large number of metaparticles condensing into a small area. Thus an excessive amount of computational resources are used in a small area, while the dynamics in the area with low density may not be captured at all. This sampling problem can be remedied by splitting particles far from any others, while pruning particles very close to each other with similar velocities and representing the same particle type (e.g. 1 mm dust grains). This procedure can be performed efficiently when the drag timescale is very short, e.g. due to high density, since many particles will in this case be co-moving and can be merged, but some variation in velocity is inevitable and will in general result in a loss of information.

3.2.3 Mixed Methods

As demonstrated the grid and particle methods have different strengths and weaknesses. The grid methods are highly regular and fast and represent gravitational potentials, magnetic fields, and equilibrated gasses nicely. On the other hand, their fundamental property of representing single-valued fields lends itself badly to simulating dust, planets, stars, or other "particles" that are collisionless and therefore potentially multi-valued in a small volume. The grid methods on the other hand have the exact opposite properties. As is typical in numerical schemes, it is advantageous to consider a heterogeneous approach. By carefully overlaying particles moving on a background of gas and magnetic fields, one can reap the benefits from both methods at the cost of programmatic complexity. The interactions within this heterogeneous setup can be handled as follows:

- Gravity can be handled by the multigrid Jacobi methods introduced above acting on a source term of the total gravitational mass, i.e. the sum of gas and particles within each cell. The resulting gradient of the potential which is calculated to accelerate the gas similarly is added to the dust acceleration.

- Drag between dust and gas works by calculating the momentum transfer in a timestep from the dust to the gas in the cell they exist in, and moves that amount of momentum from the dust to the gas.

- In a similar way, accretion can be performed by subtracting mass and corresponding momentum from the dust particles and gas cells close to a given accretion target and adding it to the accretion target.

3.3 Time evolution

With the proper spatial representation of the system, we can evaluate a suitable approximation to $\frac{du}{dt} = h(u, x, t)$ for any property u where u is an arbitrary function, possibly involving spatial differentials. Mathematically, we would like to evaluate $u(t + dt) = u(t) + h(u, x, t)dt$ continuously, but numerically we have to make do with finite evaluations with some finite Δt. We will take as examples three obvious choices for how to do this discretization (where i and $i + 1$ subscripts denote the value at time t and $t + \Delta t$ respectively):

$$u_{i+1} = u_i + h(u_i, x_i, t_i)\Delta t \qquad \text{Forward time diff} \tag{22}$$

$$u_{i+1} = u_i + h(u_{i+1}, x_{i+1}, t_{i+1})\Delta t \qquad \text{Backward time diff} \tag{23}$$

$$u_{i+1} = u_i + \frac{h(u_i, x_i, t_i) + h(u_{i+1}, x_{i+1}, t_{i+1})}{2}\Delta t \qquad \text{Centered time diff} \tag{24}$$

The forward differentiation scheme is called an "explicit" scheme since the right-hand side can readily be calculated explicitly from the current state of the system. In contrast, the backward and centered schemes are "implicit" since they are formulated using the system state in the future and are thus implicit equations in u_{i+1}. For special cases, this can be derived explicitly, but in general, an iterative solver is used for this. Explicit schemes are typically significantly cheaper to evaluate, but at the cost of often requiring a large number of evaluations, or even becoming unstable. To illustrate this, let's look at a particle moving under drag in a steady liquid, in units where the stopping time is unitless:

$$\frac{dv}{dt} = -v \tag{25}$$

The three discretizations of this problem look as follows:

$$v_{i+1} = v_i - v_i\Delta t \qquad \text{Forward time diff} \tag{26}$$

$$= v_i(1 - \Delta t) \tag{27}$$

$$v_{i+1} = v_i - v_{i+1}\Delta t \qquad \text{Backward time diff} \tag{28}$$

$$\implies v_{i+1} = \frac{v_i}{1 + \Delta t} \tag{29}$$

$$v_{i+1} = v_i + \frac{-v_i - v_{i+1}}{2}\Delta t \qquad \text{Centered time diff} \tag{30}$$

$$\implies v_{i+1} = -v_i\frac{\Delta t - 2}{\Delta t + 2} \tag{31}$$

$$\tag{32}$$

It is clear that forward differentiation breaks catastrophically at $\Delta t = 2$: At this point, the time update exponentially increases the velocity while flip-flopping the direction. Intuitively this is because we linearly extrapolate the deceleration far beyond time where this is physical. This instability at large-time steps is normal for explicit schemes and is characterized using the Courant–Friedrichs–Lewy condition. This heuristically states that the time-step Δt has some maximum value (in this case, 2 times the stopping time) above which it does not converge. Therefore we should choose the timestep to be some fraction (called the courant number, C) of this maximum stable timestep.

Calculating this maximal stable value is difficult for all but the simplest schemes, and will typically depend on the state of the system, like the gas viscosity in the example above. For grid-based methods, the Courant number is often a function of the smallest time for information propagation across a cell, i.e. in a simple hydrodynamic fluid it is

$$\delta t_{max} \propto \max\{v + c_s, v - c_s\}/\Delta x \tag{33}$$

with the exact proportionality constant depending on the exact method used.

The two implicit schemes, on the other hand, have very different behavior for $\Delta t \gg 1$: As we can see from figure 2, the error increases for larger timesteps, but this is simply a misestimation of the exact breaking behavior and not numerically unstable behavior. This means that if we are integrating particles through a low Stokes number, i.e. high-density gas where the stopping time may be orders of magnitude lower than any scales of interest, an implicit integrator may drastically speed up computations.

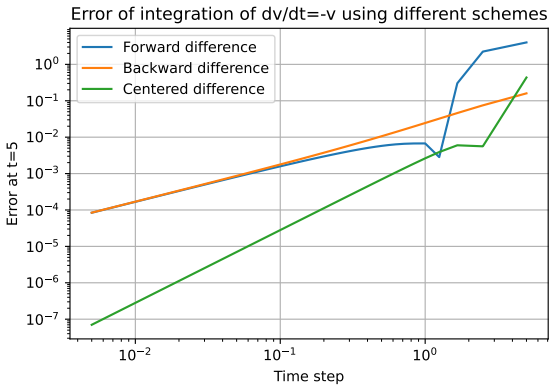


Figure 2: Convergence behavior of various integration schemes. The oscillation breakdown of the implicit schemes is not visible, since they still damp the velocity. The dip for the forward difference is artificial: it simply corresponds to an oscillating solution that randomly is near.

In figure 2 it is also worth noting the slope of the schemes. While the error of the forward and backward difference decreases linearly in the time step, the error of the centered difference scheme decreases quadratically. This shows that while both methods are convergent, meaning that they will in theory exactly replicate the correct physics if allowed small enough time steps, the centered difference method will achieve the desired accuracy using far fewer time steps.

3.4 Convergence, consistency

A discretization of a differential equation is said to be convergent if results from simulations performed at increasingly higher resolutions approach a fixed value, while it is called consistent if this fixed value is physically correct. To show this property mathematically, one must show that any error scales inversely with the resolution, and thus disappear for infinite resolution. Total convergence of a numerical scheme as the resolution approaches infinity is typically impossible, due to the finite precision of floating points on computers, but instead, the scheme is analyzed under a computational model assuming infinite precision.

For example, to show analytically that the forward time difference discretization of the drag problem is consistent, as figure 2 indicated, consider the relative error in the value of v_{i+1} given a value v_i:

$$\begin{aligned}\Delta v_{i+1} &= v_i e^{-\Delta t} - v_i(1-\Delta t) \\ &= v_i(1-\Delta t + O(\Delta t^2) - v_i(1-\Delta t) \\ &= v_i O(\Delta t^2)\end{aligned}$$

Thus the error of each timestep is quadratic in the time step and the method is said to be accurate to first order.

This mathematical proof of convergence gives certainty in the method, but for highly coupled, heterogeneous simulations, showing this is often impossible. Furthermore, for physical study the error of interest is typically not the microscopic error of each quantity, but one of the statistical and/or macroscopic quantities, e.g. the initial mass function, the strength of an outflow, or the flux of dust crossing some barrier, etc. This much weaker "convergence" criteria can be experimentally validated by performing experiments at increasingly higher resolutions, with the assumption that if the quantity approaches some value this is physical, while a spurious result arising due to resolution or quantization effects would disappear at higher resolutions.

Although there are many dials to tune when it comes to varying the resolution, some of which we will discuss in section 3.5, it is often not meaningful to tune these independently. If we decrease the timestep by some orders of magnitude, we may remove a large fraction of the temporal discretization error, but the spatial error remains, meaning that small structures can still not be represented. Worse, as argued in the previous section, increasing the spatial resolution independent of the temporal resolution increases the Courant number and may make the whole simulation unstable. In general, we want to pick the lowest hanging fruit, and so if any individual source completely dominates the error, improving this should be prioritized.

3.5 Mesh definition

When defining the mesh, there are two main decisions to make:

- Regular or irregular: The domain can be split into a regular pattern using eg. wedges, cubes, or some other shape, or it may be using a more irregular partition, shaped to reflect the domain of interest, e.g. according to the shape of a wing in aerospace applications. In general, the regular grid is much more computationally efficient, so in a case like a planet and star formation where the modeled physics is not affected by rigid boundaries present in e.g. engineering problems of the flow around a structure, regular is an obvious choice.

- Static or Adaptive: The mesh defined can either stay constant during the simulation or dynamically adapt to structures that form in the simulation, like turbulence vortices or overdensities. When simulating a star-forming region, we would like to model outflows, disks, envelopes, shocks, etc. with much higher accuracy than the rest of the system. Unfortunately, we do not a priori exactly know where these will form, so the solution is to allow the grid to adapt in a process called adaptive mesh refinement (AMR). This significantly complicates the algorithms, but given the vast range of scales involved, this is a necessary sacrifice.

  As one might imagine, there are many additional implementation choices to make for such an adaptive mesh, such as when to refine, how often to allow the grid to change, etc. One of the most important of these choices is how big of a region to refine at a time: If we refine just a single cell at a time if it exceeds some set criteria, then the mesh will be able to very accurately capture features of interest like shocks or outflows. But on the other hand, the resulting highly irregular mesh is computationally not very efficient. On the other hand, some AMR techniques refine whole blocks of cells at a time, and thus do not quite as precisely refine the areas of interest, but may be more computationally efficient.

The type of mesh refinement used in this project is called patch-based AMR, illustrated in figure 3 against "standard" cell-based AMR. This works using "patches" of 2n x 2n x 2n cells, where each corner of n x n x n cells can be further refined to a new patch of 2n x 2n x 2n cells. This multiplies the scales that can be represented by a factor of a half and the volumes in each cell are reduced to an eights[2]. These coarse regions of refinement have computational advantages, in that a large number of cells ($8n^3$) can be updated in one go, minimizing the computational time used on overhead related to the tree-based mesh for each cell update. If the size of the full domain has a side length of L and k recursive refinement steps are allowed, then the highest resolution cells have a side length of $L_n = \frac{L}{2^k \cdot n} = L \cdot 2^{-k-4}$. Here n is chosen as a trade-off between computational efficiency and the ability to adapt the mesh to the physical structures modeled, with 16 being a reasonable choice that will be assumed going forward. To for example perform a simulation spanning from the scale of a molecular cloud (parsec) to the scale of tens of cells within a single orbit (au), i.e. about 6 orders of magnitude, this requires about 20 levels of recursive refinement.

One complication when using AMR is that to calculate the differential operators for time evolution, it is necessary to gather eg. the gas properties in the neighboring region. If the neighbor patches are at a lower resolution, these might have to be interpolated to the desired position. For very fine-grained AMR, this means that each cell update can require a very large number of accesses to data which

[2]Though other splitting ratios could be chosen

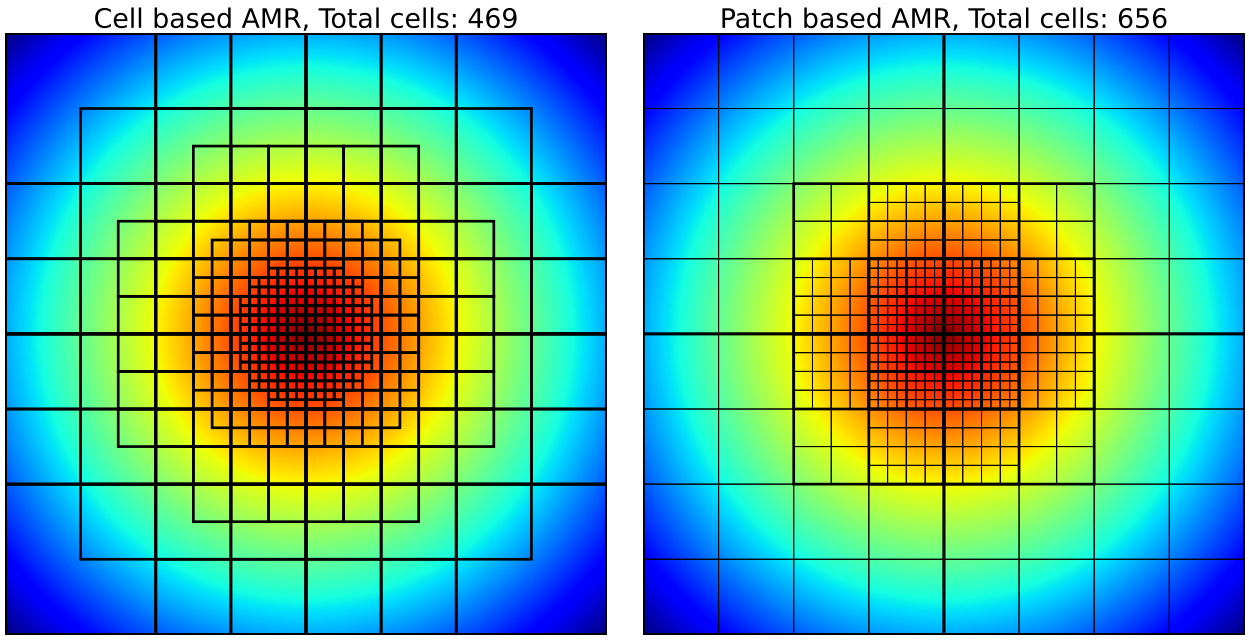


Figure 3: Example of Adaptive Mesh refinement, where the area of interest (the center) has much higher resolution. Thin lines delineate cells, while thick lines delineate separate patches. On the left, 5 levels of refinement allow the resolution to closely follow the physical structure, while the 3 levels of refinement to the right do not follow the borders quite as closely, but the 4x4 cell patches can get the same maximum resolution. Note that the example on the right has neighboring regions with more than 1 level difference, and thus most applications would require additional refined patches to be valid. This means that in practice, it requires even more cells to follow sharp structures.

are scattered around in a not very structured manner. For the patch-based AMR on the other hand, gathering the neighbor data is only necessary on the boundary, drastically reducing the time needed to perform this process.

One important consideration when using a variable grid size is that of timestep size. As explained in section 3.4, the longest timestep allowed is inversely proportional to the cell size. In the example of resolving the planetary orbits in a molecular cloud simulation, the maximum timestep at the highest resolution will therefore also be 6 orders of magnitude shorter. In practice, it will be reduced even more, since even closer orbits now resolved have higher gas and dust velocities. This means that to efficiently perform AMR it is necessary to allow the timestep to vary between different levels in the simulation.

It can be advantageous to consider the mesh as a tree, where the full domain is the root, and the patches in each successive layer of refinement are the children of the patches in the previous layer. In the case of a 3-dimensional grid where each cell can have each of the 8 corners refined, this is an octree. For extreme refinement criteria where all possible cells are refined up to the maximum set level, the mesh is then represented by a complete tree. By considering the mesh in this way, it allows us to make precise statements about the mesh and the relation between any region of the mesh and its neighbors.

When performing time evolution of the system, it is necessary to gather the information from neighbor cells to calculate eg. fluxes consistently. To significantly simplify this procedure, it is advantageous to place spatial restrictions on the mesh, like for the difference between neighboring regions to not have refinement levels differ by more than 1. The patch-based AMR in figure 3 fails on this. The cost is that these restrictions make the AMR tree broader than it would otherwise be. Notably, when a region is chosen to be refined, these requirements may necessitate a cascading of mesh refinements to uphold this requirement since the new regions refined may recursively require even more refinements, resulting in worst case a number of refinements linear in the refinement level.

A last important consideration is how to handle regions of the lower-resolution patches covered in high-

resolution patches: If some cells have been refined, should the data for these cells still be stored and time stepped forward? Although this is technically an unnecessary overhead, the direct children covering any given region will contain 2^3 times as many cells and require ~ 2 as many time steps, meaning that the memory overhead is 1/8 and time overhead is on order 1/16. Even for an infinitely deeply refined region, the overhead of time-stepping the region in parent patches is $\sum_{n=1}^{\infty} \frac{1}{(2^n)^4} = \frac{1/16}{1-1/16} = \frac{1}{15}$. Therefore, duplicating any work at higher levels can be worth it for simplifying code and avoiding a bunch of special conditions.

3.6 Units and precision

To represent the physical quantities in the simulation, computers provide floating point numbers natively in two variants: 32 bits and 64 bits, called float 32 and float 64, or single and double precision floats. Roughly, these represent a number in exponential notation as $(-1)^s 1.f \cdot 2^e$ where s is a bit giving the sign, f is a binary 23 or 52-bit signed number and e is an 8 or 11-bit signed exponent. This means that the largest number representable by 32-bit numbers is on order $2^{2^7} \approx 3.4 \cdot 10^{38}$, and for 64-bit numbers $2^{2^{10}} \approx 1.8 \cdot 10^{308}$.

These might both seem excessively large for any practical application, e.g. but a box with side length 1 pc has a volume of $\sim 3 \cdot 10^{55} \mathrm{cm}^3$, and other quantities like the mass in individual cells can easily have cgs values far larger than that representable by 32-bit numbers. Furthermore, for many intermediate calculations, one might calculate arbitrary powers of these numbers.

For reasons of memory use as well as the higher performance of multiplying 32-bit values on most hardware, we want to overcome this difficulty so we don't have to double our memory budget. The solution is to use code units more appropriate for the specific physics and not waste the numerical range on quantities unfit for the setup. Defining e.g. the unit of length as the total domain width and the unit of mass to equal the total contained mass we avoid the need for extreme exponents. For some calculations, accuracy may still dictate the use of higher precision 64-bit numbers, but not having to use them everywhere has considerable advantages.

4 High-performance parallel computation

High-performance parallel computation is an umbrella term for a large number of techniques used to implement numerical schemes and have them run as fast as possible. For large-scale numerical simulations such as those in this project, this often allows orders of magnitudes more calculations in a given time frame compared to naïve implementations in e.g. python, and is thus critical to cutting-edge computational research.

Any large-scale cutting-edge simulation today is run on supercomputers, which may take up whole warehouses and have thousands of individual computers, or nodes. Each of these nodes contains up to hundreds of individual hardware threads that independently perform calculations for a simulation. An example of such a supercomputer is LUMI, which has 200.00 computing cores and is able to perform 375×10^{15} calculations every second, corresponding to 1.5 million modern laptops[3]. To utilize this hardware optimally, simulation software mirrors the hardware and employs a vast range of techniques. These range from ways to improve the raw performance of the algorithm on the thread level, to how to split the workload across the threads within a node and across the nodes and communicate together to arrive at a global solution to finally how to practically organize the complex programs that result.

This section will present the concepts of high-performance computing used in and relevant to this project. For this reason, this should not be seen as an exhaustive walk-through of concepts in this field, and many important subjects such as GPU computing, NUMA, parallel I/O, and compiler and environment configuration are left out since they were not of prime concern in this project.

4.1 Single core performance

The first step in simulating a numerical scheme as fast as possible is to make sure that the fundamental calculations utilize the hardware efficiently. This means understanding the hardware.

The standard model for a single-threaded CPU is the von Neuman architecture as illustrated in figure 4. In this model, the CPU has a Control Unit (CU) which dispatches instructions to the Aritmetic-Logic Unit (ALU) to perform calculations on numbers and logical comparisons. These calculations are performed on data that exist in the memory hierarchy, ranging from the large-but-slow Random Access Memory, to increasingly faster but smaller caches much closer to the CPU.

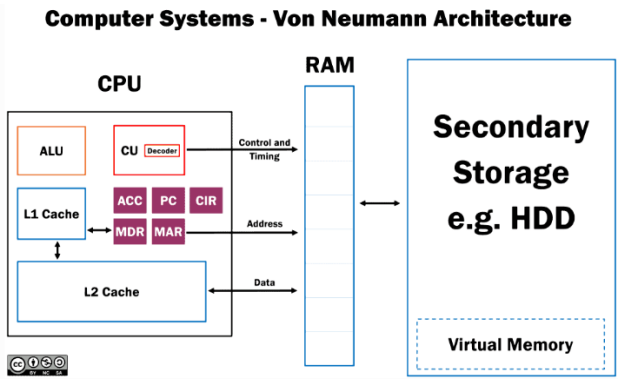


Figure 4: Illustration of the Von Neumann architecture. Source: https://history-computer.com/the-complete-guide-to-von-neumann-architecture/

4.1.1 Pointer-based data structures

The memory on computers can be thought of as one huge array of billions of byte elements, indexed by a pointer. To store any physical quantity like a 3-D array of scalars, we map this to a single 1-dimensional array, and this is then stored in a chunk of the memory array. For systematic structures where we a priori know the data to be stored, we can request that number and size of memory chunks, and store the data there.

For unstructured data structures such as the AMR tree where the amount and location of refined regions are continuously changing, we need to store each refined region independently around the memory so that they can be created and deleted independently. We then couple the different regions by storing pointers from each refined region to e.g. the parents and children, using a special marker value if these do not currently exist called a "null pointer".

[3] Most of this on GPUs

The pointer-based structure thus offers flexibility, but traversing this structure requires so-called pointer chasing where we jump around the memory and have to wait for the next address to be fetched in order to search for the next pointer in a new location, a process known as pointer indirection. This can take a significant amount of time, and so should be minimized as possible. Modern CPUs are good at changing the order of individual operations to keep themselves busy while waiting for memory, a feature called out-of-order execution, but with hundreds of CPU cycles between each memory fetch there is little to do. An efficient design therefore has many calculations that can be performed for each pointer indirection.

4.1.2 Arrays of structures vs structure of arrays

Another important concern for efficient memory use is how CPUs fetch and cache data and how this interacts with the layout of arrays. Consider the case of a structured mesh containing both gas and gravitational potential for each cell. In that case, there are two primary ways to store this in continuous memory: Either we can first store all data for cell 1, then store all data for cell 2, etc., cell by cell, called "Array of Structures". Alternatively, we can choose to store all density data, then the internal energy, then each of the components of the momentum, and finally the gravitational potential, called a "Structure of Arrays". This choice might seem arbitrary, but in actuality can have large performance implications on modern computers.

The memory system stores its data in chunks of typically 64 bytes called cache lines which are always moved together. This means that when the CPU requests e.g. a 4 byte float, the memory system also fetches the next 15 and places them in the closest cache. As other data is subsequently fetched, these newer data elements will displace this cache line in the closest caches.

This means that if we have a large grid and try to run our algorithm for solving the gravitational potential, then out of every consecutive 6 values the algorithm only use 2 before the line is purged, wasting 66% of the memory bandwidth. This is very significant for algorithms that are limited by memory bandwidth.

4.1.3 Temporal locality

The two previous sub-chapters were examples of problems with bad spatial locality, for 4.1.1 because the pointers were far away from each other, and for 4.1.2 because the data of interest were interleaved with data not currently of interest.

Another type of locality of interest is temporal locality. Temporal locality is when an algorithm uses a piece of data and then uses it again shortly after. Imagine that the whole data structure we are working on is only a tiny amount of data that can fit in one of the smallest caches. In this case, after a short time, all data will be in the cache, and thus these problems of low spatial locality matter little: Data access is going to be fast anyway since the cache stores everything.

The canonical example of an algorithm optimization to exploit this principle is chunked matrix multiplication. Consider multiplying two square matrices A and B of size NxN, where only a few rows can fit in the cache. The naive algorithm will loop over each row in A and each column in B and perform a dot product to calculate a value in the result. After calculating the first row of results, everything in the cache has been replaced a few times, and calculating the second row thus involves reloading each column of B all over. This will repeat N times. The solution to this problem is called chunking. Here, we consider our matrices to consist of eg. submatrices of $16x16$ elements (conveniently, 1-2 cache lines per row), and perform the matrix multiplication of each of these. This makes sure that at any time the working data set is probably in the cache, minimizing the time used to reload it.

In numerical simulation, such as grid-based MHD, we run a number of algorithms for each timestep, such as one to determine the max timestep, one to estimate gas flows, one to update the values, etc. If possible, by chunking these updates on parts of the grid that fit in RAM, we reduce required memory traffic from the RAM to the cache from being linear in the number of algorithms to being constant.

One thing to remember if doing this, and then trying to diagnose the run-time performance of the code, is that it might significantly skew relative performance. It can be easy to conclude that one algorithm is extremely slow as compared to the others, but that might be because the time used to load the chunk of data into the cache is attributed to only this algorithm.

4.1.4 SIMD

Besides optimizing the memory access, significant performance is to be found with a proper implementation of the algorithm. In the same way that the cache system is optimized for contiguous access by loading whole lines, the ALU/CU also has similar optimizations that should be considered. Out of these, the most significant is the ability of the ALU to perform so-called vector operations known as the Single Instruction Multiple Data (SIMD) paradigm. These are special instructions that the CU can dispatch to the ALU where a mathematical or logical operation is performed in parallel on multiple pieces (vectors) of data. Different CPUs support different vector lengths, typically 128 or 256 bits. This means that for 32 bit float data, up to 8 grid points/particles can be processed in the same time as 1 normally. This assumes no overhead and that the memory can keep up. Being able to perform twice as many calculations in a given time makes 32 bit floats very attractive over 64 bit floats when core algorithms can be vectorized.

In practice, compilers for languages such as c++ and Fortran are able to detect loops where mathematical and logical operations are applied to simple contiguous loops, and "autovectorize" these[28][28]. From an algorithmic design perspective then, we mainly need to make sure that we are only using vectorizable instructions, and ideally, that array sizes are multiples of the vector lengths. The main restriction here is the use of branches (ie. if statements, and variable length loops) and pointer indirections (different loop iterations jumping unpredictably around memory).

Although one cannot use branches in a vectorized loop, some uses of if statements such as the implementation of piecewise functions or skipping calculations for individual elements can still be performed by calculating both results and merging or only conditionally saving the results for elements in the vectors. In cases where special care needs to be taken for boundary elements, it may also be worthwhile to split loops up to not have to do this duplicate work for every element.

Since vectorization performs operations on vectors of elements, the Structure of Arrays data layout discussed in sec 4.1.2 is beneficial, since this avoids having to gather non-contiguous data elements and shuffle them into a vector.

4.2 Parallel computation

For decades computational physicists could write fast sequential codes, and trust the code becoming faster every year, simply because the hardware improved drastically year by year, as shown in figure 5. Unfortunately, we are not so fortunate anymore. The unstoppable rise of single-core frequencies stopped around 2005 but engineers were able to keep the performance going for another 5 years using ingenious tricks such as SIMD, branch prediction, speculative execution, out-of-order execution, etc. Since 2010, only incremental improvement has been seen in this field, and instead, the main improvement has been made by fitting more and more independent processing cores into the same chip.

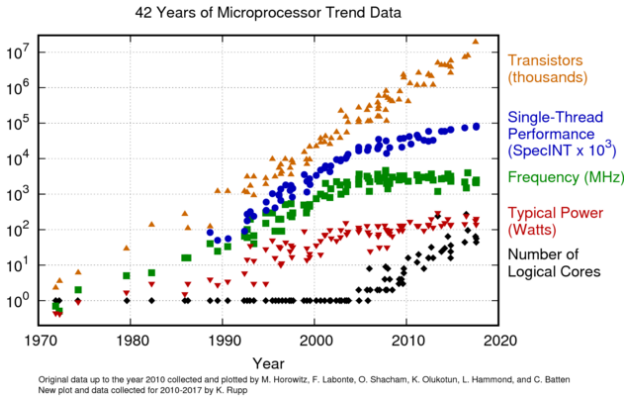


Figure 5: Performance improvement of computer hardware. Source: https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/

Today, individual CPUs contain up to 64 physical cores which can each execute independent code.

To exploit this hardware efficiently, we therefore need to understand how to perform parallelization and map our algorithm to a large number of threads of work that can be run on the cores. On a practical level, this multithreading is often managed in code using the OPENMP standard [29].

4.2.1 Task parallelism

Different ways of thinking about parallelization exist. We have already looked at SIMD which is a version of data parallelism where work units are processing individual pieces of data simultaneously by splitting iterations of a core loop onto multiple threads. This paradigm is also very applicable to GPUs or libraries like NUMPY (where e.g. MATMUL does this). The problem with this paradigm is that for problems with lots of dependencies, all threads will be waiting ("synchronize") until all threads have finished that part of e.g. the loop which is paralleled. This means that if each sub-loop in the simulation is not sufficiently big to parallelize, we will get bad scaling.

In this project, we will instead use the paradigm of task parallelism, where the total work is split into a number of pieces of work which each process their own data. Typically these tasks are posted in a task queue where individual threads that finish their work can fetch a new part of the problem to crunch until all work is done. This more coarse approach to parallelism requires one to explicitly write the algorithm for a smaller part of the problem and think about how the solution to individual subproblems can be combined to solve the global problem. For instance, if one task updates the hydrodynamics on one part of the grid, and another task updates the next part, who handles flows between them? These are issues that have to be solved specifically for every given problem but should be kept in mind.

4.2.2 Scaling

An important concept in parallel computing is that of scaling, i.e. "If I get a computer with twice as many cores, how much faster does my code go"?. Ideally, we would want the answer to be "twice", but many issues may cause this not to be the case. In the same way that we don't expect 36 PhD students to finish a thesis in a month, overheads and communication overhead are all but inevitable. Specifically, we discuss strong and weak scaling. In brief, strong scaling refers to how the computation time decreases as more processing units are used for a fixed problem size. Weak scaling, on the other hand, refers to the ability of a system to handle larger problems as more processing units are added, keeping the runtime constant.

As an extreme example, consider the case of calculating the mass on a mesh. Here, the volume of each cell should be multiplied by the density and added up. The simplest algorithm would be one where a float is initialized to 0, and every thread simply adds to this. Unfortunately, every thread would be sitting almost all of the time and waiting for their turn to add to the value. Worse even, if (as is the case for modern CPUs) each core has its own cache, then thread A writing to the value would mean that thread B's copy was invalid, necessitating significant time used for data transfer between them. As a result, this algorithm would have negative scaling.

In this case, an easy fix exists: By having each thread count up their mass in a local variable and adding them together at the end means each thread only has to modify the global sum once. This shows how algorithmic implementation can affect scaling, and that minimizing communication improves scaling.

4.2.3 Data ownership and locks

In practice, in the above example of threads adding to a shared mass sum, we would not just see performance degradation, but rather a completely wrong result. This is because as one thread reads the sum and is in the process of calculating the new answer, some other thread might already have updated the sum. This is called a read-after-write hazard, a type of race condition where the threads are "racing" to modify the same variable, conflicting with each other. Only by using special much slower "atomic operations", one can get the correct but slow behavior assumed above.

Specifically, this problem is caused by having one thread read from the variable while another writes to it in an unknown order without coordinating their work. Another problematic case is that of multiple

threads writing to the same variable. In this case, the final value of the variable may be that coming from either of them or possibly even a jumbled mess of the output from one and the other. Lastly in the case of both threads simply reading the data, there is no potential issue or conflict. A program taking care of synchronizing the accesses to shared data properly is said to be thread-safe, a property that is necessary for a multithreaded program.

The most important tool at our disposal to avoid these data races is locks, which for this reason is provided in OPENMP. When thread A modifies a piece of data, it sets a flag, "activating the lock" which specifies that the data is unavailable to other threads. When A is done modifying the data, it will unset the flag, deactivating the lock. If a second thread B then tries to modify the data, it will see that the lock is already locked and will wait for its turn. Thus all access will be sequential, avoiding race conditions. Of course, the time spent waiting by B is wasted so this should be minimized as much as possible.

Furthermore when using locks one has to take care to avoid deadlocking, where the program halts due to all threads waiting for a lock. This can be (1) mutual deadlocking where thread A is waiting for thread B to unset a lock, but thread B has already stopped waiting for thread A to unset a lock, or (2) recursive deadlocking where a thread has set a lock, but within a function tries to set the same lock, thus waiting for itself.

One optimization related to locks is the so-called "multi-reader lock" which splits "locking" into requesting read access and requesting write access. If the lock is currently under a read lock, any other threads are allowed to get a read lock, while being activated with a write lock prevents any other concurrent threads from locking. In case data are read often but written rarely, this can be a significant performance improvement. In our project, we will have dozens of tasks for each thread, and thus only a small percentage is updating at any one time, so this is very useful.

To avoid these problems of race conditions within the context of task parallelism, it is advantageous to consider all data to be "owned" by a specific task. Any task is then allowed only to update their own data and not anything else. Communication between tasks e.g. containing fluxes flowing from one part of the mesh to another is then handled indirectly by sending messages from one task to another. Any thread is then free to modify its own data, but if any of it is needed for other tasks, this update happens under the previously mentioned multi-reader lock.

For locking to work, it is important to consider what a data update is, and thus what should be the scope of a lock. For performance reasons locks should only be held for as short of a time as possible for correctness, but no shorter. If e.g. a hydrodynamics update has been run and multiple properties are being updated, it is not sufficient to lock and update one property at a time. As seen from another thread trying to read the data, this would manifest itself as the system being in an invalid mixture of the states from two different times. Thus it is important that all externally visible properties are in a collective "valid" state at any time the locks are not held. Much optimization work can be done to make this as granular as possible, eg. by preparing the new state outside of any locks, and then just swapping a pointer under the lock.

From a programmatic perspective, writing a thread-safe program is often quite error-prone and hard to debug since errors are non-deterministic and often only occur very rarely. To avoid errors, it is therefore often useful to use already implemented and tested data structures that handle data modification in a thread-safe manner. Many such algorithms might exist with different performance and complexity characteristics, but an important one for this project is the "multi-writer single-reader queue". This allows many threads to append data to a list and a single thread, the reader or consumer, to modify and delete this data. This queue is implemented as a list of elements - nodes - where each has a pointer to the next node (and optionally the previous if so-called a doubly linked list), as well as a pointer to the first element, "head". This is illustrated in 6. Traversal of the list is done by iteratively following pointers to the next elements. The advantage is that the addition and removal of elements is local, i.e. only requires modification of the current element and the pointers of the immediate neighbors, also shown in the figure.

In the multithreaded multi-writer single-reader scenario, multiple threads are allowed to append data to the end of the queue by simply adjusting the pointer of the last node, while locking the reader thread from deleting this node. The reader is then free to investigate and modify the data in the list at any time, even without a lock, since it is always in a valid state and other threads only depend on the linking pointers. The reader only has to acquire the lock for the deletion of nodes. By encapsulating these operations within a well-tested module, we can reduce the prevalence of threading errors and remove the complexity of the locking mechanism from the algorithm using it.

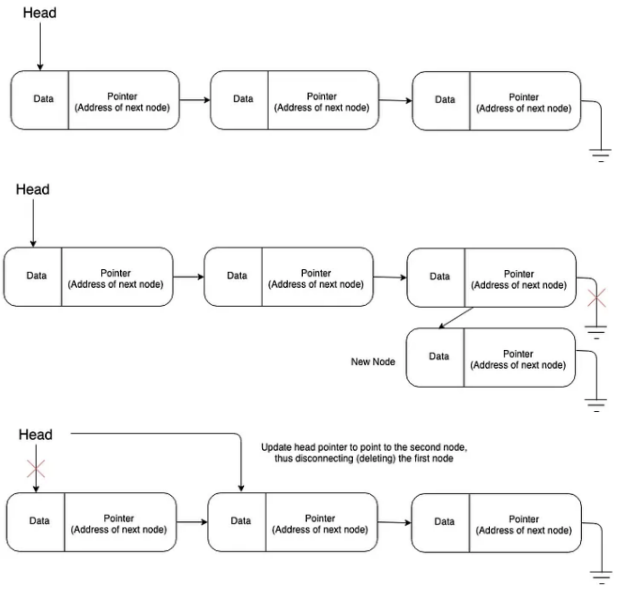


Figure 6: Illustration of the workings of a linked list source showing from the top (1) a list with 3 elements, (2) appending an element (3) removal of the first element. Source https://towardsdatascience.com/linked-lists-vs-arrays-78746f983267

This is just one example of a thread-safe structure, and many others may exist with various characteristics. For example to adapt the above structure to be a multi-writer multi-reader queue (i.e., more than one thread is allowed to modify and remove elements), one would additionally have to make use of locks during traversal and data modification since this might cause a read-after-write conflict. This is illustrative of the trade-off that is often present in thread-free data structures where the extra preconditions on the use (in this case, only having a single reader) allow a more efficient implementation with less locking.

4.3 Massively Parallel Computing

If we want even greater performance than what a single computer can offer and/or want to simulate a model too large to fit even on a large modern computer, huge supercomputers allow us far more memory and power than a single computer can offer. In order to use these, we need to be able to run our program on multiple processes (ranks) that are distributed across the many individual compute nodes that make up a supercomputer (often one rank per node). Just as OPENMP provides a framework for doing multithreading, the message-passing interfaceMPI is the standard used for node-to-node parallelization [30]. This might sound like a simple generalization from thread parallelization, but in practice, a number of complications exist when we try to perform massively parallel computing.

4.3.1 Global operations

While scaling is important for parallel computing within a node, scaling to 50-100 threads is usually sufficient to get good hardware utilization. For massively parallel computing, on the other hand, any algorithm has to be able to scale to up to thousands or millions of threads. This means that global operations such as synchronizing between time steps, agreeing on global step sizes and so on will have far larger impacts, and global operations should be exceedingly rare to get good scaling in this setup. MPI provides a number of these global operation primitives such as getting the minimum value of some variable across all ranks.

4.3.2 Shared vs distributed memory

While different threads on a node have different caches and benefit from working on their own part of the data, all threads still have access to the same view of the data. If one thread as part of the AMR refines part of the grid or steps part of the system forward, this update is immediately visible

everywhere. This is in contrast to distributed memory where each node has its own memory and updates have to be transferred from node to node over networking.

When using MPI, this message handing is performed completely manually by the program, thus we have to specify in our data both how to send and receive data. Using this interface, most computing clusters and supercomputers are able to efficiently transfer messages from any rank to any other rank as defined by the user. Much can be said about how the physical layout of inter-node networking (the topology) and how the various ways of sending or receiving messages inMPI have performance implications for the code, as well as how to avoid deadlocking for nodes waiting to receive messages from each other. For this project, though this is out of scope since it has not been explicitly applied, but keeping in mind the asynchronous and distributed nature can be important when interacting between tasks.

One important concern in distributed memory is to minimize the amount of information needed on each rank. A naive approach to AMR where every node has full knowledge of the AMR tree and broadcasts any changes e.g. when an MHD update is performed or a region is refined will quickly swamp the intra-node networking, require large overheads destroying scaling and limit the simulation size to be able to fit in the memory of each individual node. This is an extension of the practical restrictions on global operations, further highlighting the importance of local simulations.

4.3.3 Hybrid parallelization

On a supercomputer with thousands of nodes, each with hundreds of cores, one way to run anMPI application would be to initialize a rank per core. This might work, but the restrictions imposed by the distributed memory model and unnecessary messages to synchronize ranks residing on the same node would make this approach wasteful. A more optimal solution is therefore to have one (Or in specific cases e.g. related to scaling or non-uniform memory access between cores on the same rank, a few) MPI rank per node, with shared memory parallelization within the rank. This hybrid approach eliminates overhead, but as with all complexity, there may be a cost in code complexity and potential for bugs.

Typically it is also the case that for larger subproblems, communication requirements grow slower than the problem size. Take for example the case of mesh simulation where each rank handles a chunk of the domain. If the domain is properly split over the ranks, the volume of (and thus the cost of updating) the chunk handled by each rank goes with the side length cubed. We need to synchronize the data with other ranks only on the surface of the owned volume, the size of which only grows quadratic. Thus relativeMPI overhead decreases with larger problems on each rank, allowed by hybrid parallelization

4.3.4 Shadow copies

When we use hybrid parallelization, any piece of code depending on data from other tasks will have to find out whether the data for the task is on the same rank or a different rank and gather the data in the appropriate way, e.g. using reading under a lock or getting theMPI message. If we have to perform any time inter/extrapolation to perform updates, e.g. at the boundary between AMR refinement levels, we might also have to store previous messages. This can significantly complicate the code.

One solution to handling this complication is to handle it in one place using a technique known as shadow tasks. Here each rank stores not only the tasks owned by itself but also all tasks needed by all owned tasks. These tasks are identical to owned tasks when not worked on, but an update of these only involves gathering data from the owning rank. Thus when accessing other tasks we do not have to worry about where each task is located, minimizing code volume and the prevalence of bugs.

4.3.5 Load Balancing

When performing a simulation, we need to ensure that no rank falls drastically behind other tasks in its simulation. In case this happens, other ranks would have to wait to let it keep up. Ideally for a simulation where each part takes a constant amount of time, simply distributing the tasks properly at the start should ensure this. But for simulations where the runtime is data-dependent such as adaptive

mesh refinement or particles moving around, this might change over time. For this, adaptive load balancing is necessary.

Adaptive load balancing refers to the process where the tasks are moved around the nodes dynamically to keep the workload for all nodes constant and reduce or eliminate waiting. To do this, the code should be able to detect that a node is running behind, and then select an appropriate task to hand off to another node. Done badly, this might quickly result in tasks being scattered around the system, raising the amount of intra-node communication needed. To avoid this, the load-balancing algorithm needs to balance the goal of minimizing the "surface area" between tasks on different nodes, with quickly addressing nodes falling behind.

When a task is chosen to be moved to another rank, it needs to be transmitted. While a task may contain data spread across multiple arrays in the memory of the sending rank, all data needs to be packed together into a single array to be transmitted across the networking. This package is then unpacked on the receiving end. This serialization/deserialization is conveniently identical to the process of packing the data into a binary file to store on the disc for later data analysis. Notably, to maintain correctness, moving individual tasks typically does not only require that specific tasks be moved but also handles the change in the required shadow copies.

4.4 Good coding practices and object-oriented programming

Given the vast scale of codes required for cutting-edge physical codes simulating complex physics in an optimized, scaleable manner, the complexity and size of the code can be daunting. This is made even more true by the fact that for performance reasons compiled, low-level languages such as C, C++, or FORTRAN are often used for this, further increasing the amount of code necessary. On top of this, these codes are often developed over the years by teams of people with no detailed understanding of the code beyond the part they are working on.

Without any further care, a code will quickly devolve into so-called "spaghetti code" where the code has complex inter-dependencies all throughout, many pieces of the code modifying the same global data, "temporary" fixes, all making the code error-prone and less and less readable and modifiable over time. To avoid this, the code must be written according to fixed, well-known patterns agreed upon by all authors called the programming paradigm.

The probably most used paradigm in large codes is called Object Oriented Programming,. In this paradigm, the fundamental units in the code are Classes (e.g. "task") and instantiations of these, Objects (i.e. task #4). A "program" is then viewed as a number of these objects interacting with each other. Each object has its state, a number of properties (e.g. the data and simulation time), and actions (functions) that may modify these properties (e.g. "update"). These actions may then have dependencies on other objects, where invoking one action results in invoking other actions on other objects (e.g. one task asked whether it is ready to update may have to ask neighbor tasks about their state).

This paradigm thus offers good encapsulation of behavior and state, meaning that code and data related to the same thing are kept together in code, making it possible to quickly get an overview of the actions and dependencies of a piece of code, while modeling an intuitive "interacting objects" view that improve understanding.

Additionally, most modern languages have building support for object-oriented programming, making these concepts clear in the code.

4.4.1 Polymorphism

Going further, we often have multiple different sub-types, or derived types, of the "base" classes (e.g. "gas task" and "particles task" are both a type of task). These derived types then inherit the properties and actions defined for the more general type of object and are able to add new properties and actions

or even overload (redefine) the general ones (e.g. updating different tasks will probably be different function)[31].

Any piece of code using an instance of the base class should be able to use any of the derived types in its place (e.g. if a function accepts "task" objects, both "gas task" and "particles task" objects should work for that parameter). In the language, this is enabled by simply declaring a parameter or variable "polymorphic"[4], in which case an argument of any derived type is accepted. This means that when an action is invoked on this parameter, multiple different implementations may exist to implement this action (the different update implementations). To facilitate this, the language will check at runtime at every use of a polymorphic variable which function implementation should be used, and call this function. This dispatching to the correct function adds overhead to every function call, along with the uncertainty inhibiting the optimizing compiler. For this reason, programming languages often provide both polymorphic and non-polymorphic variables and pointers. In practice, this overhead is rarely significant, but polymorphic function calls should be avoided in performance-critical loops.

Deciding exactly which objects to define and what properties these should have is more an art than a science, but the two primary concerns are to minimize code replication while maximizing the readability of the code. A useful set of guidelines for the design of these classes are the SOLID principles [32]. For example, the Liskov substitution principle states that "Where a base class is used, one should be able to use a sub-class without any changes", which codifies the expected behavior assumed in the description of polymorphism above. This principle ensures that it is possible to add new physics to the existing code, e.g. extending a "star" class to a "supernova star" by deriving the star class without having to duplicate any code. Thus one can add new physics without having to duplicate any existing code and use the work already done.

In part to enforce this Liskov substitution principle, programming languages typically do not allow calling functions or addressing properties defined in a derived class on a variable defined as the base class. In some cases, this restriction is problematic (e.g. if we know the task is a gas task, it could be useful to read the density). For this, we have typecasting. In the same way that the dynamic function dispatch can check the object type and call the correct code, the user code may do the same thing. In this way, we can go from a base object to a specific derived object and use data and functions specific to this derived implementation. Casting in this direction is called down-casting. The opposite case of referring to an object according to one of its base classes is called up-casting.

4.4.2 Cyclic dependencies

In the example above, consider if we want to define an interaction between the different types of tasks. In this case, each type of task will have to know about all other types of tasks. The reusability is reduced since these two objects are now inseparably coupled together. From a readability perspective, cyclic dependencies drastically increase the possible dependencies between objects. Lastly from a very practical perspective, the coupling means that these objects have to be compiled together, making retesting slower.

[4] In FORTRAN by specifying parameters as CLASS

5 The Dispatch Framework

The DISPATCH Framework [1] is a developing computational framework for massively parallel, three-dimensional simulations of various astrophysical phenomena ranging from molecular clouds simulation to solar physics. It incorporates a range of advanced techniques to efficiently model complex astrophysical phenomena in an adaptable, object-oriented framework, making it an ideal tool for studying the process of planetary formation. Using DISPATCH, we were able to implement particle integration as part of a heterogeneous particle-mesh simulation.

This section provides an overview of the key features of the DISPATCH Framework relevant to this particle implementation and the constraints of this project.

5.1 Object hierarchy

The DISPATCH framework is an object-oriented framework for performing task-based multithreaded simulation. On a basic level, the framework handles pieces of work, objects of type TASK_T. This class is a definition of a common interface for all possible physical tasks. By calling functions defined in this common interface, the framework can instruct the task to perform various actions needed for the simulation, as well as to query for requirements of the task. A task therefore just has to "do as instructed", responding to the queries from the framework.

Before the task is instructed to perform an update, the framework has made several checks and prepared a list of all nearby (neighbor) tasks, the so-called nbor list. When selected for execution, the framework then instructs the task to update in a series of steps such as COURANT_CONDITION, PRE_UPDATE, and DELETE. We will mention the specific functions and their purpose when relevant.[5].

This framework supports a large number of possible configurations, but in this project, we are describing simulations of the interstellar medium around a forming protostar. In this setup the space is split into a large number of individual "patches", volumes of space, represented by the PATCH_T type, as well as several (proto-)stars represented by the SINK_T type. While sink tasks are relatively simple and isolated to one module, the patch tasks contain several modules that can individually be added or removed such as cooling, chemical networks, self-gravity, accelerated coordinate systems, and magnetohydrodynamic solvers. During each of the TASK_T calls, any number of these will also need to be called. The job of coordinating and selecting these physics modules resides with EXTRAS_T which is called whenever a patch is scheduled on a thread for task update. This is illustrated in figure 7.

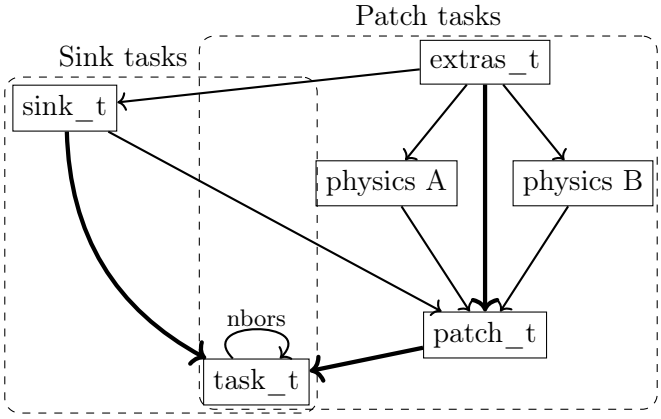


Figure 7: Simplified diagram of the classes of DISPATCH relevant for this thesis. Thick lines denote inheritance, e.g. SINK_T extends TASK_T. Thin lines denote dependency, e.g. EXTRAS_T imports SINK_T. Dashed boxes show task ownership, e.g. EXTRAS_T knows about the existence of sink types, but their data is not owned by the same task, so reading or modifying has to happen with care under a lock. Notably, this is just a tiny, simplified slice, but it is enough to understand this project. For instance a level above PATCH, the GPATCH_T will be neglected.

[5]To be precise, some of these calls are not done by the framework on the TASK_T type, but rather by the EXTRAS_T derived type SOLVER_T, which calls these functions.

5.2 Task Queue and Scheduling

On a fundamental level, the simulation loop of DISPATCH consists of several threads each grabbing a task from a task queue, performing this task, and then grabbing a new one. Each of these tasks represents a piece of work that needs to be done, which in our case will be updating the state of a patch of the simulation domain or updating a sink, but in general, could be any number of other physical processes. By carefully scheduling these tasks in an appropriate order, the scheduler ensures that the simulation progresses in a coordinated and efficient manner. Before any task is selected by a thread to be updated, it is first checked that they are ready to update, and e.g. their neighbors are not too far ahead. If this is not true for any tasks not currently updating, for example, if some tasks are much slower to advance than others, then many threads will have no work to do, bottlenecking the simulation as a whole.

To avoid a few tasks holding the whole simulation back and other threads wasting valuable computing time, there should preferably be 10-100 more tasks than threads at the highest resolution to guarantee that every thread always has something to work on. This means that at any one time by far most of the tasks are inactive, and so any temporary data should be removed from tasks between updates to avoid wasting memory.

An important part of the framework is handling their neighbor list (nbors). To perform the update on a task, it is often necessary to refer to the state of the surrounding local region, and for this, each task specifies the "size" of its influence. For patches, this would be their width plus the buffer to include the region necessary for their spatial derivatives (their "guard zones"), and for sinks the radius from which they are accreting. The framework then makes sure to give the task a list of references to all tasks within this distance (neighbors), allowing (relatively) easy implementation of communication on the part of the task.

Important here is data ownership. Every piece of physics data is owned by a specific task and is updated by that task, while it can only be read under locks by other tasks. The concept has already been presented extensively in Section 4.2.3, but the dashed boxes in figure 7 illustrate this concept in the specific case of our setup DISPATCH, illustrating the boxes of ownership. If a task leaves its domain, it has to take care to coordinate this with the (potentially active) owner task.

This neighbor referencing could be problematic if tasks are deleted while their neighbor is referencing them. For this reason, When a task is marked for deletion they are not immediately deleted. Instead, they are no longer added to any neighbor lists during task update preparation. By keeping track of the number of references to any task at any time, tasks are then only deleted when safe. This means that we do not have to worry about neighbor tasks disappearing at any time.

5.3 Independent Timestepping

In the DISPATCH Framework, each region of the simulation domain is advanced in time independently. This independent time-stepping means that e.g. high keplarian speeds, outflows, or shocks anywhere in the simulation will not require a complete slowdown of the global simulation elsewhere. This is in contrast to a constant sub-cycling factor where child patches are simply updated twice for every parent patch update, the approach used in [10]. This approach maximizes the efficiency of the simulation, ensuring that computational resources are not wasted on unnecessary updates. In addition, a possibly equally important advantage is the lack of global synchronization or communication needed for local time-steps, improving scaling.

On the other hand, these advantages do come at the cost of a more complex code. Since neighbor tasks can be either ahead or behind by any arbitrary amount, when values at a specific time are needed (e.g. for the spatial differentials in the solvers) it is necessary to interpolate in time. Patches handle this problem by padding their arrays on either side and filling them up with interpolated values at the start of the update. This is not free but has the advantage that all solvers can simply operate on standard contiguous arrays of data and not do special treatment of the boundaries.

In the case of e.g. hydrodynamics, independent time stepping can result in two neighbors disagreeing about the fluxes between them. To solve this, each task can add up this difference and compensate for the difference later.

Another issue is that patches may end up ahead of a neighbor which is currently propagating a supersonic shock across the mutual boundary. Not capturing this would quickly destroy the shock. To avoid this, the situation is detected and the patch update is rollen back the previous step and tried again with a smaller step using the new boundary values. This is amusingly called a Mulligan, which is a friendly expression used in recreational golf when a player makes a bad shot and tries again without getting marked down for it.

These are examples of the care that must be taken at the interactions between patches due to local time-steps, but in some cases this results in a 30 times speedup [1] so is well worth it for a code made to have cutting edge performance.

5.4 Paralellization and load balancing

DISPATCH is made to scale to arbitrarily big systems with perfect scaling demonstrated up to systems with 100.000 cores [1]. This is possible using the techniques described in section 4.3, i.e. by using local operations and even local time steps, parallelized using a hybrid parallel approach with OPENMP to parallelize on individual nodes and MPI to parallelize across ranks, and dynamically load balancing across ranks with shadow tasks enabling neighbor accesses across patches.

In DISPATCH, this is all handled "behind the scenes" by the framework, significantly easing the work when implementing a physics module. only the serialization methods for transferring tasks across nodes and serialization of the updates of shadow tasks need to be user-defined.

5.5 Patch-based Adaptive Mesh Refinement

The DISPATCH Framework uses the patch-based adaptive mesh refinement (AMR) technique presented previously. This allows extremely high per-cell-update performance (a few micro-seconds each) and means that all bookkeeping and the function call overhead is amortized over in our case 16x16x16=4096 cell updates. This number is a trade-off between being able to focus the resolution where it is needed (smaller patches are better) and being able to amortize the constant costs (larger is better). Any module should be patch size-agnostic such that this can be adjusted according to the specific physical system simulated.

This adaptive mesh refinement works by allowing each of the eight corners of the patch to create a "child" patch if any number of refinement conditions are met. This means that a patch can have anywhere between 0 and 26 neighbors at the same level ("siblings") (3x3x3-1) and 0-64 children. If a patch is lacking any of its siblings, then the guard zones where this sibling would have been is filled up using data from one of the parent patches. This would make it very complicated if the parent patches were also only partially filled, and so DISPATCH guarantees that any patch will always be completely "supported" by a full set of parents. These guard zones are then striped again just before the end uf the patch update to avoid wasting memory: With a required guard zone width of e.g. 3 cells in either direction, the data including guard zones take up $(3 + 16 + 3)^3/16^3 = 2.6$ times as much memory as the patch without.

When we talk about a patch, the region that is covered by it is called the region of authority (ROA), while the region covered by its guard zones AND itself is called the region of interest (ROI). This nomenclature will become convenient.

The restrict procedure is a key part of the DISPATCH Framework's AMR system. Where a patch has children, those children provide a higher-resolution version of the same physical system. By averaging the eight cells in the child patch, the restrict procedure ensures that the state at the lower resolution region is accurately updated, maintaining the overall accuracy of the simulation.

5.6 Interaction between components

While the nbor list makes referencing neighbor tasks easy, different physics modules on the same patch cannot call each other. This is intended since these types of interactions would create cyclic dependencies, make the code very modular, and generally be a mess. But coming down from the high horse of proper coding practices, how can e.g. the gravitational solver communicate its calculated gravitational acceleration to the gas integrator?
For this communication between modules, the information has to be communicated in a commonly understood format, in this case, a float array containing the gravitational acceleration in every cell, with agreement about whether this, includes guard cells or not. When transmitting the data, we have two approaches illustrated in figure 8:

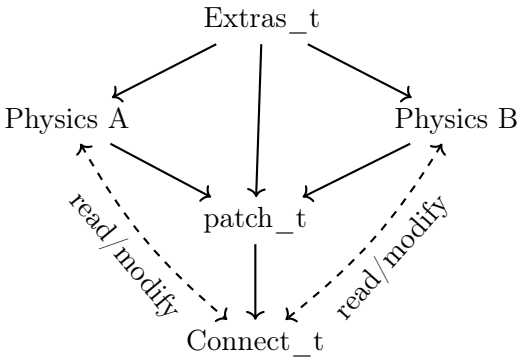


Figure 8: Illustration of how two physics modules A and B can communicate without knowing about each other through a common interface

- We can "go up" communicating through the EXTRAS_T module. In practice this works by having the functions called by EXTRAS_T in e.g. PRE_UPDATE return the relevant arrays, and then passing it on to PRE_UPDATE. This makes the dependence between the modules clear at the cost of making the EXTRAS_T module more complex.

- Alternatively we can "go down" through the PATCH_T module. By having one module write to a field in the PATCH_T object, other modules can later access and read this information. This makes it simpler for the user to add the module, but can be dangerous if the module writing data and the module reading it both perform this work e.g. the PRE_UPDATE function since the dependency is hidden. In this case, simply doing an innocently looking swap of two lines can make the code silently fail. In the case of e.g. gravity, this is worth it, since many different physical modules may be interested in adding sources of acceleration or reading the gravitational acceleration. Adding these up manually in the EXTRAS_T module would be a hassle.

  To allow custom modules to add data to the PATCH_T class without actually modifying the class directly, one of the properties of PATCH_T are CONNECT_T which is simply a data member with no functionality and a handful of members defined for this purpose. This is also where the above-mentioned gravitational acceleration is communicated.

5.7 Initialization and Zoom in simulations

To simulate a system, we need to start somewhere. This can be some generated initial states such as some turbulent box of gas and dust, but getting from this state to a system of interest (a collapsing prestellar core in this case) can require an extreme amount of simulation time. This is especially problematic since it is unknown a priori where in this model the phenomena will occur, so high resolution will have to be used everywhere. To speed this up, we use a technique called zoom-in simulations, first presented in [33]. With this technique, a large supercomputer simulation of a Giant Molecular Cloud has been run, which spontaneously creates proto-stars of many types. In this simulation, a large number of individual star systems spontaneously form. We then choose one of the formed stars and select the data of this system just before collapse. These are now ideal initial conditions for simulating the collapse and early history of this protostar. We then rerun a new simulation with resolution centered at this protostar and the bulk movement of the core removed with a Galilean transformation. Thus we can efficiently focus resolution on the specific phenomena of interest. Notably, we do not completely remove the data far away from the sink, but simply degrade the resolution. This allows the infalling gas to form realistic boundary conditions for the sink evolution.

5.8 I/O

When running a zoom-in simulation, the starting point is getting the data into the simulation, and similarly, the endpoint will hopefully be a nice plot in a thesis or paper.

To perform this input and output (I/O), we have to be able to map the storage of the data in the fields of separate task objects, thousands or millions of separate places in memory, to a format of one or multiple data files stored on disk. This mapping is called serialization and deserialization and is similar to what is required when transmitting tasks over MPI.

With our setup, every task is asked at a given cadence, e.g. every 20 years of simulation time, to serialize their data binary data stream, which is then written serially to the disk, with some metadata written to help locate the section corresponding to any given task. If these files are to be used for later restarting, a bunch of data known as the header containing all bunch of metadata is usually written to the start of this stream to reconstruct it precisely.

It is important to consider the volumes of data this can produce: with a simulation using e.g. 15 GB of RAM, that will also result in every single data output being at the same size, which can quickly produce impractical amounts of data and use significant time writing to file

When simulating on a supercomputer, in case the code crashes it is also very practical to be able to restart the simulation from the point at which it stopped. If such a crash occurs after a million CPU hours, one will be very happy if it is possible to simply up a few tolerances or decrease the max allowed courant number and reload the last produced output.

6 Implementing particles

With a solid understanding of the physics we attempt to model, the relevant numerical techniques, computational concerns, and existing framework constraints, this chapter will present the particle integrator component created in this project. Each subsection will present the solution to a given subset of the problem.

In broad lines, we have chosen a solver that mirrors how gas and potentials are handled by having a separate instance of a particle solver handle all particles within a given patch and updating these in lockstep with the gas solver. This simplifies some aspects, like the fact that a thread owning the particles can also safely access and modify the gas properties. We will in this section describe how the implementation solves each of the many sub-problems for such an implementation, namely:

- The overacrching structure of the code implementation 6.1
- Updating the particles one time-step, under the influence of self-gravity and gas drag (6.2)
- Initializing the particle population when a patch is created, either when loaded or when the patch is refined (Sec 6.4)
- Performing proper two-way interaction between particles and other physics modules 6.5
- Accreting particles (Sec 6.6)
- Making sure that the particle distributions are smoothly represented everywhere (Sec 6.7)
- How to represent particles in a manner consistent with the AMR hierachy (Sec 6.8)
- Transferring particles between patches (Sec 6.9)
- How to handle the removal of patches (Sec 6.10)
- Efficiently representing the data with an eye to precision and performance (Sec 6.11)
- Getting data in and out to be able to restart the simulation and analyze the data (Sec 6.12)

Of note, before this project, a particle module was implemented for the article [34]. This particle module has been used as a rough template for the work done in this project, but an almost complete rewrite has been done to significantly simplify the code and use of the module, as well as to much better handle the interaction with adaptive mesh refinement as well as some other improvements. The only remainging code are the difinition for how particles are described in memory, some of the assosiated memory management (allocation, deletion), the time-step algorithm and interaction with self-gravity as well as some error-checking functions, although no functions have been untouched and no code has simply been copied untouched. At the end of the description of each sub-module, we will properly attribute the work.[6]

6.1 Code structure

The implementation of a particle module will require a large amount of code and hopefully be used for many years, and thus code quality is important. Therefore, even though this is a physics master, we will describe the overarching architecture in this section. The code is designed and written with the following constraints in mind:

- Logical code: The code should be structured in a sensical way, with as few internal cross-dependencies and "smart tricks" as possible to make it easy to read and modify. To achieve this on a design level, we split the code into two classes:

[6] Readers note: The textual description is admittedly in some places quite dense, but this describes in detail the working principles and considerations behind around 3000 lines of complex, carefully written code. Much about the implementation has been simplified and glossed over in this description.

- PARTICLES_T is a container for storing a collection of particles. These particles can be inside of a patch, on their way from one patch to another. Being a container for particles, this is where arrays for storing particles are allocated and deallocated and can serialize and deserialize the data. It has several debug functions to make simple sanity checks for the validity of the data. Lastly, it currently supports the physics functionality, i.e. functionality for initializing the particles, renormalizing them, and taking a timestep.
- PARTICLE_SOLVER_T is the solver exposing all particle functionality to the EXTRAS_T module such as input/output, calling the appropriate update functions at the correct times in the dispatch update order. This also involves handling particles moving from one patch to another which is far from trivial.

The physics-related functions of the PARTICLES_T class should probably be moved either to a separate module exclusively handling the physics or into the PARTICLE_SOLVER_T, and are only in the PARTICLES_T class for legacy reasons [7].

On a broader level, we have sought to follow standard coding principles of meaningful variable naming and code commenting, single-responsibility functions, etc.

- Ease-of-use: Adding particle integration to an existing or new experiment should be as simple as possible. This involves not locking into physics which may not be appropriate in other experiments, such as stars (sinks), magnetism, or gravity, but also being open to possibly adding other physics like accelerated coordinate systems. Locking into such a physics module would also prevent the use of any other module handling the specific type of physics, like a different gravitational solver.

  Most of the particle module exists completely separately from the rest of the code and other implemented physics and simply operates on the particles within a patch, but several exceptions exist that must be handled. For example, the effect of their drag and gravity needs to be properly communicated with other physical modules. These will be handled through the solutions presented in 5.6.

- The structure should allow a maximally efficient implementation. This does not mean that this specific implementation should be optimal (surely it is not) but simply that data structures should be designed to allow efficient processing by storing data in flat arrays with an efficient representation, and the architecture should minimize the number of necessary data copies and ensure e.g. not to make polymorphic dynamic function calls in performance-critical loops.

- It should be possible to further adapt the code, e.g. for accelerating coordinate systems, different integration schemes, or adding charged particles.

  In practice to do this, one should make a new type inheriting the particle container or the particle solver, or possibly both. Then any functionality can be added by overloading the relevant function(s), possibly subsequently calling the "standard" implementation. As previously discussed, this requires using exclusively polymorphic type pointers and not relying on the exact type.

6.2 Particle Timestep Procedure

For each patch update, in the same way that the gas properties are updated, so should the particle positions and velocities. For this we will use a slightly adapted version of the implicit integrator presented in [34][8]. The fundamental requirements of the integrator are as follows:

- Speed. We would like to have 10s of particles per cell. With hundreds of thousands of cell updates per second per core, this easily translates to hundreds of millions of particle updates per second.
- Stability of long time-steps compared to stopping time to handle light particles and high gas densities. Thus we need an implicit integrator.

[7] I.e. "don't fix it if it ain't broken"

[8] Same recipe except for first order Euler instead of leapfrog

- Correctness for massive particles. When particles break under friction, they impart momentum to the gas, accelerating the gas.

The acceleration of a neutral subsonic dust particle of radius s, mass m, position $\vec{p}$ and velocity $\vec{v}$ under the influence of Epstein drag through gas with density ρ and gas velocity $\vec{u}$ and thermal speed u_{th} and gravitational potential ϕ can be written as

$$m\frac{d\vec{v}}{dt} = -m\nabla\phi + \frac{4}{3}\pi\rho s^2 u_{th}(\vec{u}-\vec{v}) \tag{34}$$

$$= m\vec{a_g} + \underbrace{\sqrt{\frac{\pi}{8}}\frac{1}{\rho_d}}_{c_d}\frac{m\rho c_s}{s}(\vec{u}-\vec{v}) \tag{35}$$

$$\implies \frac{d\vec{v}}{dt} = \vec{a_g} + \underbrace{c_d\frac{\rho c_s}{s}}_{p_c}(\vec{u}-\vec{v}) \tag{36}$$

$$\tag{37}$$

Where we defined the gravitational acceleration $\vec{a_g}$ used the dust solid density ρ_d together with a spherical assumption on the dust and sound speed $v_{th} = \sqrt{\frac{\pi}{8}}c_S$ to define the drag coefficient c_d and the inverse stopping time $p_c = 1/t_s$. Similarly, the position update is simply per definition

$$\frac{d\vec{p_d}}{dt} = \vec{v} \tag{38}$$

Instead of integrating individual particles, our scheme of meta-particles requires integrating N particles at a time. Conveniently, the above equation directly applies to these meta-particles, which we will for sanity's sake simply call "particles" from now on.

Since we are integrating massive particles, these impart momentum to the gas. For a particle interacting under drag with a gas with volume V and density ρ_g, this can be written as

$$V\rho_g\frac{d\vec{u}}{dt} - mN\frac{d\vec{v}}{dt} = 0 \tag{39}$$

To remove references to the volume, it is convenient to parameterize the dust mass per metaparticle using the total mass per cell volume $w = \frac{mN}{V}$, changing the above equation to

$$\frac{d\vec{u}}{dt} = \frac{w}{\rho_g}\frac{d\vec{v}}{dt} \tag{40}$$

Equipped with these updated equations, we would like to discretize them. As previously discussed, such a discretization of a linear system can happen in many ways. In this case of a particle solver in lockstep with the gas solver, the timestep needs to be on order $1/5$ of the time taken to cross the cell, i.e. very short in comparison with eg the orbital timescale. Furthermore, for the hydrodynamical simulator to be accurate, multiple grid points need to resolve any structure of interest, leading to relatively low gradients on the scale of a fraction of a cell. On the other hand, particles may be arbitrarily tightly bound to the gas with potentially stopping times orders of magnitudes shorter than the time step.

Due to these factors, we estimate that a high-order discretization in both time and space is of low importance, but an implicit solver is of high importance. We, therefore, choose a simple implicit Euler scheme, with gas properties read and written to the nearest cell center. In this scheme, the two above equations are parameterized as follows, with values indexed with 0 at the current time t and 1 for the value after the timestep at time $t + \Delta t$:

$$\vec{\Delta v} = \Delta t \vec{a_g^1} + \Delta t p_c^1 \left(\vec{u^1} - \vec{v^1} \right) \tag{41}$$

$$= \Delta t \vec{a_g} + \Delta t p_c \left(-\vec{\Delta v} + \vec{\Delta u} + \vec{u^0} - \vec{v^0} \right) \tag{42}$$

$$\Delta \vec{p_d} = \Delta t \vec{v^1} \tag{43}$$

$$\vec{\Delta u} = -\frac{w}{\rho_g} \vec{\Delta v} \tag{44}$$

We removed the indexes on the gas quantities and gravitational accretion, since those are updated independently and for this purpose assumed constant. Solving the unknowns $\vec{\Delta v}$ and $\vec{\Delta u}$ yields:

$$\vec{\Delta v} = \Delta t \frac{\left(\vec{a_g} + p_c (\vec{u^0} - \vec{v^0}) \right)}{1 + \Delta t p_c (1 + w/\rho_g)} \tag{45}$$

$$\vec{\Delta u} = -\frac{w}{\rho_g} \vec{\Delta v} \tag{46}$$

Conveniently the solution is thus rather simple, and we avoid having to use an iterative solver for the dust evolution. A few limiting behaviors of this solver are interesting:

- For particles with very little mass $w \ll \rho_g$, the gas remains unchanged as expected, while particle velocity update reduces to the Euler implicit solver without gas feedback $\vec{\Delta v} = \Delta t \frac{\left(\vec{a_g} + p_c (\vec{u^0} - \vec{v^0}) \right)}{1 + \Delta t p_c}$
- For loosely coupled particles $\Delta t p_c \ll 1$, the denominator reduces to 1, and the update reduces to a simple linearization of the differential equation
- For tightly coupled particles $\Delta t p_c \gg 1$ and negligible gravity compared to drag, the particle update reduces to $\vec{\Delta v} = \frac{\vec{u^0} - \vec{v^0}}{1 + w/\rho_g}$ which for very light particles simply corresponds an instant velocity change to the gas velocity, while any higher velocity corresponds to inelastic collision.

For the Stokes regime where the gas density is extremely low, these equations do not reflect to quadratic temperature invariant drag force of Stokes drag, and so will yield wrong trajectories in these domains. With our interest mainly focused on the inner disk, this has little consequence, and having a more complex dependency of the drag on the velocity would significantly complicate the derivation of the drag.

Similarly properly implementing mach number correction to the Epstein drag in a self-consistent implicit manner would complicate the derivation since the Mach number is velocity-dependent. One could assume the Mach number to be constant and simply consider this part of the (constant) stopping frequency p_c, but validating this is left for further work.

The original solver described in [34] suggests a kick-drift-kick symplectic integrator, which has the advantage of conserving the energy of orbital bodies up to numerical accuracy, but due to the multitude of energy loss mechanisms available in Dispatch such as radiation transfer, cooling, chemical networks, etc, energy conservation is not expected anyway. Additionally with the tiny timesteps the error from true orbital trajectory for uncoupled particles is all but negible anyway. Therefore the speed of individual updates has been prioritized, leaving us with this simple analytical prescription for the time step operator.

It is worth commenting on how this plays out with multiple dust particles: While the above derivation is correct for one metaparticle interacting with one gas cell, in the simulation dozens of particles will interact with each gas cell. This means that even if the individual interactions are physical, an optimized implementation where all particles have their drag calculated in parallel and then afterward deposit their momentum into the gas might result in unphysical results, such as the gas ending up

moving faster than the initial velocity of any individual particle. In theory, the solution is to have the algorithm identify all N particles in the same cell, and then simultaneously solve the $N+1$-dimensional system of equations consisting of the momentum equation and the drag equations of each of the N particles. Instead of this quite expensive "true" solution, we instead approximate it by considering each collision to happen independently in an arbitrary order. Each particle then interacts with the combined result of all previous interactions.

Physically, this drag has another effect on the gas: The drag transforms kinetic or gravitational energy into internal energy per unit volume (u) in the gas as

$$\Delta u = w\Delta\phi + \frac{1}{2}w((v_p^0)^2 - (v_p^1)^2) + \frac{1}{2}w((v_g^0)^2 - (v_g^1)^2) \tag{47}$$

. This will then heat it up. Due to concerns about causing numerical instabilities in the gas solver due to extreme heating, we have chosen for this project to neglect this factor and let that be up for further work. This can physically be interpreted e.g. as assuming that the energy put into the gas is quickly radiated away. If one feels particularly pedantic, one might notice that this change in internal energy during a time-step will change the stopping time, meaning that a fully implicit solver should take this into the case. In practice though level of accuracy is completely swamped by regular discretization error sources in all relevant situations.

The probably most important assumption made in this derivation is the spatial gradients of gravity and the gas being negligible over the length of a timestep. This is in most cases true, but experimentally we have found that within a couple of cells of the central core, the gravitational gradients are very large. This means that in a single step, the velocity of the particle changes by an order of magnitude, making the particle instantly move dozens of cells. This is a consequence of a non-resolved feature, so we will not attempt to physically capture the true sub-step dynamics. Rather we simply cap the maximal step to one cell, by changing the update equation for the position to

$$\Delta\vec{p_d} = \min(\max(\Delta\vec{v^1}, -1), 1) \tag{48}$$

in units of cells, a process known as "clamping" or "clipping". This should be a very rare save-guard, owing to its nonphysical nature, and the Courant condition discussed next ensures that to be true.

The integrator described in this chapter was already implemented, but the changed data structure necessitated syntactic-level changes. Additionally, during the validation of the algorithm, the above-mentioned race condition was found and fixed, and the clamping of the step was implemented.

6.3 Courant condition

Without the dust integration, the maximum timestep of a patch update is set according to the propagation speed in the gas, which is a combination of advection, pressure waves, and Alfven waves. With the addition of particles, there is now the possibility of these to move at different speeds. This is mainly the case in the inner disk, where big particles rotate with Keplerian speed, while the gas rotates sub-keplarian due to the pressure support. Experimentally, we found during the implementation that not taking this into account would often result in particles moving a handful of cells or more in a single update.

The first function call that the framework does to the extras module exists exactly for this purpose, letting sub-modules adjust the courant number. Therefore when this is called, we set the max wave speed to the maximal particle speed in any axis.

This works, but particle solvers are in general a lot more stable than gas solvers, so the Courant multiplier to get the timestep is excessively conservative. This is especially bad since the inner disk where particle speeds dominate is exactly the region that is already most expensive due to short timesteps and high resolution. To remedy this, we have introduced a COURANT_MULTIPLIER (as of now, set to 2.5) which step speed is divided by. For a typical MHD solver courant condition of 0.2 (cells per update), this means that particles are allowed to move at most 0.5 cells per update.

6.4 Initial distributions

To trace the particles, they must be created. In the ISM simulation from which we start this zoom-in simulation, there are no dust particles, so we need to initialize a population according to the gas density. For this purpose, we sample in each cell several meta-particles intending to accurately capture the dust population across different dust sizes. Importantly this means that the number density of meta-particles should be constant and not represent the actual dust density, which is represented by a "weight" parameter w.

To this end, the dust initialization procedure has five parameters: number of particles per cell n_{cell}, minimum $s_{\min}$ and maximum $s_{\max}$ dust size, probability density exponent α_{dust} and dust-to-gas ratio $r_{d/g}$. The procedure then samples in each cell n_{cell} particles, each with the following properties:

- Their size are drawn randomly independently such that the logarithm of their size is uniformly distributed between the min and max $ln(s) \sim U(\ln(s_{\min}), \ln(s_{\min})$. This ensures that we sample the full-size spectrum under study.
- To get a uniform initial position distribution, we randomly "jiggle" the position of each within their cell
- Their total weight w is drawn from $w(s) \sim n_d(s)m_d(s)ds \propto s^\alpha s^3 s = s^{\alpha+4}$ where n_d is the number density of particles at any given size assumed to be a power law distribution, m_d is the weight of each and ds is the particle sizes covered by each metaparticle. In our simulations, we will assume $\alpha = -3.5$, corresponding to the canonical MRN dust distribution usually assumed for the ISM [35]. To ensure the dust particles in total have the correct mass irrespective of sampling, their masses are renormalized for each cell.
- Their velocity is initially assumed to simply be the same as the gas.

The addition of these particles to the system will add momentum, energy, and mass. Besides the obvious problem of physically simulating a slightly different system than the input, there can also be numerical stability problems, since this will materialize as a significant change to the gravitational potential. Therefore the mass added as dust is removed from the gas, along with the energy and momentum to not modify the velocity and temperature of the gas.

This method already existed and has only been syntactically modified, as well as the removal of the corresponding mass, pressure, and internal energy from the gas.

6.5 Integration with Self-Gravity and Gas Backaction

For gas-only simulation, the self-gravitational solver simply reads the gas mass and uses this for the Laplace solver. In this project, we add a new type of mass which is represented separately. To not rewrite the Laplace solver, we intercept the call to it and deposit the dust mass in the corresponding cells. This mass is then removed again after the call, allowing for example the hydrodynamics solver to work on the correct data.

The $\nabla\phi$ calculated by the self-gravity solver is then evaluated at the positions of all particles, yielding an "external" acceleration for every particle to be used in the solver. This has the advantage that other types of external acceleration such as the fictitious forces from an accelerated coordinate system can be added trivially on top. Similarly, the absence of gravity can simply be accommodated by decoupling the particle solver from the gravitational solver.

For the drag interaction, this might seem trivial since the particle time-step procedure already adds momentum to the gas. In this case, though, some hydrodynamical solvers may have a prediction step where knowing the force before applying it improves numerical accuracy. For this reason, we calculate for each cell the implied force per volume V

$$\frac{F}{V} = \frac{\rho_g * \Delta v}{\Delta t}$$

And communicate this to the MHD module through the CONNECT_T based approach presented in section 5.6[9]

The core of the physical interaction was already present but has been almost completely syntactically rewritten, with several user-level simplifications.

6.6 Accretion

Accretion is the process where gas and dust are moved from the circumstellar medium to the sink itself. Physically, this happens when dust and gas fall on the surface of the star, but by its very nature a sink is not resolved by the grid and thus neither is the infall.

Rather, the sink module looks at the cells in its immediate vicinity and uses a numerical prescription to estimate how much of the gas in each cell would be accreted onto the star in the given timestep, based on eg. the gas temperature, rotational vs gravitational energy, etc. To avoid discontinuities in the density, this prescription is written to smooth the accretion across several cells across, 8 by default. When the accretion ratio δ of each cell has been calculated, this ratio of the mass, momentum, and internal energy in each cell is removed and deposited onto the sink. This same accretion ratio we would like to use for particle accretion.

Unfortunately, the accretion process is currently done on the task of the sink, locking the patches from which gas is accreted. For this reason, we cannot follow the same prescription, since this would require modifying the sink module and coupling it to the particle module.

Accretion is thus another interesting case of interaction with another module, this time even more complex since the other module is running on another task, possibly concurrently.

Due to time constraints, we have chosen to solve a slightly simpler problem, that of only removing the dust particles, leaving the problem of depositing back the accreted mass and momentum for later.

For this, we have chosen an approach using the CONNECT_T module, adding an ACCRETION_EFFICIENCY field. When the sink module accretes from a patch, it accumulates the previous and new delta to this field $\delta_{\text{new}} = 1 - (1 - \delta_0)(1 - \delta_1)$ (which simplifies to simply δ_1 if $\delta_0 = 0$).

When the patch performs its update, this ACCRETION_EFFICIENCY is extracted and the field is cleared. This accretion efficiently, possibly accumulated over multiple sink updates, is then given to the particle solver. The particle solver then for every particle extracts the accretion efficiency at the relevant coordinate and then removes the particle according to this probability.

To ease the future implementation of a procedure depositing the mass and momentum back to the star, the particle simulator accumulates the removed mass and momentum and returns it to the extras module. Thus in theory no further modifications to this module should be necessary to support proper accretion.

To avoid any multi-threading conflicts between these two tasks, the modifications to the ACCRETION_EFFICIENCY field happen under the patch write lock.

Is all of this necessary? why not just let particles build up near the central core? In the course of developing the particle integrator, the accretion module was the very last one added, and it has allowed us to run about an order of magnitude faster simulations. This is because previously huge amounts of tracer particles accumulated in the very central patch, making updating this extremely expensive. Since the rest of the simulation has to wait, this effectively bottlenecks the whole simulation.

The previous code had no notion of accretion to a sink, so this is completely written from scratch.

[9] Technically, here we use a module called SCRATCH_T which has a slight optimization to avoid allocations, but the behavior is identical.

6.7 Normalization and tracer particles

Although the particle integrator has been derived to implicitly solve the update equation for the particle and gas velocity simultaneously, we still do not want the code to ever be in a position where only very few or even none exist in a given cell. This can cause problems due to the approximate handing of multiple particles interacting with the gas, the discrete representation of the particle distributions as well as cause problems for the hydrodynamics due to the discontinuous force on the gas.

The opposite situation where particles accumulate in a region is also problematic since this will hurt load balancing by increasing the computational workload for some patches relative to the rest. This can lead to a situation where a few patches are bottlenecking the integration of the whole system. On the other hand, this does not reduce accuracy or lead to numerical instability.

To avoid these situations, we regularly renormalize our distribution of particles in the following two ways:

- PRUNE is a procedure for merging similar particles. Here "similar" is meant as particles that are of the same size, move in the same direction, and ideally contain as little weight as possible. It is important to note that this will inevitably result in numerical diffusion due to the loss of information in the averaging process here. The exact process of identifying such a pair to merge is more of an art than a science and needs to reflect which physical aspects are most important to capture. Ideally, this might for example be formulated using the Stokes number or the velocity relative to the gas velocity. In practice, an implementation has to weigh performance (which is very important given the quadratic nature of finding pairs), responsiveness to a rise in the number of particles, and the accepted accuracy of a particle merge.

  In the implementation made for this project, an algorithm was used which selects pairs of particles that fulfill the following criteria: (1) They exist in the same cell. (2) At least one of them represents a mass lower than $\frac{m_{\text{dust}}}{2n_{\text{cell}}}$ where m_{dust} is the total dust mass in the cell and n_{cell} was the desired number of particles per cell set for initialization. (3) Their velocity difference is at most $|\vec{v_i} - \vec{v_j}|^2 < 0.2|\vec{v_i}|^2 + |\vec{v_j}|^2$. This is quite crude and works poorly with multiple particle sizes, but for this project, it serves its purpose of load balancing.

  When two particles are identified, the process is much more straightforward: One of them is removed, while the other is assigned the sum of their masses $w_{\text{new}} = w_i + w_j$ and the sum of their momentum $\vec{v_{new}} = \frac{\vec{v_i} w_i + \vec{v_j} w_j}{w_i w_j}$. The position we leave unchanged for simplicity, since we do not expect any sub-cell accuracy anyway. Like at initialization, there is also the question of internal energy, but for this project, we will consider this negligible.

- RENORMALIZE is a procedure to split particles with too much weight. The ideal split function is the inverse of PRUNE, but of course, such a function does not exist, since information is lost on averaging. Instead, we approximate it. For this project, we chose a very simple implementation where particles with more weight than e.g. 30% of the total dust in a cell are split into many particles, such that they on average represent about $1/n_{\text{cell}}$ of the dust mass in the cell. These split particles are then randomly assigned a position within half a cell distance, and the mass is split across them. We have chosen not to add any velocity or particle size perturbation.

One important problem with these re-normalization procedures (specifically the lossy TEXT_SC procedure) is the inherent loss of history: When we merge particles, we lose the ability to say "These particles originated at location x". For this reason, we have chosen to have a special class of particles called "tracers". These are particles that are excepted from all normalization rules. Furthermore, we have added a "particle id" to all particles, so it is possible to either from output files or "online" in the simulation to track these. Since the lack of normalization means that these can significantly bunch up, the number of these tracers should be restricted e.g. to 100 per patch initially.

The implementation was made during this project, but the coding itself was done by my co-supervisor Åke Nordlund based on mutual discussions of the working principles.

6.8 AMR

As argued multiple times, any simulation of the extreme variations of temporal and spatial scales present in star formation all but necessitates adaptive mesh refinement. Handing this aspect correctly is thus of uttermost importance. If any particles exist in regions, not at the highest resolution, how do these particles relate to the particles at higher resolutions, and how do we handle the destruction and creation of patches?

For this description, let us assume just two levels of patches, denoted as the "low" and "high" resolution areas, with the high-resolution patches considered to be the children of the low-resolution patches. Each region of interest here will then be either covered by both levels or just in low resolution, separated by a "high-resolution boundary". This description is without loss of generality since they are applied recursively throughout the AMR tree, and patch support rules guarantee that the low-resolution region will cover both sides of the high-resolution boundary. It is useful to keep in mind that the primary behavior for both particles and gas is simply advection: Particles are flowing in from the downwind direction and flowing out to the upwind direction. We thus need to be able to transport particles moving in from a low-resolution area, into a high-resolution area and out of it again.

We have chosen an approach that mirrors the handling of the gas: Like with the gas, the particle distribution at high resolution is considered the "ground truth", and the parent patch simply contains a coarse version of this distribution which it then uses for purposes of back-reaction through gas drag and gravitational self-gravity. The accuracy of this is of low importance since any errors will simply be overwritten at the next restriction, but the particle density is important to get correct, at least in the mean, since otherwise, patches at different levels will be attempting to solve completely different Poisson problems.

For this project, this "restriction" procedure has been written in the simplest possible way, simply calculating the average velocity, spread in velocity, and average dust density for both the cells of the parent and child and correcting the parent distribution accordingly. This process involves two different patches, so to safely read from the child patches we need to lock it from being modified from other threads, i.e. activating their read lock. Unfortunately, we did not have time to properly activate and validate this module.

This viewpoint of particles in low-resolution patches representing a coarser version of the same distribution makes the handling of particles exiting a high-resolution area trivial: Through the restriction, they have already given all information possible to the parent patch particle distribution, and so they are simply removed with the parent patch particles carrying their legacy. Similarly, if the AMR chooses to remove a patch, then these particles can be removed without a problem.

Having handled the case of particles moving from a high-resolution area to a low-resolution area, we shifted the focus to the opposite case. Fundamentally this process of refining the data will be imperfect, and if we have any hope of accuracy gains, the particle will have to move around in this high-resolution area for a long time. Since we consider the highest resolution patch at any given point as the ground truth, these low-resolution dust particles are thus considered to provide the boundary conditions needed for the high-resolution regions where needed. This is true both in the case of the downwind and in case of mesh refinement creating a new patch to be filled with particles, and we will call this process exporting particles down the mesh hierarchy. This process is illustrated in figure 9.

Lastly, let's consider tracers. On exiting and entering high-resolution areas, or on patch creation and removal these should not just be copied or removed in the same way, since that would remove the point of them conserving the history. Instead, as previously discussed, we would like for these to always be moved to the highest patch level possible.

All ownership change of particles between patches, including particles from a low level moving to a high level is handled as part of the ordinary particle exchange process described below.

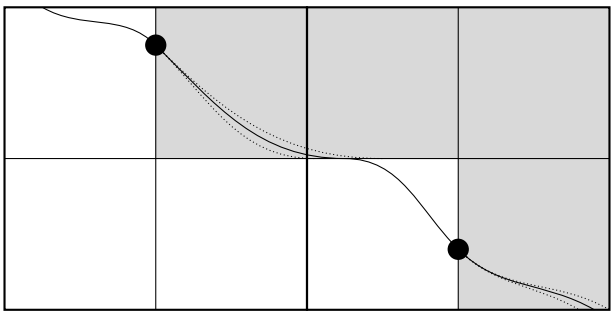


Figure 9: Illustration of a particle moving between high resolution (gray) and low resolution (white) sub-regions of two neighboring patches, providing boundary conditions multiple times at the dots drawn, initiating a population of particles at the higher level (dotted lines) which disappear at the resolution boundary.

6.9 Export/Import

As particles move around, they pass from patch to patch, transferring across resolution boundaries as they go. While the previous section presented the desired model, or invariants of the data structure as relates to the adaptive mesh refinement of the ownership, this section will describe the algorithm we have developed so that these desired invariants are maintained. This might sound like a trivial problem, but making this work correctly has been the main work in this project due to the many systems interacting. In broad strokes, the main concerns for this procedure are as follows:

- **Performance:**
  Being part of a large high-performance simulation with millions of cells, performance is of the essence. Each patch of 16x16x16 cells and e.g. 40 particles per cell will have ∼163.000 particles, so each loop is expensive.
- **Multithreading correctness:**
  The import/export procedure needs to handle the transfer of particles between patches which may be updated simultaneously by different threads.
- **Individual timesteps:**
  When a patch moves from one patch to another they will almost certainly not be at the same time. For a particle moving from patch A, across the corner of patch B, and finally into patch C, who owns it at which times, and what happens if patch B never has a time step where the particle is within it? When we have billions of particles and equally many iterations of the update, what can happen will inevitably happen.
- **Correct AMR behavior :** Sending and receiving have to have the behavior described in section 6.8, including tracers, down exporting, and normal advection of particles.
- **Correct dependency model:** The algorithm should be written such that it can be implemented using the dependencies of the code, i.e. without referring to the EXTRAS_T module.

6.9.1 Mode of communication

First, let us address the mode of communication and how we define the change of ownership. How do two patches communicate who should own which particles on their boundary? Here, three approaches could exist for handling the hand-off process and choosing which particles to transfer from one to another: A "push-based" approach where the previous owner of each particle decides, a "pull-based" approach where the receiver decides or a "consensus-based" where the two parties collectively decide. Since the change of ownership necessarily involves removal from the previous owner, and the Dispatch framework involves no modification of data not owned by the task itself, the deciding party needs to send a message to the other party about removal or deletion.

For reasons of performance, the pull-based approach is sub-optimal, since it would involve having to check all neighbors' data for particles to grab, possibly up to the previously mentioned on-order

100k particles for each of the about 100 neighbors. Worse, this would have to happen under a read lock, further worsening it. The independent timestamps also make the consensus-based approach problematic, since such an algorithm would have to make consistent choices for the sender and receiver, even though their view of the state of the system is at different simulation times.

Instead, we choose a push-based messaging approach where each patch investigates its particles, checking for each particle whether it should be handed off to any neighbor, and if so puts them into corresponding packages to be sent to the neighbors. These packages are then marked with the time of their departure and sent to the neighbor. When sent, they are then the responsibility of the receiver to "import", even though it may happen that the receiver never sees the particle as within their region of authority. In this case, it might immediately export it further, but that is no problem.

An additional advantage to this message-based model is that it maps naturally onto MPI-based communication if the messages are written in such a way that they can be serialized and deserialized.

We will for simplicity not discuss the possibility of different communication models for the different types of communication like down-exporting or tracer exporting, since we have found no such useful model, let alone justifying the extra complexity.

So how does this transfer of the particle packages between tasks work? Each particle solver has a member IMPORTS which is a multi-writer single-reader as described in section 4.2.3. When we iterate through the neighbor list to get the patches available for export, each of these has a particle solver associated. The problem here is that accessing this property would require casting the patch to a EXTRAS_T class. Doing this in the particle solver would create a cyclic dependency between the particle solver, and so this would have to happen in the extras module. This is what the previous code did, but it has the consequence of "polluting" the extras module with a bunch of code which must then be replicated between uses of the code. To overcome this, at initialization the extras module places a pointer in the CONNECT_T of each patch to the particle solver, thus linking the patch to the particle solver directly. This is illustrated in figure 10, where the resulting possible path is illustrated. Importantly, the pointer from CONNECT_T to PARTICLE_SOLVER_T is upcasted to an arbitrary void pointer to not create a cyclic dependency and thus has to be downcasted on every use.

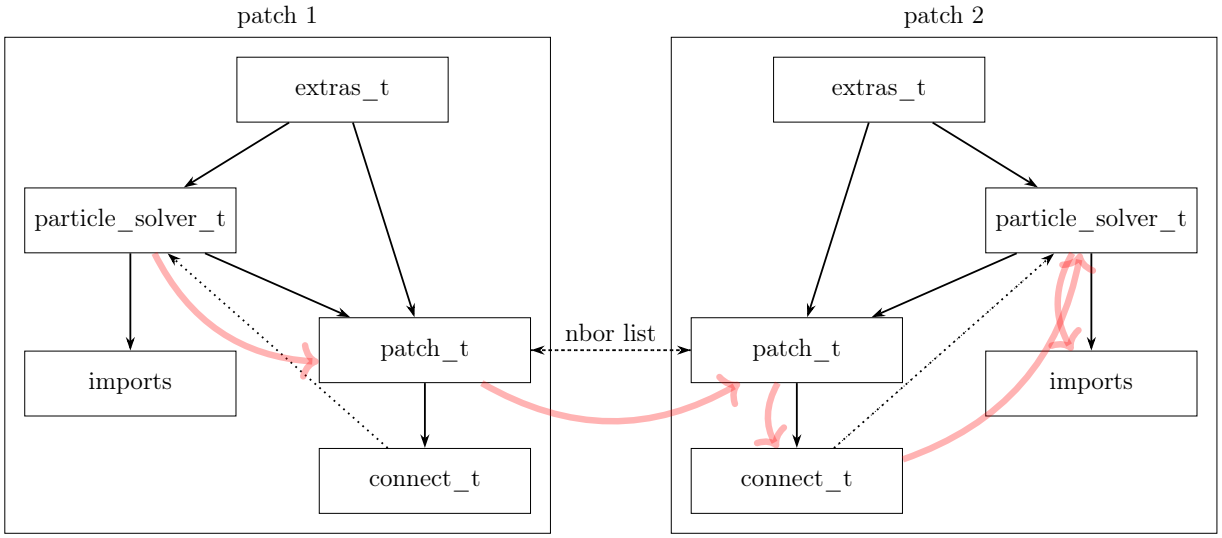


Figure 10: Illustration of the dependency model of particle solvers and how a particle solver locates the import lists of neighbor patches (transparent red path). Full arrows denote dependency, while dotted lines denote weak references.

6.9.2 Export procedure

With the process of sending a package of particles handled, the next is to define which particles to send and to where. For this, we will assume the existence of two functions. TRY_FIND_TARGET(LEVEL, PARTICLE which checks for all patches at LEVEL whether it covers the PARTICLE, and tries to return

this target. ADD_TO_EXPORT(TARGET, PARTICLE, REMOVE then adds this particle to the export package destined for TARGET in local coordinates of the receiving patch, and optionally removes it from the current patch. [10]

For tracers the desired export procedure corresponds to finding the highest level patch which includes the position of a given particle. Due to AMR, the particle can enter the region of a higher level patch at any time, but only if it exits the patch we need to consider neighbors at a same or lower level. In pseudo-code, the process looks like follows

```
def export_tracer():
  target = try_find_target(level + 1, particle)
  if target exists:
    add_to_export(target, particle, remove=True)
    return
  if particle outside self:
    target = try_find_target(level, particle)
    if target exists:
          add_to_export(target, particle, remove=True)
          return
    target = try_find_target(level - 1, particle)
    if target exists:
        add_to_export(target, particle, remove=True)
      return
```

For "normal" particles carrying mass, this process is slightly different. Here the communication lo lower levels is handed by the RESTRICT procedure. Furthermore, export to higher levels is not considered to be a change of ownership but simply provides boundary conditions, and should not result in removal of the particle from the current patch. On the other hand, we need to keep track of whether particles leave the high-resolution area and need to be available to again provide boundary conditions. For this, we define a property LEVEL which is increased and decreased to track this. The implementation therefore becomes:

```
def export_normal():
  target = try_find_target(level + 1, particle)
  if target exists and particle%level == level:
    add_to_export(target, particle, remove=False)
    particle%level = level + 1
  else if not target exists and particle%level == level + 1:
    particle%level = level
  if particle outside self:
    target = try_find_target(level, particle)
    if target exists:
      add_to_export(target, particle, remove=True)
      return
```

We should note that this export procedure not only provides boundary conditions when the particles pass into the region of high resolution but also particle initial conditions when patch refinement creates a new patch. In this case, nothing special happens, since particles throughout the region of authority (ROA) of the new patch will LEVEL defined such that they are ready to export, and TRY_FIND_TARGET will find this patch. On one of the first update iterations, the new patch will thus receive a bunch of data populating it. Similarly, tracers will register this new export target and be part of this first package to the new patch. The performance of this is of low importance since refinement happens rarely.

[10] In the code the chosen function names and splitting of the export functions are slightly different, and have been changed here to be more appropriate to this explanation.

To maximize performance, the check for particles being outside of the current patch is done in the particle solver when the particle data is in registers anyway, and recorded for each particle to be used here. Besides this especially TRY_FIND_TARGET needs to have high performance since this has to check every particle against the possibly 64 neighbors at higher levels (children and nephews) at every iteration. Similarly, we want ADD_TO_EXPORT to be efficient, but since this is only called several times proportional to the number of exited particles this is much less critical. Due to this performance requirement, iterating through the linked list of neighbors for every particle is out of the question. Instead, we preprocess the neighbor list once at the start and construct a structure of array (SOA) for each of the three levels relevant so TRY_FIND_TARGET can check the overlap with all of the patches efficiently and in a vectorized manner. In addition to the data about the position and size of each of the patches, this SOA also contains for each neighbor patch a pointer to an export package to which particles are added as well as a pointer to the receiving import list, allowing easy sendoff and cleaning when done adding particles.

A small optimization added to this export procedure relates to the export packages. During profiling, it was found that the allocation of the export containers was significant. To remove this overhead, we preallocate the arrays to be able to contain all of the particles of one face (16x16 particles per cell), meaning that appending particles should rarely require resizing. This by itself would be wasteful in many cases eg. for a fully populated child layer where children are never exported to. To avoid this, we recycle unused export containers through a cache shared between all threads[11]. Only if this is empty, we allocate new containers. This cache is the same linked list datatype used for imports. Since we do not need to traverse or modify elements of the cache in place, but simply remove and add elements, no additional locking is necessary.

The TRY_FIND_TARGET function looks through the candidate patches for one where the difference between the particle position $\vec{p}$ and the patch position $\vec{P}$ differ by less than half the width of the patch Δx for all coordinates:

$$|\vec{p}^k - \vec{P}^k| < \Delta x/2 \quad \forall\, k \in \{x, y, z\}$$

which unfortunately on computers is calculated with finite precision. In a 32-bit floating point number, the 24-bit significance will use all of its precision (at the boundaries of the box) to specify the patch location and not leave any sub-cell precision at all. Just small rounding errors on the last digit will thus cause particles to move several cells. To avoid this, we make sure not to convert all positions to be relative to the sender patch center.

Unfortunately, due to rounding errors, particles at the faces of patches may still compare "outside" of all patches. This is extremely rare, but that is not good enough: To simulate the formation of a star for hundreds of kiloyears with a maximal resolution of a day would require some individual patches to be updated on order billions of times. Thus if the export algorithm due to numerical accuracy has just a one-in-a-million risk of missing a particle, almost all dust mass will be lost and tracers will be unusable. To fix this, we accept an export target to "contain" the particle as long as it is within half a cell of the patch.

With this padding, the only way to lose particles would be if they stepped across a whole neighbor patch in just one step. The Courant condition and the fail-safe on the integrator make sure this does not happen. In the code, we check whether the procedure fails to find a proper neighbor, and as of time, the code has been running for 70 core days without this happening.

6.9.3 Import procedure

While the export procedure was very complex in workings[12], the import procedure is much simpler: When importing particles, the particle solver already has a reference to the import list, the list is owned by the same task and all particles herein are unconditionally the responsibility of the solver.

[11] This is identical to just using a specialized allocation

[12] And similarly so in debugging. Errors occurring at a frequency of core-days are typical in this context.

The procedure works by iterating through the import list and checking all particles. In general, we assume that the particles drift with a constant velocity between these times. If our current patch is behind the code time at which the package was sent, and the particle has not yet drifted into the patch, we will just handle this particle later to avoid handing particles outside of the ROA if not necessary.

When importing particles, there are two categories of particles to handle: particle ownership transfer (i.e. massive particles from the same level and tracers) and boundary condition particles (i.e. particles from the lower level).

While the first case is very simple, especially since the exporting particle already transformed to coordinate local coordinates, the second case requires a bit of care. For each cell in the lower level patch, 8 cells exist in the higher level patch with each 1/8 of the mass. To conserve the number of particles per cell, we therefore want to split the incoming particles into 8 parts, moving them in one cell in each direction to ensure an even density between cells.

When particle packages are emptied of all particles, we remove these from the import list and add them back to the export cache to be recycled for another round of particle exporting.

All code related to importing and exporting is completely written from scratch.

6.10 Patch deallocation

In the same way that AMR creates new patches where they are necessary, it similarly implies the removal of patches where they are not. To support this, we have to define a proper cleanup procedure. This involves both low practically deallocating all used memory to not heave leakage, as well as the physical handling of tracers which should not be lost in this deallocation.

To not lose the tracer particles, we first make sure to import all particles in all packages in the import list, even those that have not yet drifted into the patch. We can be sure that no more are being prepared since deallocation only happens when the number of other tasks referencing this patch reaches zero.

After importing the particles, we iterate through the particles and use the normal export functions to export all tracers to the parent patch, which is not responsible for the region of space. After this, everything is deallocated, and we ensure the conservation of the tracers.

This code is completely written from scratch.

6.11 Data layout

As the title of the textbook "Algorithms + Data Structures = Programs" eludes, the design of the data structure should be done with similar care as the algorithms themselves. While particle update and handling are correct to get right, at any one time by far the majority of patches will be waiting to update, just consuming memory. While the MHD data has on order 10 variables of four bytes each in each cell, the dozens of particles in the same cell will consume much more. Reducing the memory requirement for each particle is thus of prime importance.

As eluded to earlier, float32 precision is completely insufficient for particles if stored in global coordinates. To avoid using 3x4 bytes more per particle, we therefore use local patch coordinates in units of cells. We have chosen to represent the position as $q + p$ where q is a 3-vector of one-byte integers specifying the closest cell, and p is a float 32 representing the distance from the center. This has a slight precision advantage, but primarily it allows efficient lookup for patch gas data since q are the indexes in these arrays.

With this representation, we use 15 bytes for the position of each particle. Beyond these, each particle has five floats representing the x-, y- and z-velocity, mass, and particle size, a four-byte particle ID useful for tracers, and two 1-byte integers saving their level and whether they have left the patch. In total this representation requires 41 bytes.

For obvious performance reasons, these are all stored as separate flat arrays in the particles_mod structure as a structure of arrays.

The accretion, export, and pruning algorithms all work by removing individual particles. A naive algorithm to remove a particle as would feasibly be used in e.g. Python LIST.REMOVE() is copying all elements after the one being removed, and moving them one step to the left. This would be extremely expensive to do for every particle removal here. Instead, we have chosen an algorithm where removed particles are assigned a mess of "-1", clearly distinguishing them from actual particles. A COMPACTIFY function can run through arrays and remove any holes.

To similarly optimize particle insertion, we use a classic dynamic array algorithm for adding particles to not reallocate and move all data at every insertion: At any time the arrays have space for m particles but only contain n particles, the last of which is at space l ($l - n$ = number of holes and $m - n$ = number of free spots). If $m > l$, then adding is as simple as putting the data into slots l and incrementing l. Otherwise, the data is reallocated to have $2m$ slots. Like normal dynamic array insertion, this gives constant time insertion on average.

Currently, we do not ever release this leftover memory, but if memory became an issue resizing when half full would be reasonable, although given the renormalisation this would not be a typical occurrence

This data structure has not been significantly modified.

6.12 I/O

I/O or Input/ Output is the procedure to get data in and out of the simulation. While the output procedures are working well, the input procedure allowing simulation restart from a file has not been written yet. Therefore we will only discuss the output.

Since particle data is huge and gas data already takes up terabytes, we want to be economical about it. For this reason, we have three different types of output:

- High-frequency tracer output. The tracers compose a tiny fraction of the data, and we may be interested in tracking these at a high cadence.
- Low-frequency dust averages. When outputting MHD data, we also output the cell-averaged dust density, momentum, and number of particles. This might e.g. be useful to track dust concentration over time.
- Very low-frequency dust dump. To restart a simulation from a certain point, we need to dump all data into a file so it can be reloaded again. This is thus critical for large-scale simulations.

Just after importing and evolving particles and before exporting, the particle solver checks whether it is time to perform any of the output types. If it is, a file is opened (if not already done), and the data is written to it.

While the data structure in the program is composed of many arrays connected by pointers, this is not something we can write to a file. For this, we need to "serialize" or "pack" the data, i.e. transform it to a flat array of bytes that can be written to e.g. a file. We have implemented a "pack" function to perform this, selecting the data based on the type of output, putting it together, and writing the correct metadata.

For this output code to be useful, we need to be able to read it. For this, Python code was written to parse this and read it in so it could be plotted. For symmetry, the corresponding unpack function was also written in the particle container.

The packing and unpacking code is almost completely rewritten along with the output type selector, but the file name handling is pretty unchanged. Due to the high cadence output only present for tracers, a threading error was found and fixed in the file handling of dispatch.

6.13 MPI parallelization

In this project, we did not have time to implement MPI integration with particle integration, but we have structured the code to make this easy in the future. Due to the task shadow copy model of dispatch, the problems to be solved will be the following:

- How to forward exports to a "shadow" patch to the real patch on another rank. This should be made significantly easier by already having pack/unpack methods for the transmitted messages. And
- How to allow the load balancer to transfer the particle solver and patch to another rank. Similarly, this will assumably make use of the ability to pack and unpack, and the ability to construct a particle solver instance from this package data will probably share code with the file input implementation.

Other challenges will certainly show themselves, but we have at least designed the code to be adaptable to the anticipated major obstacles.

7 Results

In this project, we have implemented active dust and tracer particles into a large, physically realistic model of star formation. The motivation for this has been to allow the inclusion of dust that can become dynamically important in the disk, play an important role as an observable tracer, reprocesses the starlight of the newborn star and set the local temperature of the gas, and ultimately is the raw material for planets. The short-term goal of the project has been studying the accretion and ejection ratios of dust in planetary systems, which a functioning and high-performance implementation will enable. This section will address these two goals in succession.

All plots are produced from the same simulation, which we have run twice for redundancy in case of a crash. All plots were produced for both result sets producing no notable difference. These simulations are run as a zoom-in simulation of a forming core in a larger molecular cloud simulation by [36], specifically core 13 as described in [37]. We simulated this core from initial collapse and 13.000 years forward.

In the simulation we allow up to patch level 20, giving a max cell resolution of $0.78AU$. The refinement criteria refine patches where the cell size is above 1/16th of a jeans length, resulting in around 550-600 patches at the maximum resolution, corresponding to 2.3 million cells. In total among all levels, a total of just above 10.000 patches exist resulting in 40 million cells. Of these, only around 4000 are initially created, with AMR creating most of the rest at levels 8-10.

As for particles, we are simulating sizes distributed logarithmically from 1 micrometer to 1 centimeter, represented by 40 particles per cell giving 10 particles per cell per decade in particle size, giving a total of about 100 million particles. At initialization of particles on the original 4000 patches, 1 in 400 particles were selected as tracers across all sizes. This gives 1.45 million tracers. These particles are all assumed to have densities of 3 g/cm^3, similar to the bulk density of primitive chondrites that are made of primordial Solar System material [38],

For output, we make a trace output every 21 years and do a MHD output every 210 years. Since the particles in total take up 100 gigabytes and we do not intend to do any analysis on these, we have disabled full particle output.

The simulated data is the result of running each simulation just shy of three days of simulation on 32 CPU cores, corresponding to 90 CPU days. In this time, each system has evolved around 13.000 years, involving $1.1 \cdot 10^8$ patch updates, corresponding to 400 billion cell updates or 18 trillion particle updates. This has produced 1420 GB of analysis data.

7.1 Performance

One goal of the project was to create a high-performance simulation of the dust particles. Therefore we will evaluate the run time consequences of having the dust module activated in a DISPATCH simulation. Running DISPATCH in the described configuration, we have recorded the amount of time used in the various functions. The parts of the particle integrator using more than 1% of the total simulation CPU time (i.e. including MHD, sinks, self-gravity and all bookkeeping) are from most to least significant:

- The prune function uses around 13% of the CPU time. Currently, this is a $O(n_{cell}^2)$ algorithm and is run on every single patch update. This function is a prime candidate for further optimization, both regarding its algorithm and implementation.
- The implicit update uses 11% of the time. This time includes multiple reads and writes to patch data as well as a quite expensive update for every single particle, and so this is expected.
- The export function uses 8% of the time. The requirement on the AMR-related checks even for particles inside the patch makes this very expensive. Part of this expense comes from the iteration through all neighboors at a given layer for every particle. By implementing a look-up based check, this may be sped up.

- Error checking functions are run multiple times per update, using about 7% of the runtime. These can readily be disabled.
- Calculating cell averages like mass and momentum which are used in the prune and split functions as well as for self-gravity and data output use 7%. Currently, this is called multiple times per update, so reusing the data can remove some of this
- Finally, a handful of percent of the time is used for various bookkeeping not included in the other times. This could be optimized

Thus in total, the particle integrator running with 40 particles per cell uses quite precisely 50% of the total runtime. The particle integrator thus uses about $\sim 6\frac{\text{core}-\mu s}{\text{cell update}}$ at 40 particles per cell or $\sim 150\frac{\text{core}-ns}{\text{particle update}}$, with the self-gravity and MHD plus the framework for controlling the simulation using the other $\sim 6\frac{\text{core}-\mu s}{\text{cell update}}$

Possibly the particle integrator causes a further performance degradation beyond simply the time spent in the functions. The unpredictability of the exact runtime of the particle solver for each patch means that some patches may be slowed down significantly. If this is e.g. a central patch, the extra latency may bottleneck the whole simulation by starving the task queue from any work. Since only one thread is working in the particle integrator, it may only show up as 1/32th $\sim$ 3% of the time used, with 95% used in NO_QUEUE. This was the case before implementing the accretion module.

7.2 Validation of implementation

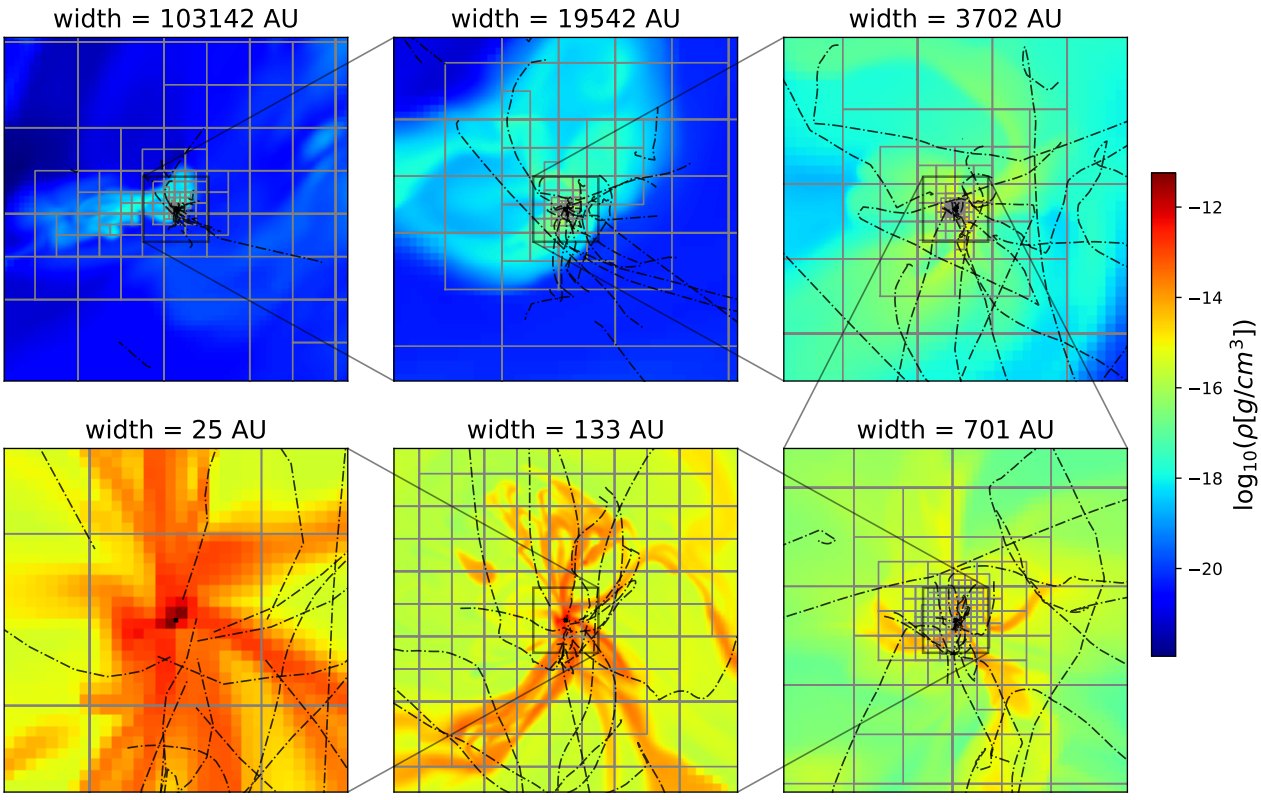


Figure 11: The gas density and patch mesh shown across a large range of scales centered on core 13, showing the state of the system 3842 years after sink creation. On top of these, a random selection of 50 tracer particle trajectories was integrated for 13.3 kyr. Note that while the gas density is not stationary, the sink is.

In the context of software validation, tests exist on a spectrum from unit tests to integration tests. Unit tests validate each of the tiny moving parts like "We correctly export particles in this very specific condition" or "Does this function correctly remove particles according to the accretion ratio". Integration tests on the other hand validate that all components play nicely together and that the result has the desired behaviour. In this project, showing validation of every single feature would be

completely infeasible. Instead, we have made observations and analyses that support that important aspects of the resulting behavior are working as intended illustrated by figure 11 and 12. On these figures, notice the following:

- Particles generally move in sensible paths, moving in almost straight or hyperbolic trajectories outside of the high-density regions, while having much more curved paths when under the influence of drag near the core and generally bending towards the core due to gravity. This shows the correctness of the particle integrator and gravitational solver.

  Note that this is simply a 2-dimensional cross-section of a 3-dimensional simulation, so particles at the center of the plot are not necessarily close to the core, and particles that look stationary on the plot may be moving in the z-direction. Similarly the gas background changes significantly over time, this is just one snapshot, while the tracer particle trajectories show the time-dependent path. Before the start of the integration, the simulation has been Galilean transformed into the rest frame of the pre-stellar core, making the newborn star relatively at rest, and minimizing bulk drifts.

- As particles move across the system, passing between patches and up and down levels (shown as color changes on fig 12), we see no sudden direction or position change or even particles disappearing, showing the correctness of the export and import modules. Among these close-approach particles, we interestingly even see a few passing back and forth between levels.

- At the center, a lot of tracks stop as the particles are accreted by the star. This validates the implementation of accretion.

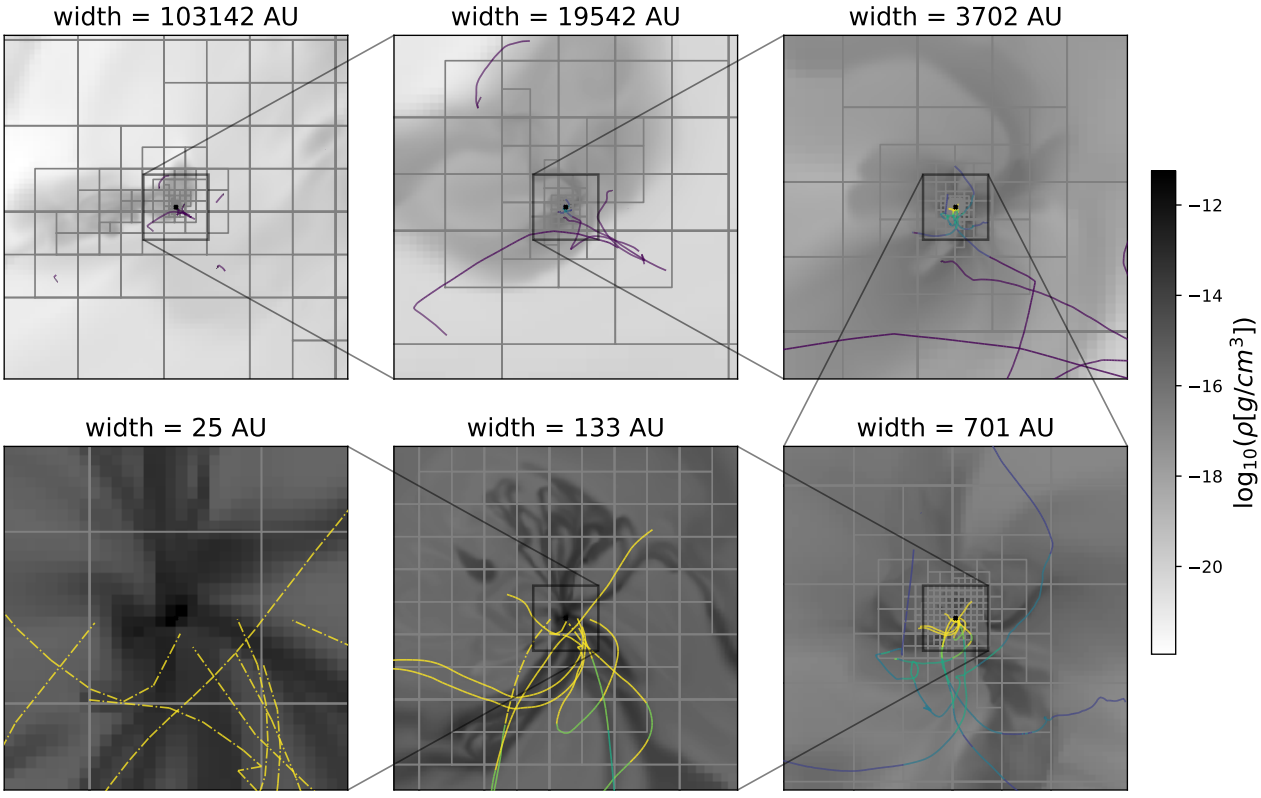


Figure 12: Similar to 11, a new set of particles and the particle trajectories are colored by the current particle level, highlighting patch-level changes. The color ranges from yellow at level 20 to magenta at level 15 or below.

7.2.1 Near-core behavior and particle size effects

Moving closer to the overarching goal of studying near-approaches and accretion of dust, figure 13 shows the behavior of a random selection of particles with close approaches to the core. There is a very clear selection bias here, with almost all of the particles accreted being small and non-accreted

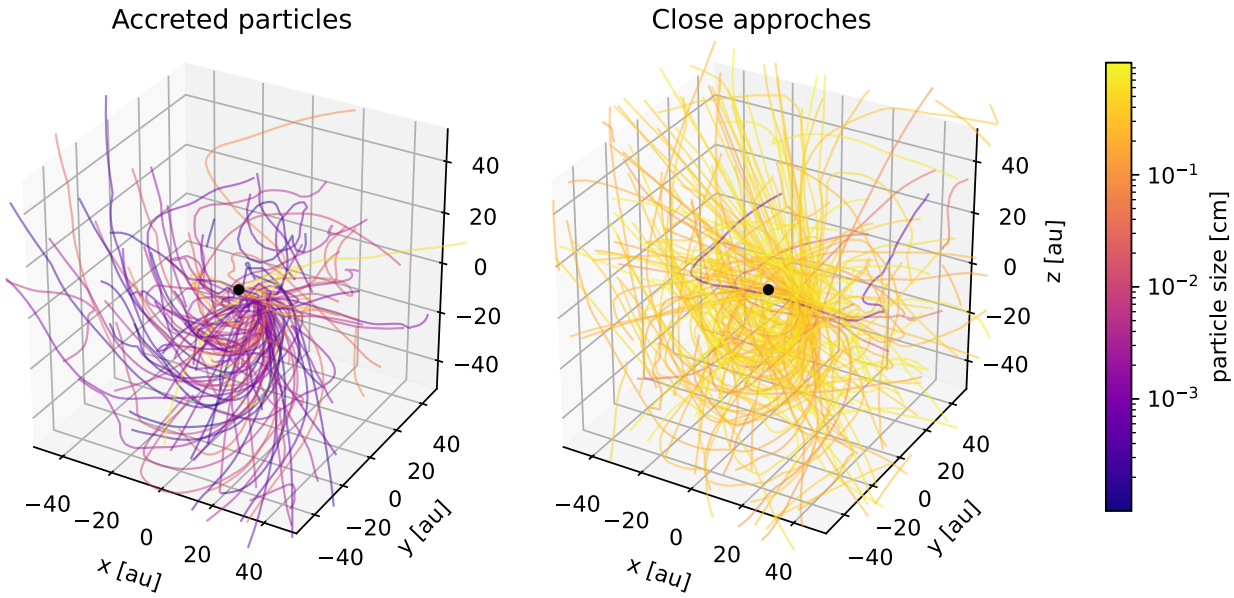


Figure 13: 3D view of 50 random particles with close approaches (defined as passing within 50 AU). To the left, only accreted particles are selected, while to the right only non-accreted particles are shown.

particles being big. This is in line with our physical intuition since we expect small particles to be tightly bound to the gas and thus be accreted with it, while large particles to a larger degree ignore the gas and move on ballistic trajectories, without any way to dissipate orbital energy. It is also clear that the small particles are moving in a circular motion around the z-axis which heavily implies that it traces the forming gas disk.

Notably, if we were able to actually capture the disk at a later time when it had properly formed, we would expect this to reverse: As the gas rotates at a sub-Keplarian motion with the particles rotating Keplarian, particles large enough to be only loosely coupled to the gas, e.g. with a Stokes number close to unity would constantly be slowed down by friction, losing their energy and falling into the star. In a dynamically cold and thin protoplanetary disk drag forces in the vertical direction will on time scales similar to the orbital time scale damp the vertical motion of particles larger than 100 microns accumulating them in the midplane[39]. Radial drag forces then efficiently move them towards the star. The height of such a dust layer depends on the particle size, the gas scale height, and stirring caused by large-scale flows from e.g. the vertical shear instability, streamers landing fresh material, and other disk scale instabilities. Once this happens, large dust grains will not reach the upper layers of the disk, corresponding to a few gas scale heights, where the wind is launched, and therefore the wind will be depleted in large dust grains [40]. In the current simulations such a quiescent state is far from have been reached and it is an open, but very interesting, question when stirring becomes ineffective in launching large grains. If this only happens in the late Class 0 phase after e.g. 100 kyr the early outflows that carry most of the mass may be rich in large grains.

This physically plausible behavior gives further confidence to the implicit particle solver and its interaction with the gas through drag.

7.3 Preliminary physical results: Processing efficiency analysis

Having validated the algorithms, we turn to the question of accretion and ejection efficiency: How large a fraction of the particle mass entering the system turns is being ejected again? With the resolution we are simulating, tracking particles to orbits close enough to get significant heating from the protostar is not possible. But with a hope of self-similar behavior of the system, we hope to gain information from the more large-scale dynamics we have simulated. Specifically, we will be tracking particles that enter a given threshold radius from the star, e.g. 20 AU of the star (about 1 patch). Of these, some will be accreted, some will simply orbit there until the end of the simulation and finally, some will be shot out

due to interactions with the gas. We define "Ejection" to imply that the particle has entered a given radius, but ends up at least twice as far away. In future work, a much higher resolution simulation that is able to resolve the inner jet may more precisely look at the exact behavior close to the protostar.

The tracer particles do not represent any dust mass since they are initialized uniformly according to the resolution, not the density. This means that some tracers may trace the paths of orders of magnitude more dust mass than others. Since the physical quantity of interest is the ratio of the mass accreted and ejected, we distribute the gas mass of each patch among the tracers according to the same $\alpha = -3.5$ power law distribution as at initialization. In this way, the physical dust mass from all points in space is equally represented, even if the tracer spacing is not, and we can make statements about the physical dust population. This has to be done with care since any part of the patch refined by child patches contains no tracers and only gas, so this gas mass should be subtracted before spreading it to the particles. Finally, distributing all mass does not follow the dust-to-gas ratio which is only on order 1%, but as long as we are only discussing ratios, this proportionality is of no consequence.

During the simulation, the core grows from $0.0055M_\odot$ at the first snapshot after sink creation (the snapshot used to attribute mass to tracers) to a mass of $0.0325M_\odot$. Our process of distributing gas mass to the tracers and summing up all accreted tracers accounts for a total of 96% of this mass. The lacking 4% deficit physically makes sense, since inertial particles do not accrete as we also saw in figure 13.

Even with the large amount of tracer particles in our simulation, the many orders of magnitude differences in the densities involved may make individual tracers with large masses dominate the statistics. In figure 14 we can see how the mass distributuions for accreted and ejected particles are not dominated by any single particles but rather quite smooth lines. The figure also shows that this could easilly NOT be the case, since all particles at low patch levels each corespond to a huge amount of mass. This shows that even though cells at level 8 has 5-10 orders of magnitude lower gas densities than cells at the core at level 20, each cell has a volume $(2^{12})^3 \sim 7 \times 10^{10}$ times larger. Just one particle from level 8 would have completely dominated any statistic.

So is the absense of these in the accretion pure luck? Remember that these particles of very low resolution are only created at large distances where the gravitational attraction is similarly very low. In our simulation, the closest particle with a mass larger than $10^{-2}M_\odot$ is a distance of 24kAU away. Given enough time these particles may drift in, highlighting the importance of the particle splitting procedure for particles with drag and gravitational feedback on the gas.

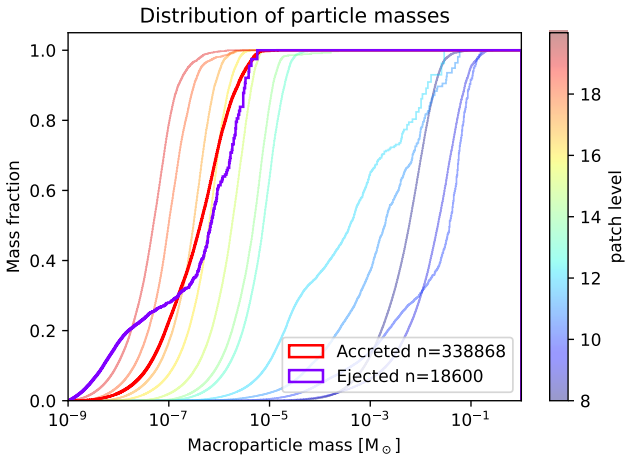


Figure 14: Cumulative distribution of accreted and ejected (threshold 20AU) particle masses as a function of individual tracer particle masses (thick lines), compared to the initial particle mass distributions for each patch level (faded lines).

In figure 15 we show the ratio between the ejected and accreted mass carried by the tracers as a function of the cutoff ratio. By normalizing the accreted mass, we can compare the efficiency of particle ejection across particle sizes and cutoff ratios, even though the total mass budget of each of these vary by many orders of magnitude. Interestingly, it seems like particles below 0.1 mm have very little variation in the accretion. This is probably because particles this small simply trace the gas, irrespective of their actual size. This is futher confirmed by looking at how the 1-0.1 mm particles diverge from smaller particles at 10-20 AU. At this point, the gas density becomes high enough for mm-sized particles to start feeling the gas friction and follow the gas motion; their Stokes number becomes less than unity.

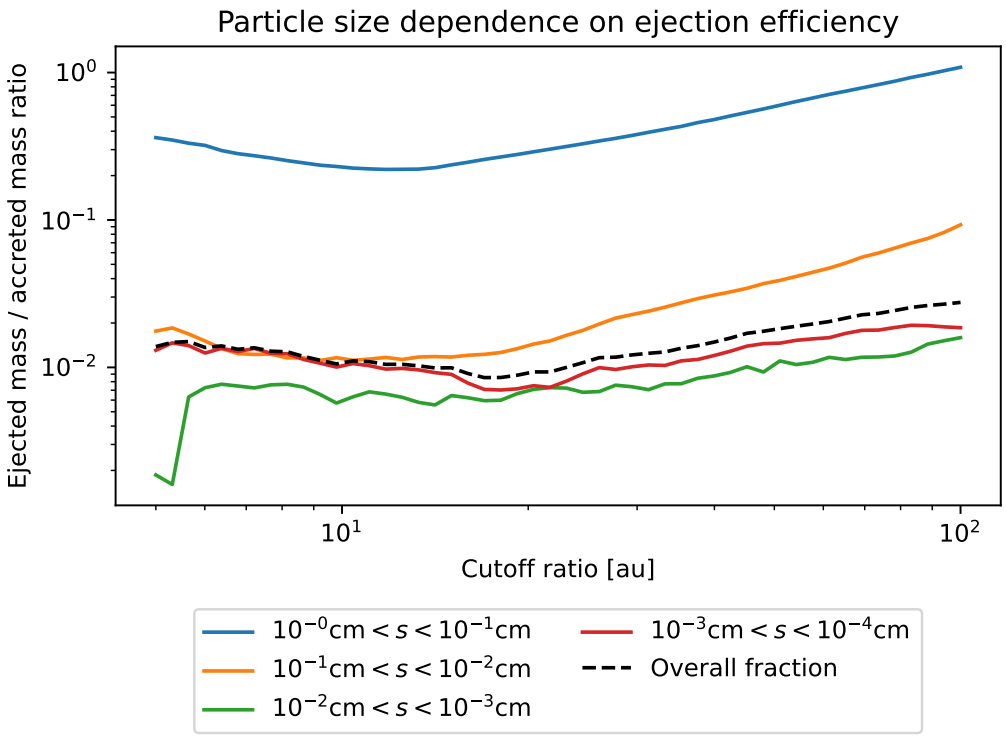


Figure 15: Ratio of ejected (comes within the given cutoff but ends up beyond it) and accreted mass split up across four size buckets of an order of magnitude. This is plotted as a function of the cutoff ratio chosen to show the dependence on this parameter.

The smallest scale we can hope to capture in this simulation would be some multiple of the size of the region from which gas and particles are accreted. In the code, this is currently 8 cells in each direction or about 6 AU. This is very possibly the cause for the plateauing of around 10 AU of the ejection ratios for all particle sizes.

Notably, this is all looking at the very early system, a few thousand years after the second Larson core formation. One might reasonably expect that later in the evolution when the disk is formed and the vertical outflows are more established, ratios will be different, and that resolving the disk and outflow properly is central to getting the efficiency right. To investigate this, we have analyzed tracer data from a level 24 zoom-in simulation performed using RAMSES [33]. These four extra levels provide $2^4 = 16$ times better resolution resulting in a cell width of 0.05 AU. This should much better resolve the system. Unfortunately, the particle tracers in RAMSES are simply non-inertial particles, blindly following the gas, so we cannot compare them with the particle's separated ejection ratios. Furthermore, due to the much higher resolution, supplemented with the fact that DISPATCH is a much higher performance code than RAMSES, the available dataset only spans 400 years. This means that for many of the particles, we do not know where they ultimately end up: Back in the ISM or the star? With this data, we have an output cadence of two years, but the much lower dynamical timescale means that we are limited by the sampling frequency: defining the cutoff too small, and only very few of the particles that get within it are recorded, messing up the statistics. Experimentally, we have found that a radius cutoff of 10 AU will ensure most particles stay within it for multiple snapshots.

Figure 16 shows the resulting ejection efficiency. Notably, the resulting 17% is much higher than the one found with level 20 integration, especially when considering that one should compare with the small DISPATCH particles. This hints that discretization errors are still very significant, showing the importance of this high-performance particle integration in DISPATCH to explore this further in future work.

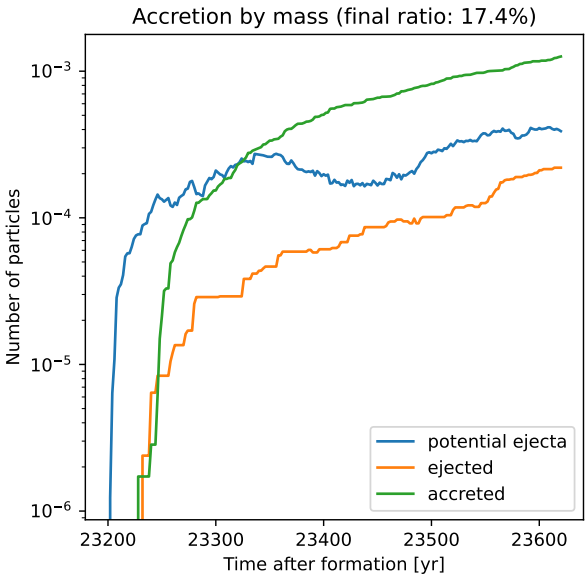


Figure 16: RAMSES level 24 zoom-in simulation of the core slightly later showing the amount of cumulative ejected and accreted medium, as well as mass within ice radius but not yet ejected (potential ejecta). Note that the y-label is wrong, and is simply the amount of accreted mass in code units.

It should be emphasized that the RAMSES model is a state-of-the-art, specially tuned version, which is optimized to zoom-in models, and a few to 20 times faster in run time than similar models with GADGET[41], ENZO[42], FLASH[43], or the public RAMSES code. Not to mention AREPO[44] which is almost 50 times slower, but also gains a similar factor from individual time stepping of the cells. Yet, this very model, only using MHD dynamics and simple passive tracer particles, has taken close to one month to run with level 20 resolution for 20 kyr on 1,000 CPU cores, and the follow-up level 22 and then level 24 runs for just a few thousand years have taken another 6 months to integrate on 1,000 cores. DISPATCH is radically faster, and even with active dust particles and a much higher number of particles than in the RAMSES models, 13 kyr of integration time was accomplished on a single 32-core machine with three days of integration, which is 100 times faster than RAMSES, and close to 1,000 times faster than most leading codes

8 Conclusion

In this project, we have demonstrated the functioning implementation of a particle simulator as part of the large-scale astrophysical simulation framework DISPATCH, enabling large, detailed heterogeneous studies of the complex and dynamic systems that are protostellar and protoplanetary. This is motivated by the desire to study the size-dependent dust enrichment of material as it flows in from the surrounding molecular cloud and concentrates in the protoplanetary disk, how it is ejected again in outflows, as well as a general desire to enable future studies of the many consequences of dust in these systems.

Owing to the high performance of the simulation, we have been able to integrate 60 million particles in, and interact with, the magnetohydrodynamically and self-gravitationally modeled environment around a forming protostar for more than 10,000 years of simulation time on just a single CPU running for three days, with an average per-particle cost of 150 core-ns/particle update. We have addressed the numerous numerical, algorithmic, and programmatic challenges, and presented the algorithms we have developed for solving these.

Specifically, our implantation mirrors the patch-based adaptive mesh refinement approach already implemented for gas, gravity, and magnetic fields by having each patch own a large number of dust particles and updating these in lockstep with the field updates. This enables fast cell-level feedback with the gas through drag and proper integration with self-gravity while using local time steps and refinement of the resolution where necessary in a way that scales to multi-core computers. We have shown how to properly represent large particle collections, initialize them, integrate them using an implicit Euler method under the influence of gas and gravity, transport them between patch boundaries and levels, re-normalize to ensure constant sampling resolution over time, and finally accrete into the star. During the implementation, care has been taken to make the code modular and modifiable by writing it in an object-oriented manner, easing future expansion of the code.

With the fiducial simulation of the region around a forming star within a larger molecular cloud, we were able to demonstrate the correctness of the implemented procedures, as well as perform a preliminary investigation of the particle size fractionation, with heavy particles overwhelmingly scattering around the protostar, while smaller particles trace the gas and accrete. Looking at the scaling behavior of the accretion and ejection rates in our simulation, as well as comparing with higher resolution simulations made in RAMSES, we can conclude that quite a significant fraction of the dust may be ejected, possibly up to or more than 10%, but also that higher resolution simulations are needed to determine this more exactly.

From our simulations, we have observed that in answering this question, special focus should be paid to the role of particle sizes (or equivalently, Stokes numbers), since at early times, when the stellar mass is low and the disk is not yet established, small particles that simply trace the gas are much easier accreted while the ballistic trajectories of big particles to a larger degree pass by instead. How this will interplay with a resolved outflow will be interesting to see, but simple gas-tracing approaches probably do not suffice for this study.

Beyond performing higher resolution runs with the simulations, the ability to model non-collisional mass could also make particle integration useful in the low-density environments of the ISM, possibly with the integrator changed to model stokes drag. For supercomputer-scale runs, which are especially needed for these larger-scale models, the MPI integration will also need to be finished, as well as restarting ability in case code crashes.

In this project, we did not derive any physical consequences of the drag- and gravitational effects on the gas. Investigating when and where this becomes significant would be interesting in itself, but would also further guide the normalization procedures and understanding which properties are most important to conserve in the pruning procedure and where it may be useful to keep more particles than average.

For even more realistic models of planetary formation, it would be natural to add particle evolution such as coagulation, fraction, gas accretion, evaporation, etc. This is not too dissimilar to how the

re-normalization procedures work, since it similarly compares nearby particles, and produces a new set of particles from these.

In conclusion, by enabling these future possibilities, this project has taken a small step towards advancing the state-of-the-art planetary simulations, and will hopefully soon be used for meaningful studies into the field of dust dynamics, dust-gas coupling, and ultimately establishing when and where the conditions for planetesimals and planets to form are right in the disks around young stellar objects. While these are future perspectives, with the first fiducial runs that were carried out at the very end of the project, we have established that dust can efficiently be recycled early on and carried away in outflows. Inside 10 AU from a young star dust, dirty ice enriched in metals may evaporate, and recondense to the bare grains, giving optimal conditions for reprocessing of metals into solid form. These models are to our knowledge among the very first to demonstrate this and show that young stellar objects are potentially important for the dust life cycle in the Universe.

References

1. Nordlund, Å., Ramsey, J. P., Popovas, A. & Kuffmeier, M. DISPATCH: A Numerical Simulation Framework for the Exa-scale Era. I. Fundamentals. *Monthly Notices of the Royal Astronomical Society* **477,** 624–638 (June 2018).
2. Boss, A. P. Temperatures in Protoplanetary Disks. *Annual Review of Earth and Planetary Sciences* **26,** 53–80 (1998).
3. Van Dishoeck, E. F. Chemistry in low-mass protostellar and protoplanetary regions. *Proceedings of the National Academy of Sciences* **103,** 12249–12256 (Aug. 2006).
4. Sai, J. *et al.* Early Planet Formation in Embedded Disks (eDisk) V: Possible Annular Substructure in a Circumstellar Disk in the Ced110 IRS4 System. en. *The Astrophysical Journal* **954,** 67 (Sept. 2023).
5. Currie, T. *et al.* Images of embedded Jovian planet formation at a wide separation around AB Aurigae. en. *Nature Astronomy* **6,** 751–759 (June 2022).
6. Shakura, N. I. & Sunyaev, R. A. Black holes in binary systems. Observational appearance. *Astronomy and Astrophysics* **24,** 337–355 (Jan. 1973).
7. Lesur, G. A systematic description of wind-driven protoplanetary discs. *Astronomy & Astrophysics* **650,** A35 (June 2021).
8. Birnstiel, T., Fang, M. & Johansen, A. Dust Evolution and the Formation of Planetesimals. *Space Science Reviews* **205,** 41–75 (Dec. 2016).
9. Binkert, F., Szulágyi, J. & Birnstiel, T. Three-dimensional dust stirring by a giant planet embedded in a protoplanetary disc. *Monthly Notices of the Royal Astronomical Society* **523,** 55–79 (July 2023).
10. Teyssier, R. Cosmological Hydrodynamics with Adaptive Mesh Refinement: a new high resolution code called RAMSES. *Astronomy & Astrophysics* **385,** 337–364 (Apr. 2002).
11. Price, D. J. *et al.* Phantom: A smoothed particle hydrodynamics and magnetohydrodynamics code for astrophysics. *Publications of the Astronomical Society of Australia* **35,** e031 (2018).
12. Colzi, L. *Isotopic fractionation study towards massive star-forming regions across the Galaxy* (Jan. 2021).
13. Schulz, N. S. *The Formation and Early Evolution of Stars* (Springer, Berlin, Heidelberg, 2012).
14. Lee, Y.-N. *et al.* The Origin of the Stellar Mass Distribution and Multiplicity. *Space Science Reviews* **216,** 70 (June 2020).
15. Larson, R. B. Numerical Calculations of the Dynamics of a Collapsing Proto-Star. *Monthly Notices of the Royal Astronomical Society* **145,** 271–295 (Aug. 1969).
16. Armitage, P. J. *Astrophysics of Planet Formation* 2nd ed. (Cambridge University Press, Cambridge, 2020).
17. Velikhov, E. Stability of an ideally conducting liquid flowing between cylinders rotating in a magnetic field. *Journal of Experimental and Theoretical Physics* (1959).
18. Lee, C.-F., Li, Z.-Y., Shang, H. & Hirano, N. Magnetocentrifugal Origin for Protostellar Jets Validated through Detection of Radial Flow at the Jet Base. en. *The Astrophysical Journal Letters* **927,** L27 (Mar. 2022).
19. Freidberg, J. P. *Ideal MHD* (Cambridge University Press, Cambridge, 2014).
20. Finlay, C. en. in *Encyclopedia of Geomagnetism and Paleomagnetism* (eds Gubbins, D. & Herrero-Bervera, E.) 3–6 (Springer Netherlands, Dordrecht, 2007).
21. Krumholz, M. R. & Federrath, C. The Role of Magnetic Fields in Setting the Star Formation Rate and the Initial Mass Function. *Frontiers in Astronomy and Space Sciences* **6** (2019).
22. Picogna, G., Stoll, M. H. R. & Kley, W. Particle accretion onto planets in discs with hydrodynamic turbulence. en. *Astronomy & Astrophysics* **616,** A116 (Aug. 2018).

23. Youdin, A. N. & Goodman, J. Streaming Instabilities in Protoplanetary Disks. en. *The Astrophysical Journal* **620,** 459 (Feb. 2005).

24. Lombardi, M., Alves, J. & Lada, C. J. Molecular clouds have power-law probability distribution functions. en. *Astronomy & Astrophysics* **576,** L1 (Apr. 2015).

25. Evans, C. R. & Hawley, J. F. Simulation of Magnetohydrodynamic Flows: A Constrained Transport Model. *The Astrophysical Journal* **332,** 659 (Sept. 1988).

26. Ramsey, J. P., Haugbølle, T. & Nordlund, Å. A simple and efficient solver for self-gravity in the DISPATCH astrophysical simulation framework. *Journal of Physics: Conference Series* **1031,** 012021 (May 2018).

27. Nelson, A. F. Numerical requirements for simulations of self-gravitating and non-self-gravitating discs. *Monthly Notices of the Royal Astronomical Society* **373,** 1039–1073 (Dec. 2006).

28. *Auto-vectorization in GCC - GNU Project*

29. Dagum, L. & Menon, R. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science and Engineering* **5,** 46–55 (Jan. 1998).

30. *MPI: A message passing interface* in *Supercomputing '93:Proceedings of the 1993 ACM/IEEE Conference on Supercomputing* (Nov. 1993), 878–883.

31. Pardoe, J. & King, M. en. in *Object Oriented Programming Using C++: An Introduction* (eds Pardoe, J. & King, M.) 177–188 (Macmillan Education UK, London, 1997).

32. Martin, R. C. Design Principles and Design Patterns. en (2000).

33. Kuffmeier, M., Haugbølle, T. & Nordlund, Å. Zoom-in Simulations of Protoplanetary Disks Starting from GMC Scales. *The Astrophysical Journal* **846,** 7 (Sept. 2017).

34. Popovas, A., Nordlund, Å., Ramsey, J. P. & Ormel, C. W. Pebble dynamics and accretion on to rocky planets – I. Adiabatic and convective models. en. *Monthly Notices of the Royal Astronomical Society* **479,** 5136–5156 (Oct. 2018).

35. Mathis, J. S., Rumpl, W. & Nordsieck, K. H. The size distribution of interstellar grains. *The Astrophysical Journal* **217,** 425–433 (Oct. 1977).

36. Jørgensen, J. K. *et al.* Binarity of a protostar affects the evolution of the disk and planets. *Nature* **606,** 272–275 (May 2022).

37. Tuhtan, V., Al-Belmpeisi, R., Bregning Christensen, M., Kuruwita, R. L. & Haugbølle, T. *Simulated Analogues I: apparent and physical evolution of young binary protostellar systems* July 2023.

38. Consolmagno, G., Britt, D. & Macke, R. The significance of meteorite density and porosity. en. *Geochemistry* **68,** 1–29 (Apr. 2008).

39. Lebreuilly, U., Commerçon, B. & Laibe, G. Protostellar collapse: the conditions to form dust-rich protoplanetary disks. en. *Astronomy & Astrophysics* **641,** A112 (Sept. 2020).

40. Bjerkeli, P., van der Wiel, M. H. D., Harsono, D., Ramsey, J. P. & Jørgensen, J. K. Resolved images of a protostellar outflow driven by an extended disk wind. en. *Nature* **540,** 406–409 (Dec. 2016).

41. Pakmor, R., Edelmann, P., Röpke, F. K. & Hillebrandt, W. Stellar GADGET: a smoothed particle hydrodynamics code for stellar astrophysics and its application to Type Ia supernovae from white dwarf mergers. *Monthly Notices of the Royal Astronomical Society* **424,** 2222–2231 (Aug. 2012).

42. *The Enzo Project*

43. team, T. F.-X. *Flash-X: A Multiphysics Scientific Software System* en.

44. Weinberger, R., Springel, V. & Pakmor, R. The Arepo public code release. *The Astrophysical Journal Supplement Series* **248,** 32 (June 2020).