A Task Scheduler for Astaroth, the Astrophysics Simulation Framework

Oskar Lappi 37146
MSc, Eng. Computer Engineering

Instructors: Johannes Pekkilä
Maarit Käpylä
Astroinformatics group
Aalto University

Supervisor: Jan Westerholm
Faculty of Science and Engineering
Åbo Akademi University
2021

Abstract

Computational scientists use numerical simulations performed by computers to create new knowledge about complex systems. To give an example, computational astrophysicists run large magnetohydrodynamics simulations to study the evolution of magnetic fields in stars [1].

Simulation workloads are growing in size as scientists are interested in studying systems of increasingly large scales and high resolutions. As a consequence, performance requirements for the underlying computing platforms are becoming tougher.

To meet these requirements, many high-performance computer clusters use GPUs together with high-speed networks to accelerate computation. The fluid dynamics library Astaroth is built for such GPU clusters and has demonstrated a 10x speedup when compared to the equivalent CPU-based *Pencil Code* library still used by computational physicists today [2].

This thesis has been done in coordination with the research team that created Astaroth and builds on their findings. My goal has been to create a new version of Astaroth that performs better at distributed scale.

Earlier versions of Astaroth perform well due to optimized compute kernels and a coarse pipeline that overlaps computation and communication. The work done for this thesis consists of a finer-grained dependency analysis and a task graph scheduler that exploits the analysis to schedule tasks dynamically.

The task scheduler improves the performance of Astaroth for realistic bandwidth-bound workloads by up to 25% and improves the overall scalability of Astaroth.

Keywords high-performance computing, computational fluid dynamics, finite difference methods, MPI, CUDA, dynamic scheduling, task-based parallelism, task scheduling

Acknowledgments

I would like to thank my supervisor Jan Westerholm for connecting me to the research team at Aalto, for being available for insightful discussions, and for some very thorough comments on a draft of this thesis. I would also like to thank Mats Aspnäs for valuable discussions on the nature of distributed performance.

I would like to extend my wholehearted gratitude to the research team: Maarit Käpylä, Johannes Pekkilä, Matthias Rheinhardt, and Miikka Väisälä. Thank you for this amazing opportunity to work on computational tools used for real research.

I thank Johannes for patiently guiding me through the implementation details of Astaroth and for always being available to chat about designs. Thank you also for providing some very clarifying comments on a very rough draft of this thesis.

I thank Maarit for providing support and for sharing with me an invaluable physics-readers perspective of the text.

I thank Miikka for valuable comments on the physics sections of this thesis and for providing an Astaroth users perspective during design discussions.

I thank Matthias for our many discussions on physics and the finer details of computational methods and for pointing me towards Landau and Lifshitz.

A special thanks also to Fredrik Robertsén from CSC for providing invaluable support on issues regarding CSC's Puhti cluster, which we could not have resolved without his help.

Contents

Introduction

Astaroth is a tool used by computational physicists to study multiphysics problems — problems in physics where multiple theoretical models apply. Specifically, Astaroth is being used by astrophysicists to study the flow of electrically conducting fluids, a field known as magnetohydrodynamics (MHD).

Many open questions in the field of astrophysical MHD concern phenomena relating to turbulent flows. E.g., in a 2021 paper [1], Väisälä et al. use Astaroth to study astrophysical dynamos, turbulent masses of astrophysical plasmas or liquids that generate the magnetic fields of stars and planets.

Predicting the behavior of turbulent fluids for long periods of time is very difficult, as evidenced by the unreliability of weather forecasts. Turbulent flow is mostly studied using statistics and computer simulations because analytical solutions are unstable and often intractable. However, even with computer simulations, studying turbulence is difficult.

Turbulent flow is chaotic; the fluid's velocity and concentration at two nearby points in space can differ greatly.In order to capture and model such almost noise-like, high-frequency information, a simulation requires high resolution data in both space and time.

To make matters more difficult, the problems studied using Astaroth, e.g., dynamo problems [1], concern the long-term behavior of large volumes of fluids, such as stars and planetary cores. These problems therefore require high resolution *and* large scales in space and time.

The above requirements translate into the following runtime characteristics of simulations: a large memory footprint, a large volume of computational work per iteration, and long runtimes. The runtime of large simulations is measured in weeks and they consume considerable computational resources.

The performance and scalability of these simulations therefore have a considerable impact on the productivity of computational scientists. They even impact the feasibility of certain simulations, especially for resource-constrained computational scientists or very large simulations. Because the simulations are so large, performance improvements also have a considerable effect on energy consumption. In short, any inadequate performance or scalability of computational tools are major concerns in the magnetohydrodynamics community.

Let us discuss performance, then. Single-core performance growth has slowed down in recent years. Semiconductor manufacturers are having trouble creating microprocessors that can dissipate the power needed for improved performance [3]. As performance gains based in hardware improvement are plateauing, parallelism and distribution have become new drivers of performance.

In high-performance computing (HPC), parallel processors, i.e., general-purpose graphics processing units (GPUs), have been used to improve application performance. GPUs outperform traditional central processing units (CPUs) by a significant margin, but existing software written for CPUs has to be rewritten to run on GPUs. Porting software from CPUs to GPUs is non-trivial due to fundamental changes in the execution model.

Further acceleration is possible by adding more GPUs to a node. The number of GPUs per node in HPC clusters varies, ranging from a typical 1-4 per node in a 1U server up to at least 20 or 32 per node in certain taller, specially built servers[1]. Clearly, there is a limit to how many GPUs can be crammed onto a single server. Once that limit is reached, the only way to add compute resources is to distribute the work to multiple nodes in a network.

A distributed program must communicate data between processes in a network. This is another step that requires a non-trivial rewrite of software, again due to a fundamental change in the execution model. HPC applications have traditionally used message passing, and in particular the Message Passing Interface standard (MPI), to communicate data between processes. MPI is a mature message passing standard with many robust implementations. Existing implementations include Open MPI [4], MPICH [5], and MVAPICH [6] among others.

The semantics of MPI are clear and easy to reason about, but MPI requires very explicit programming. MPI-based HPC applications sometimes use synchronous messaging patterns, even though asynchronous interfaces exist. This is likely because it is easier to reason about synchronous messages. MPI also encourages a synchronous way of writing applications by providing global synchronization mechanisms as well as synchronous interfaces for message dispatch and completion. This synchronization comes at a cost.

Aside from the execution model, the nature of performance also changes when running on multiple nodes: the network sometimes becomes the bottleneck. Network bandwidth in a cluster varies much more than the processing rate of a GPU or the bandwidth of memory buses, and so we see that at distributed scale a new problem appears: a bottleneck with unpredictable performance.

A large variance in the processing times of multiple parallel processes increases the delay between the moments that an average process and the slowest process reach the same synchronization barrier. In other words, an unpredictable bottleneck increases the

[1]E.g., the 5U G591-HS0 from Gigabyte Technology.

cost of global synchronization in a parallel program.

Before the work done for this thesis, Astaroth performed worse than expected when constrained by network bandwidth. This indicated that Astaroth was not fully equipped for dealing with the variable performance of the network.

I have worked on improving the distributed performance of Astaroth and engineering the performance of a system with an unpredictable bottleneck. The main contribution of this thesis is an asynchronous task scheduler which schedules compute work and communication work greedily, as data dependencies are satisfied. The task scheduler removes all global synchronizations and minimizes the scope of synchronizations to what is necessary, which eliminates unnecessary waiting time. The new version of Astaroth performs better than the baseline when constrained by communication and has better weak and strong scaling properties.

Previous work

My thesis builds on the work done by Johannes Peikkilä and others [2], [7], [8]. Pekkilä has already optimized the performance of compute kernels on the GPU. Pekkilä has also overlapped compute and communication work, but the order of execution is static in Pekkilä's implementation.

Pekkilä has used the roofline performance model by Williams et al. [9] to estimate the performance of Astaroth. Baseline Astaroth scales according to the model as long as the simulation has a high enough operational intensity for performance to be compute-bound. However, if a simulation exhibits low operational intensity, it performs worse than what the roofline model predicts. Operational intensity decreases as the number of processors increases, which means that Astaroth's scalability is constrained by its performance at low operational intensity.

To be able to reason about the runtime behavior of Astaroth, I have modeled the general class of program that Astaroth's runtime belongs to: iterative stencil loops (ISLs). The model for ISLs I present is a simplification of Andreas Schäfer's model from chapter 23 of the book *Grid-Computing: Eine Basistechnologie für Computational Science* [10][2].

The book [10], compiled by Dietmar Fey, is also a good introduction to the topics of computational science covered by this thesis. It includes information about Runge-Kutta schemes and finite-difference methods, as well as information about execution models for parallel processing and message passing.

My treatment adds a task graph scheduler to Astaroth. A good introduction to scheduling is Coffman and Bruno's book *Computer and job-shop scheduling theory* [12].

[2] Schäfer is the project lead of LibGeoDecomp [11], another stencil library.

Several task scheduling systems have been built for HPC before and they have influenced the design presented in this thesis. Legion [13] uses data regions to generate dependency graphs which drive task scheduling. The Open-Community Runtime (OCR) also uses data dependencies to schedule tasks [14]. Another task scheduling system, HPX, has even been applied to the astrophysical fluid simulation framework Octo-Tiger [15] by Pfander et al.. All three of these systems — Legion, OCR, HPX — inspired the design of my task scheduler in some way.

The systems mentioned above are capable of much more than the lightweight task scheduler presented in this thesis. For a more in-depth discussion on task-based systems, I refer the reader to Thoman et al.'s 2018 article: "A taxonomy of task-based parallel programming for high-performance computing" [16].

Structure of this thesis

The thesis consists of three parts, divided into 10 chapters. The first two parts contain background information (and some analysis) and the third part contains practical work and results.

Part I, *Computational Physics Background*, consists of four chapters in which I review the scientific and computational context of this work. In chapters 1 and 2, the theory and governing equations of fluid dynamics and magnetohydrodynamics are presented on a high level and chapter 3 explains the methods employed in Astaroth to solve the equations numerically.

Chapter 4 describes a model for iterative stencil loops. The model is not very rigorous but provides a good enough basis for reasoning about distributed stencil solvers. Of particular interest is the analysis of the structure of dependency in a solver that uses symmetric stencils. The data dependency analysis conducted in this part is the basis for the design of the task scheduler presented in part III.

Part II, *Technological Background*, covers the computing technologies that enable Astaroth to run efficiently. Chapter 5 concerns the abstract communication model of MPI, and chapter 6 concerns the concrete communication stack used by our HPC environment.

Part III, *Designing a Task Scheduler for Astaroth*, contains the practical work and performance results. In chapter 7, I analyze baseline Astaroth's runtime and implementation details. In chapter 8, I present the integrated task scheduler I have designed and implemented for Astaroth. In chapter 9, I compare the performance of the two versions. Finally in chapter 10, I discuss my results, limitations of the task scheduler, and future work.

Part I

Computational Physics Background

Astaroth is a tool for computational scientists studying problems in astrophysical fluid dynamics. The framework of computational fluid dynamics is rather complex — and this complexity is reflected in Astaroth. Before examining Astaroth, then, one should know enough about the relevant theory and numerical techniques that create this complexity.

I will first describe the scientific context in which Astaroth is used. The relevant fields of study are fluid dynamics (chapter 1) and magnetohydrodynamics (chapter 2).

Chapter 1 on fluid dynamics covers the fundamental equations of fluid dynamics, why we need numerical methods to solve fluid dynamics problems, and why turbulence in particular makes it difficult to solve fluid dynamics problems.

Chapter 2 on magnetohydrodynamics (MHD) begins with a section on dynamo theory, the particular context that Astaroth has been used in. I then move on to the governing equations of MHD and how they are implemented in Astaroth.

The numerical methods used by Astaroth to approximate differentials in the governing equations are then presented in **chapter 3**.

Armed with numerical methods, Astaroth simulates fluids in a discretized domain in discrete time with an iterative stencil loop (ISL). In **chapter 4**, I have modeled the execution of ISLs in a distributed computing environment. The model is used to reason about the execution of Astaroth in general and the structure of dependencies between compute and communication tasks in particular.

1
Fluid Dynamics

Let me begin by describing, in broad strokes, the theoretical framework of fluid dynamics. Fluid dynamics is an old field of study, still very active and applicable to a great many domains of science and engineering. For the interested, there are plenty of textbooks on the subject. One such popular textbook is volume 6, *Fluid Mechanics*, of Landau and Lifshitz's book series Course of Theoretical Physics. The 1987 edition of *Fluid Mechanics* [17] is the primary source for this section.

Because so much of our world consists of fluids, fluid dynamics is applied in many domains. Some examples follow. Meteorological models used for weather forecasting are based on fluid dynamics, as are the models used to study the climate and the oceans. Fluid dynamics helps engineers understand fuel flow through a pipeline. It helps medical doctors understand blood flow through an artery. Combustion engines, airplanes, wind farms, and nuclear power plants all need to operate within their fluid environment in specific ways to fulfill their function. Fluid dynamics provides us with a way to safely and accurately simulate the operation of these devices, without the need for expensive and potentially unsafe real-world tests.

The real world consists of particulate matter, but matter can be modeled in many ways. At microscopic scales, particles are directly used to model massive bodies. At macroscopic scales, however, massive bodies are typically modeled as continuous media.

This framework is called *continuum mechanics* and fluid dynamics is a part of it. In continuum mechanics, a massive body is modeled as a continuum — a continuous volume where each infinitesimal part has a mass intensity (called density).

Continua are either solids or fluids, depending on their properties. A solid is a structured body, with a defined rest shape. It does not deform unless significant force is applied. A fluid, on the other hand, has no defined rest shape and deforms easily when forces apply. Liquids are fluids, as are gases and plasmas.

Solids and fluids are studied in their respective branches of continuum mechanics: *solid mechanics* and *fluid mechanics*. Each branch is split further into the study of bodies at rest — *statics* — or bodies in motion — *dynamics*. Fluid dynamics, then, is the study of the motion of fluid bodies modeled as continua.

Symbol	Quantity
ρ	density
$\vec{u}$	flow velocity
p	pressure
η	dynamic viscosity
ζ	bulk viscosity
ε	internal energy per unit mass
$w = \varepsilon + \frac{p}{\rho}$	enthalpy
κ	thermal conductivity
T	temperature
s	entropy

Table 1.1: Physical quantities

The main idea of fluid dynamics can be described as follows: the volumetric mass density ρ is a scalar field defined over the space that the fluid inhabits (typically in $\mathbb{R}^3$). Mass flows along a velocity vector field $\vec{u}$ in the fluid, creating the mass flux $\rho\vec{u}$. Thermodynamic properties (such as pressure p, temperature T, and viscosities η, ζ) also interact with the mass flux.

The properties of a fluid are either directly related in a *state equation* or related through their derivatives in a *transport equation*. The set of transport equations that describe the flow of a fluid are called the *fundamental equations of fluid dynamics*. Together with the state equations, they form a system of equations that describes the state of a fluid for each point in space and time [17, p. 1].

1.1 The fundamental equations of fluid dynamics

The fundamental equations of fluid dynamics express conservation laws in the form of transport equations, specifically the conservation of mass, momentum, and energy.

The mass transport equation describes how density changes due to mass flow.

The momentum transport equation describes how forces cause the mass flow to shift.

Finally, the energy equation describes how the thermodynamic properties of the fluid interact with the mass flow to transform energy from one form into another.

I will present these equations one by one and explain them in short detail. The equations have all been written in a local form, describing the state at some specific infinitesimal *control volume* in the fluid, but they apply globally, at all points in the continuum.

The equations are all in $\mathbb{R}^3$ and use the mathematical notation in table 1.2.

Notation	Description
$\mathrm{grad}(a) = \nabla a = \left(\frac{\partial a}{\partial x}, \frac{\partial a}{\partial y}, \frac{\partial a}{\partial z}\right)$	the gradient operator
$\mathrm{div}(\vec{a}) = \nabla \cdot \vec{a} = \frac{\partial a_x}{\partial x} + \frac{\partial a_y}{\partial y} + \frac{\partial a_z}{\partial z}$	the divergence operator
$\mathrm{curl}(\vec{a}) = \nabla \times \vec{a} = \begin{vmatrix} \hat{\mathbf{i}} & \hat{\mathbf{j}} & \hat{\mathbf{k}} \\ \frac{\partial}{\partial x} & \frac{\partial}{\partial y} & \frac{\partial}{\partial z} \\ a_x & a_y & a_z \end{vmatrix}$	the curl operator
$\Delta a = \nabla \cdot \nabla a = \frac{\partial^2 a}{\partial x^2} + \frac{\partial^2 a}{\partial y^2} + \frac{\partial^2 a}{\partial z^2}$	the scalar Laplacian
$\Delta \vec{a} = (\Delta a_x, \Delta a_y, \Delta a_z)$	the vector Laplacian
$\mathbf{I} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$	the identity tensor

Table 1.2: Mathematical notation

The mass transport equation

$$\frac{\partial \rho}{\partial t} = -\nabla \cdot (\rho \vec{u}) \tag{1.1}$$

Equation 1.1 is the mass transport equation, also known as the continuity equation. It expresses the principle of mass conservation as follows: the rate of change in local fluid density, $\frac{\partial \rho}{\partial t}$, is equal to the mass flux into that local region, $-\nabla \cdot (\rho \vec{u})$. In other words, any change in fluid density in a region is due to mass flux into or out of that region [17, pp. 1-2].

The Navier-Stokes equation (the momentum transport equation)

$$\rho \left[\frac{\partial \vec{u}}{\partial t} + (\vec{u} \cdot \nabla)\vec{u}\right] = -\nabla p + \nabla \cdot \left[\eta \left(\nabla \vec{u} + (\nabla \vec{u})^{\mathrm{T}} - \tfrac{2}{3}(\nabla \cdot \vec{u})\mathbf{I}\right) + \zeta (\nabla \cdot \vec{u})\mathbf{I}\right] + \vec{f} \tag{1.2}$$

Equation 1.2 is the Navier-Stokes equation (NSE) in vector form. The equation expresses the principle of conservation of momentum as applied to Newtonian fluids: the rate of change of momentum in a fluid is equal to the forces acting on the fluid.

In the NSE, the rate of velocity change per control volume is described by the *material derivative* $\left[\frac{\partial \vec{u}}{\partial t} + (\vec{u}\cdot\nabla)\vec{u}\right]$ on the left-hand side of the equation. The material derivative incorporates both the local momentum rate of change, $\frac{\partial \vec{u}}{\partial t}$, and the rate of change due to momentum advection, $(\vec{u}\cdot\nabla)\vec{u}$. When multiplied with the density, the left-hand side becomes an expression for the rate of momentum change per control volume [17, pp. 3].

The right-hand side of the equation is the sum of forces acting on the control volume. Always present are forces created by pressure,

$$-\nabla p$$

and viscous stresses

$$\nabla \cdot \left[\eta\left(\nabla\vec{u} + (\nabla\vec{u})^{\mathrm{T}} - \tfrac{2}{3}(\nabla\cdot\vec{u})\mathbf{I}\right) + \zeta(\nabla\cdot\vec{u})\mathbf{I}\right].$$

The two coefficients η and ζ are the viscosity coefficients: η is called the *dynamic viscosity* and ζ the *bulk viscosity* or *second viscosity*[17, p. 45].

Other forces are here represented as a single term, $\vec{f}$. What $\vec{f}$ contains will depend on the problem. $\vec{f}$ can include external forces originating from outside the fluid, e.g., gravity, but $\vec{f}$ can also include other body forces, e.g., the Lorentz force in a magnetohydrodynamics problem.

The NSE can be simplified if one makes certain assumptions about the properties of the fluid and its environment. Under these assumptions, certain variables can be made constant, and certain terms may even disappear entirely. One typical simplification is to assume that the viscosity coefficients η and ζ are constant, which yields equation 1.2b [17, pp. 45].

The NSE with constant viscosity coefficients

$$\rho\left[\frac{\partial \vec{u}}{\partial t} + (\vec{u}\cdot\nabla)\vec{u}\right] = -\nabla p + \eta\Delta\vec{u} + \left(\zeta + \frac{1}{3}\eta\right)\nabla(\nabla\cdot\vec{u}) + \vec{f} \tag{1.2b}$$

Sometimes the *kinematic* viscosity $\nu = \frac{\eta}{\rho}$ is assumed to be constant instead, as we will see in the section on magnetohydrodynamics[1].

[1]The kinematic viscosity ν is denoted by the Greek letter *nu*, not the Latin letter *v*.

The energy transport equation

$$\frac{\partial}{\partial t}\left[\rho\left(\tfrac{u^2}{2}+\varepsilon\right)\right] = -\nabla\cdot\left\{\rho\vec{u}\left(\tfrac{u^2}{2}+w\right) - \kappa\nabla T \right.$$
$$\left. - \vec{u}\left[\eta\left(\nabla\vec{u}+(\nabla\vec{u})^{\mathrm{T}}-\tfrac{2}{3}(\nabla\cdot\vec{u})\mathbf{I}\right)+\zeta(\nabla\cdot\vec{u})\mathbf{I}\right]\right\} \quad (1.3)$$

Equation 1.3 is the transport equation for energy, where $u^2 = \vec{u}\cdot\vec{u}$. It expresses the principle of conservation of *total energy*. This means that the sum of kinetic energy $\frac{1}{2}\rho u^2$ and internal energy $\rho\varepsilon$ is conserved, even though the energies are not conserved individually.

In the energy equation, the left-hand side is the combined influx of kinetic and internal energies and the right-hand side is the sum of energy fluxes into the control volume and work. The right-hand side consists of the convective energy flux,

$$-\nabla\cdot\left(\rho\vec{u}(\tfrac{u^2}{2}+w)\right)$$

the heat flux due to thermal conduction,

$$\nabla\cdot(\kappa\nabla T)$$

and the work done by viscous stresses

$$\nabla\cdot\left\{\vec{u}\left[\eta\left(\nabla\vec{u}+(\nabla\vec{u})^{\mathrm{T}}-\tfrac{2}{3}(\nabla\cdot\vec{u})\mathbf{I}\right)+\zeta(\nabla\cdot\vec{u})\mathbf{I}\right]\right\}.$$

The heat transfer equation

$$\rho T\left[\frac{\partial s}{\partial t}+\vec{u}\cdot\nabla s\right] = \nabla\cdot(\kappa\nabla T)+\tfrac{1}{2}\eta\left[\nabla\vec{u}+(\nabla\vec{u})^{\mathrm{T}}-\tfrac{2}{3}(\nabla\cdot\vec{u})\mathbf{I}\right]^2+\zeta(\nabla\cdot\vec{u})^2 \quad (1.4)$$

Equation 1.4 is the heat transfer equation. This equation is an alternative to the energy equation which, according to Landau and Lifshitz, is more convenient to work with. The heat transfer equation can be obtained by subtracting the momentum equation from the energy equation [17, pp. 193-194].

The left-hand side of the heat transfer equation is an expression for the heat gain in a control volume. Here, s denotes entropy. The right-hand side of the equation consists of the familiar thermal conduction along the temperature gradient,

$$\nabla \cdot (\kappa \nabla T)\ ,$$

and the heat from viscous dissipation,

$$\tfrac{1}{2}\eta \left[\nabla\vec{u} + (\nabla\vec{u})^{\mathrm{T}} - \tfrac{2}{3}(\nabla\cdot\vec{u})\,\mathbf{I}\right]^2 + \zeta\,(\nabla\cdot\vec{u})^2\ .$$

1.2 Challenges in solving fluid dynamics problems

Even though the theory behind fluid dynamics is reasonably simple, the application of it is not straightforward. The fundamental equations all have a local form, but they contain temporal and spatial derivatives that are functions of values at neighboring points. When considering the entire domain, the neighborhoods connect and make all local solutions dependent on one another. Any solution to a fluid dynamics problem must therefore be a complete solution, describing the state of the fluid at each point in space over the entire period of time being investigated. A solution is also a full simulation of the fluid.

Analytical solutions have proven to be at least practically intractable except for special cases. Analytical methods can still be useful; however, solutions to more complex cases cannot be analytical. Instead, approximate solutions are obtained through numerical methods, which involves discretizing the continuum (e.g., into a regular grid) and finding approximations for the differentials in that framework.

Numerical solutions are computationally intense and still involve complex dependencies between fields. This is why fluid dynamics is often studied using tools such as Astaroth — number-crunching programs that run on powerful computers. The field of number-crunching computer programs that simulate fluid flow is aptly called computational fluid dynamics (CFD).

Because solutions are difficult to compute, the fundamental equations of fluid dynamics are often simplified by making certain assumptions (e.g., equation 1.2b). Two examples of simplified models are those for incompressible flow and laminar flow, but simplified models are only reasonable when the flow has certain characteristics.

Flows can be characterized by certain dimensionless quantities, which relate certain physical quantities to each other. Perhaps the most important ones are the Reynolds number and the Mach number, which characterize the degree of turbulence and the degree of compressibility of a flow, respectively.

The values of these dimensionless quantities determine which simplifying assumptions can be made. As hinted at earlier, laminar flow can be modeled more simply than turbulent flow, incompressible flow more simply than compressible flow. Of course, this also means that the more interesting problems in fluid dynamics concern turbulent and compressible flows. Astaroth is used to solve these types of problems.

1.3 Difficulty with turbulent flows

In fact, physicists have failed to formulate a complete and coherent theory of turbulent flow. In his 2015 book *Turbulence: An Introduction for Scientists and Engineers*, Davidson states that it is generally considered impossible to unite all the different phenomena of turbulence under one theory. For a multitude of turbulence problems, there is a multitude of theories [18, p. 30].

"But", the optimist says, "even though a satisfying theory doesn't exist, we still have computational fluid dynamics! Surely we can simply simulate any problem to get a good enough approximation for, say, an engineering application?"

It is certainly possible to simulate turbulent fluid flow. The problem is just how much, or rather how little, we can simulate. An example by Davidson illustrates the problem [18, p. 30]:

> Now suppose we wished to simulate this flow for two minutes and that the speed and direction of the wind changes during this time. Then we would have to compute flow structures (in the mean flow and the turbulence) whose dimensions varied from 0.1 mm to 100 m and whose characteristic timescales span the range 10^{-3} s to 100 s! Such a monumental calculation is well beyond the capacity of any computer which exists today.

Davidson is discussing direct numerical simulation (DNS). This is the most thorough approach, where turbulent fluid flow is directly simulated down to the scales at which energy dissipates. Essentially, in order to usefully model turbulent fluid flow, so that one can study its effects, one needs to model the fluid at a very high resolution in both space and time[2].

All is not lost, however. The performance of computers is steadily rising. Even though we will always be limited in how much turbulent fluid flow we can simulate, the limits have been pushed far enough to enable interesting research.

[2]See §7.1 of Davidson's turbulence book [18] for the precise reason why this is.

2
Magnetohydrodynamics

How exactly do planets and stars generate their magnetic fields? This is an open question in astrophysics. The generally agreed upon theory is that masses of swirling, electrically conducting fluids in the center of these bodies act as a self-excited dynamo, transforming the kinetic energy of the fluid flow into magnetic energy through electromagnetic induction [19][20].

2.1 Dynamo Theory

The general mechanism of the self-excited dynamo is a mutual amplification of two magnetic fields: one poloidal and one toroidal. The idea was first put forth by Eugene Parker in his 1955 paper "Hydromagnetic Dynamo Models" [21].

According to the theory, the amplification of the poloidal field is due to the Coriolis force[20, pp. 91-93]. The amplification of the toroidal field is due to either the Coriolis force or a differential rotation of the fluids (faster flow near the equator). The precise mechanism that generates the field is not fully understood, but it is thought to contain at least some combination of the α-effect (Coriolis force) and the Ω-effect (differential rotation) [20, p. 108].

The dynamo model is complex and provides multiple mechanisms that combine to produce a magnetic field. One phenomenon whose role isn't fully understood is that of small-scale dynamos (SSDs) which generate smaller, irregular magnetic fields [20, p. 58]. SSDs contrast with large-scale dynamos (LSDs) that are thought to be the main mechanism that generates the overall magnetic fields of astrophysical bodies [20, pp. 89-90].

Väisälä et al. have studied the interaction of SSDs and LSDs, using Astaroth to run direct numerical simulations [1]. Väisälä's simulations were run on a single computer with four GPUs, in discretized domains with dimensions 512^3. Simulations at this large a scale have only become feasible in recent years [20, p. 6]; as mentioned in the previous chapter, direct numerical simulations of turbulence are computationally demanding.

Astaroth has been built for computational research of phenomena such as this, where the theories of fluid dynamics and electromagnetics meet, a discipline known as magnetohydrodynamics (MHD). The paper [1] is the only published physics article containing results produced by Astaroth and the authoring group comprises the only physicists to have used Astaroth for research. The research interests of this core Astaroth user group are largely related to astrophysical magnetism. It is therefore safe to say that, as things stand, MHD is the primary use-case of Astaroth.

2.2 Governing MHD equations

Symbol	Quantity
$\nu = \frac{\eta}{\rho}$	kinematic viscosity
$\zeta_\nu = \frac{\zeta}{\rho}$	kinematic bulk viscosity
c_p	specific heat at constant pressure
c_v	specific heat at constant volume
c_s	adiabatic speed of sound
$\gamma = \frac{c_p}{c_v}$	adiabatic index
$\vec{J}$	electric current density
$\vec{B}$	magnetic flux density
$\vec{A}$	magnetic vector potential
μ_0	magnetic vacuum permeability
σ	electrical conductivity
$\lambda = \frac{1}{\mu_0 \sigma}$	magnetic diffusivity

Table 2.1: Additional physical quantities used in this chapter

The MHD simulations I've run for this thesis use four governing equations: the mass, momentum, and entropy equations from fluid dynamics and the induction equation from electrodynamics [1].

The predecessor of Astaroth is the *Pencil Code*, a computational MHD tool written in Fortran [22]. The governing equations used in Astaroth have the same form as the ones used in the Pencil Code and they can be found in the Pencil Code manual [23, pp. 61-62], with additional information in an article by Brandenburg [24].

The MHD equations use some new physical quantities (table 2.1) in addition to the ones already presented in the previous section (table 1.1). More thorough discussions on

[1]The GPU kernels that implement the equations have been written by Miikka Väisälä in Astaroth's domain-specific language.

the MHD equations can be found in Miikka Väisälä's PhD thesis [25] and Davidson's book *An Introduction to Magnetohydrodynamics* [26].

Fluid dynamics equations used in Astaroth/Pencil Code

The fluid dynamics equations in the Pencil Code and Astaroth differ somewhat from the ones presented in the previous section. The major difference is a change of variables from pressure p and density ρ to entropy s and *logarithmic* density $\ln\rho$.

A logarithm of a variable expresses a wider dynamic range of the variable than the variable itself. Similarly, entropy also expresses a wider dynamic range in pressure than pressure itself[2]. This makes the modified equations more suitable for simulating turbulent fluid flows, where variations in density and pressure are large [24, pp. 2-3]. I will now briefly explain how these changes affect the governing equations of fluid dynamics, beginning with the mass equation.

The mass transport equation

We can make the mass equation use logarithmic density by a simple change of variables on the left-hand side. Through the chain rule, the right-hand side is multiplied by the derivative of $\frac{\partial \ln\rho}{\partial\rho} = \frac{1}{\rho}$, which conveniently removes the density from the right-hand side.

The mass transport equation as used in Astaroth/Pencil Code

$$\frac{\partial \ln\rho}{\partial t} = -\nabla \cdot \vec{u} \tag{2.1}$$

The Navier-Stokes equation

In order to replace the linear pressure term in the Navier-Stokes equation, *dynamic* viscosity coefficients are replaced with *kinematic* viscosity coefficients. Using the constant viscosity NSE as a base (eq. 1.2b), constant dynamic viscosity η and bulk viscosity ζ have been replaced by constant kinematic viscosity $\nu = \frac{\eta}{\rho}$ and kinematic bulk viscosity $\zeta_\nu = \frac{\zeta}{\rho}$ [24, p. 3]. With this change, both sides can be divided by the density. This has the effect that the left-hand side now describes a change in momentum per unit mass instead of unit volume.

The individual terms have also been altered slightly. In the viscous force term, the constant ν can be moved outside the divergence operation — just as a constant η could

[2]Entropy s can be calculated as a linear combination of $\ln p$ and $\ln\rho$.

— but the density ρ cannot, because it is a function of x,y, and z. This yields an additional viscosity term which is not present when assuming constant dynamic viscosity.

The pressure term has changed the most. Remembering that the whole equation is divided by ρ, the pressure term $-\frac{1}{\rho}\nabla p$ becomes [24, eq. (7)]:

$$-c_s^2\left(\nabla \ln\rho + \nabla\frac{s}{c_p}\right) .$$

Here, c_s^2 is the square of the adiabatic speed of sound, which is a function of logarithmic density and entropy [24, eq. (8)]:

$$c_s^2 = \frac{\gamma p_0}{\rho_0}\exp\left[\frac{\gamma}{c_p}s + (\gamma-1)\ln\left(\frac{\rho}{\rho_0}\right)\right]$$

where p_0, ρ_0, γ, and c_p are constants. The entire derivation of the pressure term can be found on page 3 of [24].

Because we are working with electrodynamics, we also have an external force: the Lorentz force, $\vec{J}\times\vec{B}$, which is substituted for $\vec{f}$. For this term, unfortunately, one cannot replace the density with its logarithm. The density must be calculated from its logarithm.

All these changes lead to the form of the Navier-Stokes equation used in the Pencil Code and Astaroth, equation 2.2.

The Navier-Stokes equation as used in Astaroth/Pencil Code

$$\begin{aligned}\frac{\partial\vec{u}}{\partial t} + (\vec{u}\cdot\nabla)\vec{u} = &-\overbrace{c_s^2\nabla\left(\frac{s}{c_p}+\ln\rho\right)}^{\frac{\nabla p}{\rho}\text{ in terms of }\ln\rho\text{ and }s} + \nu\Delta\vec{u} + \left(\zeta_\nu + \frac{1}{3}\nu\right)\nabla(\nabla\cdot\vec{u}) \\ &+ \underbrace{\nu\left[\nabla\vec{u} + (\nabla\vec{u})^{\mathrm{T}} - \tfrac{2}{3}(\nabla\cdot\vec{u})\mathbf{I}\right]\cdot\nabla\ln\rho}_{\text{New viscous term due to kinematic viscosity}} + \underbrace{\frac{\vec{J}\times\vec{B}}{\rho}}_{\text{Lorentz force}} \end{aligned} \tag{2.2}$$

The heat transfer equation

Naturally the assumption of constant kinematic viscosities applies to all equations, including the heat transfer equation. The dynamic viscosities are therefore replaced with products of kinematic viscosity and density.

The heat conduction term $\nabla \cdot (\kappa \nabla T)$ can be expressed using logarithmic density and temperature using the following relation [23], [27, acc/mhd_solver:507]:

$$\frac{\nabla \cdot (\kappa \nabla T)}{\rho T} = \frac{K}{\rho}\Bigg[\left(\frac{\gamma \Delta s}{c_p} + (\gamma - 1)\Delta \ln \rho\right) \\ + \left(\frac{\gamma \nabla s}{c_p} + (\gamma - 1)\nabla \ln \rho\right) \cdot \left(\gamma\left(\frac{\nabla s}{c_p} + \nabla \ln \rho\right) + \nabla \ln \tfrac{K}{\rho c_p}\right)\Bigg].$$

If K is constant, then $\nabla \ln \frac{K}{\rho c_p} = -\nabla \ln \rho$, and the expression simplifies to:

$$\frac{\nabla \cdot (\kappa \nabla T)}{\rho T} = \frac{K}{\rho}\left[\left(\frac{\gamma \Delta s}{c_p} + (\gamma - 1)\Delta \ln \rho\right) + \left(\frac{\gamma \nabla s}{c_p} + (\gamma - 1)\nabla \ln \rho\right)^2\right]$$

where the squared vector term again denotes a dot product.

The equation also obtains three new terms. $\lambda \mu_0 \vec{J}^2$ describes resistive heating, while $\mathcal{H}$ and $\mathcal{C}$ are explicit heating and cooling terms, which add up to a constant heating or cooling effect.

The heat transfer equation as used in Astaroth/Pencil Code

$$\rho T\left[\frac{\partial s}{\partial t} + \vec{u} \cdot \nabla s\right] = \mathcal{H} - \mathcal{C} + \nabla \cdot (\kappa \nabla T) + \lambda \mu_0 \vec{J}^2 \\ + \tfrac{1}{2}\nu\rho\left[\nabla \vec{u} + (\nabla \vec{u})^{\mathrm{T}} - \tfrac{2}{3}(\nabla \cdot \vec{u})\mathbf{I}\right]^2 + \zeta_\nu \rho (\nabla \cdot \vec{u})^2 \quad (2.3)$$

Electrodynamics, the induction equation

One transport equation from electrodynamics is added: the induction equation. The induction equation describes the rate of change of the magnetic vector potential $\vec{A}$. We start from Ohm's Law:

$$\vec{E} = \tfrac{1}{\sigma}\vec{J} - \vec{u} \times \vec{B}\,,$$

and assume the Weyl gauge, which simplifies things [24, p. 27-29]. For the Weyl gauge, the relation $E = -\frac{\partial A}{\partial t}$ applies. We then obtain the induction equation, as used in Astaroth and the Pencil Code.

The induction equation as used in Astaroth/Pencil Code

$$\frac{\partial \vec{A}}{\partial t} = \vec{u} \times \vec{B} - \lambda \mu_0 \vec{J} \tag{2.4a}$$

The magnetic flux density (or magnetic induction) can then be obtained by the relation $\vec{B} = \nabla \times \vec{A}$.

2.3 Advantages and disadvantages of non-conservative equations

These equations do have a drawback: due to the change of variables, they are non-conservative. This means that when all differentials in the equations are replaced with approximations, the equations no longer conserve the conserved quantities exactly. To avoid errors, simulations have to be monitored for violations of conservation laws [25, p. 8].

However, in practice, the non-conservative equations can sometimes be more accurate than their conservative counterparts in certain cases. This is due to their greater dynamic range [24, pp. 2-3],[25, p. 8], as mentioned earlier.

3
Numerical Methods used by Astaroth

Now that the governing equations of our simulations have been introduced, it is time to move on to the numerical methods used to solve them. As touched upon earlier, the domain is discretized into a regular grid and the differential equations are applied to every point in the grid. Time must also be discretized and the grid then solved for each timestep.

To solve the equations in this discretized domain, we need two things.

First, we need a way to approximate the spatial differentials on the right-hand sides of the equations and solve the grid at one point in time. For this Astaroth uses a finite difference method (**section 3.1**).

Second, we need a way to integrate the time differentials on the left-hand side in discrete time steps. For this, Astaroth uses a three-step Runge-Kutta method (**section 3.2**).

3.1 Finite difference methods

Finite difference methods use *difference quotients* to approximate differentials. Difference quotients approximate the derivative at a point by sampling other, nearby points. This makes the method well suited for regular grids, where points are distributed along cardinal axes at uniform distance from each other.

3.1.1 Finite difference approximations for first order derivatives

There are three types of difference quotients that can serve as approximations of a derivative, called *forward*, *backward*, and *central* finite difference approximations. They can be derived from the Taylor series expansion of a function.

This approach also yields the order of accuracy, expressed here using $\mathcal{O}$-notation. We say that the methods are *nth-order accurate* or *nth-order approximations* when the error is $\mathcal{O}(h^n)$, where h is the distance between points. We will find that central differences have the highest accuracy of the three types of difference quotient.

Let us begin with first-order derivatives. The Taylor series expansion of a function f about a point x can be written:

$$f(x+h) = \sum_{k=0}^{\infty} \left(\frac{f^{(k)}(x)}{k!} h^k \right)$$

where $h > 0$. By manipulating this expression, we arrive at several difference quotients on the left-hand side that are equal to the derivative $f'(x)$ and an error term.

We arrive at the forward difference by subtracting $f(x)$ and dividing by h. The forward difference approximation samples f at x and $x+h$:

Two-point forward difference approximation

$$\frac{f(x+h)-f(x)}{h} = f'(x) + \underbrace{\sum_{k=2}^{\infty} \left(\frac{f^{(k)}(x)}{k!} h^{k-1} \right)}_{\mathcal{O}(h)} \tag{3.1}$$

For the backwards difference, we expand $f(x-h)$ instead of $f(x+h)$ and perform the same operation. The backward difference approximation samples f at t and $t-h$:

Two-point backward difference approximation

$$\frac{f(x)-f(x-h)}{h} = f'(x) + \underbrace{\sum_{k=2}^{\infty} \left(\frac{f^{(k)}(x)}{k!} (-h)^{k-1} \right)}_{\mathcal{O}(h)} \tag{3.2}$$

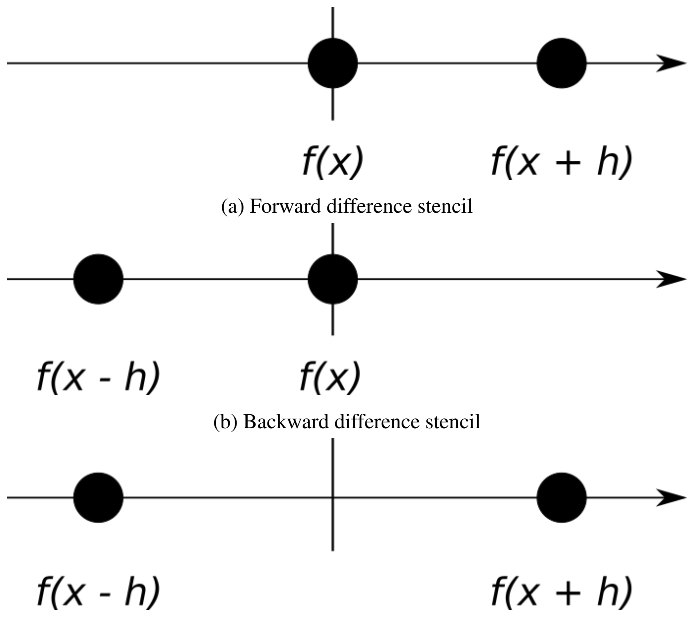
For the central difference, we take the difference of the two Taylor expansions and find $f'(x)$ in each. Because certain terms cancel, this approximation is more accurate.

The central difference approximation samples f at $x-h$ and $x+h$:

Two-point central difference approximation

$$\begin{aligned}\frac{f(x+h)-f(x-h)}{2h} &= \frac{1}{2h}\left[\sum_{k=0}^{\infty}\left(\frac{f^{(k)}(x)}{k!}h^k\right)-\sum_{k=0}^{\infty}\left(\frac{f^{(k)}(x)}{k!}(-h)^k\right)\right] \\ &= \sum_{m=0}^{\infty}\left(\frac{f^{(2m+1)}(x)}{(2m+1)!}h^{2m}\right) \\ &= f'(x)+\underbrace{\sum_{m=1}^{\infty}\left(\frac{f^{(2m+1)}(x)}{(2m+1)!}h^{2m}\right)}_{\mathcal{O}(h^2)}\end{aligned} \tag{3.3}$$

For two-point approximations, we see that forward and backward difference methods are first-order accurate because the truncation error is $\mathcal{O}(h)$, while the central difference method is second-order accurate since the truncation error is $\mathcal{O}(h^2)$. Astaroth only uses central difference approximations.



(a) Forward difference stencil

(b) Backward difference stencil

(c) Central difference stencil

Figure 3.1: Stencils of two-point first derivative difference quotients

The sample points used as inputs for approximations form the *stencil* that will define the data access pattern of Astaroth. The stencils for two-point first derivative approximations are shown in figure 3.1.

3.1.2 Obtaining higher-order approximations for first derivatives

Higher-order approximations can be obtained by several methods. The central difference approximations used in Astaroth are the same as the ones used in the Pencil Code and have been worked out by Brandenburg [24, p. 57, appendix A]. Methods for programmatically generating coefficients are also presented in [28, § 1.5] and [29]. In appendix A of this thesis, I also demonstrate a straightforward method for working out the coefficients using Richardson extrapolation.

Central difference approximations of first derivatives used in Astaroth

Astaroth supports 2nd, 4th, 6th, and 8th-order central difference approximations (eqs. 3.4-3.7). The approximations are used for spatial derivatives in some direction x with a grid resolution of $h = \delta x$. Approximations assume a unit step size δx, which is to say that they are only compatible with grids of constant cell size.

$$s_r^-(x) = f(x + r\delta x) - f(x - r\delta x)$$

$\underline{2nd \text{ order}}$

$$\frac{\partial f}{\partial x} = \frac{s_1^-(x)}{2\delta x} + \mathcal{O}(\delta x^2) \tag{3.4}$$

$\underline{4th \text{ order}}$

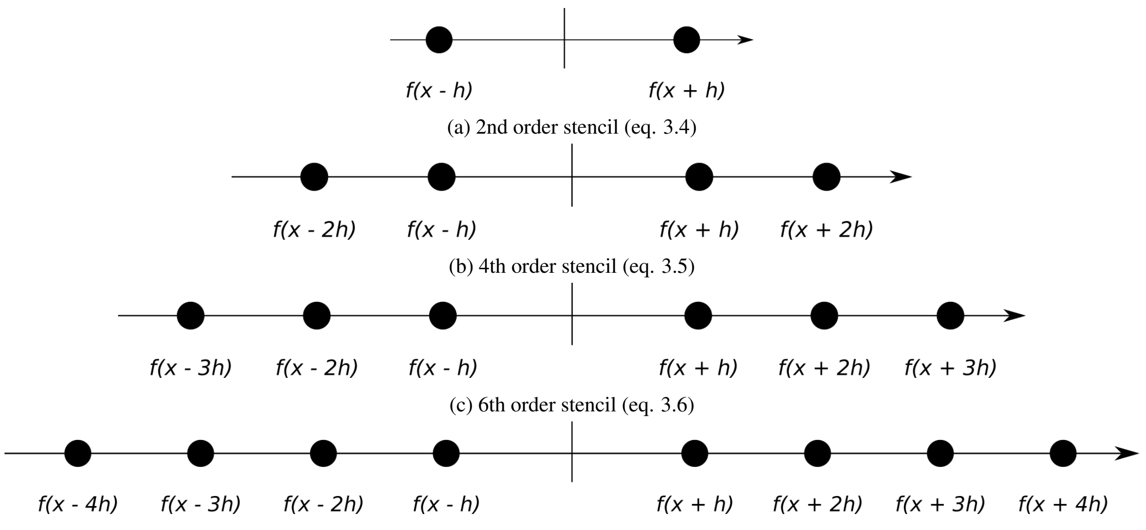
$$\frac{\partial f}{\partial x} = \frac{8s_1^-(x) - s_2^-(x)}{12\delta x} + \mathcal{O}(\delta x^4) \tag{3.5}$$

$\underline{6th \text{ order}}$

$$\frac{\partial f}{\partial x} = \frac{45s_1^-(x) - 9s_2^-(x) + s_3^-(x)}{60\delta x} + \mathcal{O}(\delta x^6) \tag{3.6}$$

$\underline{8th \text{ order}}$

$$\frac{\partial f}{\partial x} = \frac{672s_1^-(x) - 168s_2^-(x) + 32s_3^-(x) - 3s_4^-(x)}{840\delta x} + \mathcal{O}(\delta x^8) \tag{3.7}$$



(a) 2nd order stencil (eq. 3.4)

(b) 4th order stencil (eq. 3.5)

(c) 6th order stencil (eq. 3.6)

(d) 8th order stencil (eq. 3.7)

Figure 3.2: Stencils used for first derivatives in Astaroth

3.1.3 Central difference approximations for second derivatives

To obtain a central difference approximation for second derivatives the process is similar. A suitable expression for the second derivative can be obtained through a linear combination of Taylor series expansions:

Three-point second derivative central difference approximation

$$\begin{aligned}\frac{f(x+h)-2f(x)+f(x-h)}{h^2} &= \frac{1}{h^2}\left[\sum_{k=1}^{\infty}\left(\frac{f^{(k)}(x)}{k!}h^k\right)+\sum_{k=1}^{\infty}\left(\frac{f^{(k)}(x)}{k!}(-h)^k\right)\right] \\ &= \sum_{m=1}^{\infty}\left(\frac{f^{(2m)}(x)}{(2m)!}h^{2m-2}\right) \\ &= f''(x)+\underbrace{\sum_{m=2}^{\infty}\left(\frac{f^{(2m)}(x)}{(2m)!}h^{2m-2}\right)}_{\mathcal{O}(h^2)}\end{aligned} \tag{3.8}$$

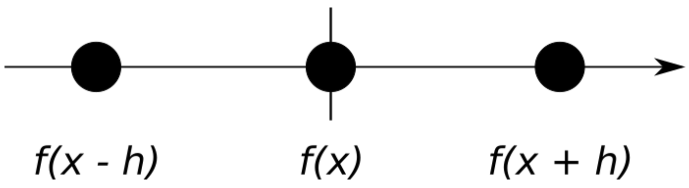


Figure 3.3: Second derivative central difference stencil

Central difference approximations of second derivatives used in Astaroth

$$s_r^+(x) = f(x + r\delta x) + f(x - r\delta x)$$

<u>2*nd* order</u>

$$\frac{\partial^2 f}{\partial x^2} = \frac{-s_0^+(x) + s_1^+(x)}{\delta x^2} + \mathcal{O}(\delta x^2) \tag{3.9}$$

<u>4*th* order</u>

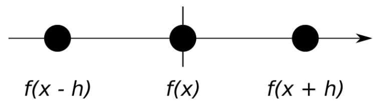
$$\frac{\partial^2 f}{\partial x^2} = \frac{-15 s_0^+(x) + 16 s_1^+(x) - s_2^+(x)}{12\delta x^2} + \mathcal{O}(\delta x^4) \tag{3.10}$$

<u>6*th* order</u>

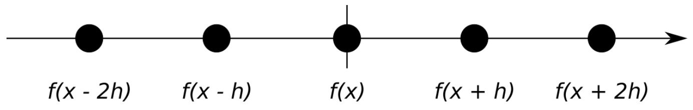
$$\frac{\partial^2 f}{\partial x^2} = \frac{-245 s_0^+(x) + 270 s_1^+(x) - 27 s_2^+(x) + 2 s_3^+(x)}{180\delta x^2} + \mathcal{O}(\delta x^6) \tag{3.11}$$

<u>8*th* order</u>

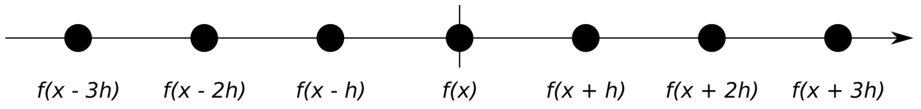
$$\frac{\partial^2 f}{\partial x^2} = \frac{-7175 s_0^+(x) + 8064 s_1^+(x) - 1008 s_2^+(x) + 128 s_3^+(x) - 9 s_4^+(x)}{5040\delta x^2} + \mathcal{O}(\delta x^8) \tag{3.12}$$



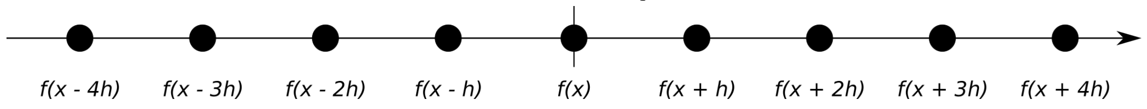
(a) 2nd order stencil (eq. 3.9)



(b) 4th order stencil (eq. 3.10)



(c) 6th order stencil (eq. 3.11)



(d) 8th order stencil (eq. 3.12)

Figure 3.4: Stencils used for second derivatives in Astaroth

3.1.4 Mixed partial derivatives

A mixed partial derivative of two orthogonal directions with uniform spacing can also be approximated using central differences. This is the simplest four-point stencil, from Abramovitz & Stegun [30, eq. 25.3.26]:

$$\frac{f(x+h,y+h)-f(x+h,y-h)+f(x-h,y+h)+f(x-h,y-h)}{4h^2}=\frac{\partial^2 f}{\partial x \partial y}+\mathcal{O}(h^2)$$

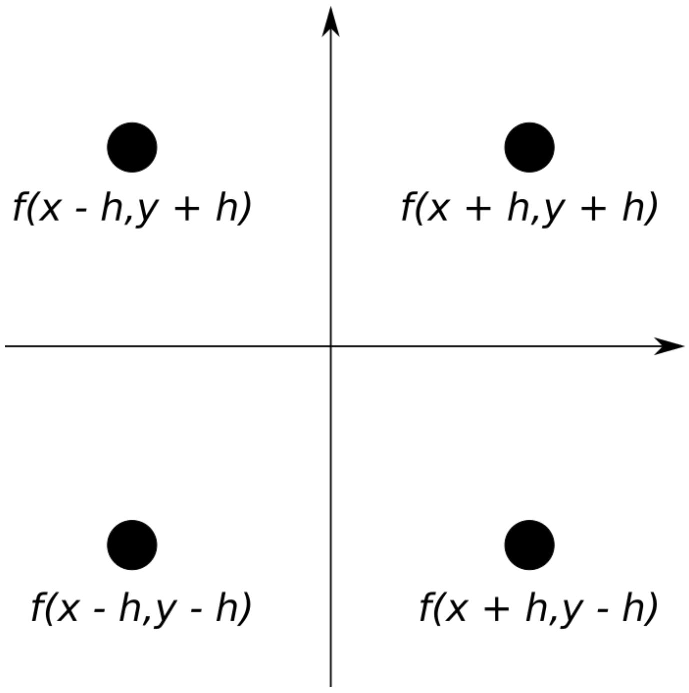


Figure 3.5: Stencil for the four-point mixed partial difference approximation [30, eq. 25.3.26]

Central difference approximations of mixed partial derivatives used in Astaroth

Astaroth uses a mixed partial derivative, inherited from the Pencil Code. These are covered by the Pencil Code manual, section H.3, "The bidiagonal scheme for cross-derivatives" [23, H.3]. The coefficients are the second order derivative coefficients scaled by a factor of $\frac{1}{4}$ and missing the coefficients for the central point.

$$\boxed{\begin{cases} \delta x = \delta y = h \\ s_r^{\square}(x) = f(x+rh,y+rh)-f(x+rh,y-rh)-f(x-rh,y+rh)+f(x-rh,y-rh) \end{cases}}$$

<u>2*nd* order</u>

$$\frac{\partial^2 f}{\partial x \partial y} = \frac{s_1^{\square}(x)}{4h^2} + \mathcal{O}(h^2) \tag{3.13}$$

<u>4*th* order</u>

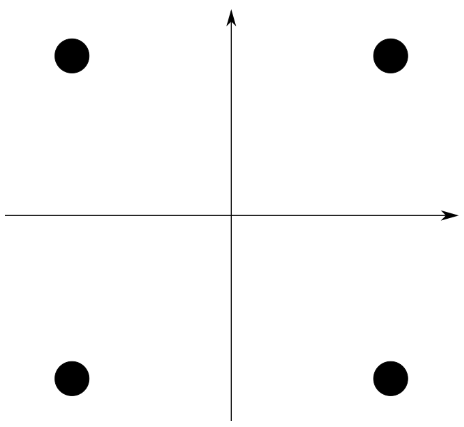
$$\frac{\partial^2 f}{\partial x \partial y} = \frac{16 s_1^{\square}(x) - s_2^{\square}(x)}{48h^2} + \mathcal{O}(h^4) \tag{3.14}$$

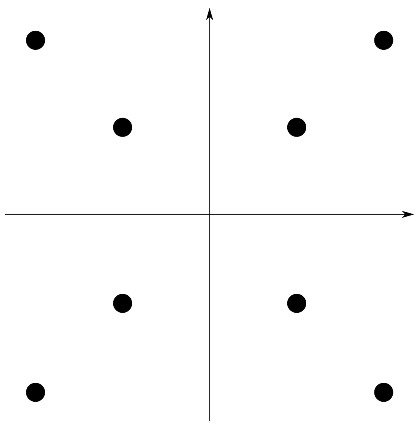
<u>6*th* order</u>

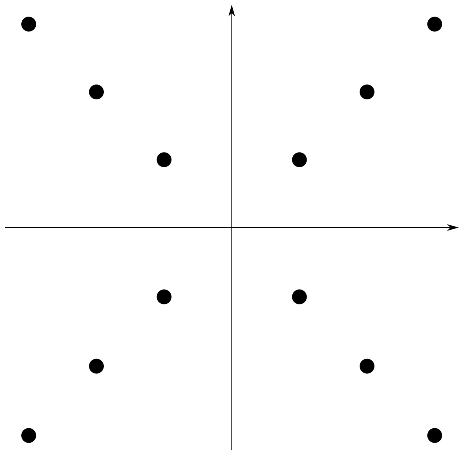
$$\frac{\partial^2 f}{\partial x \partial y} = \frac{270 s_1^{\square}(x) - 27 s_2^{\square}(x) + 2 s_3^{\square}(x)}{720h^2} + \mathcal{O}(h^6) \tag{3.15}$$

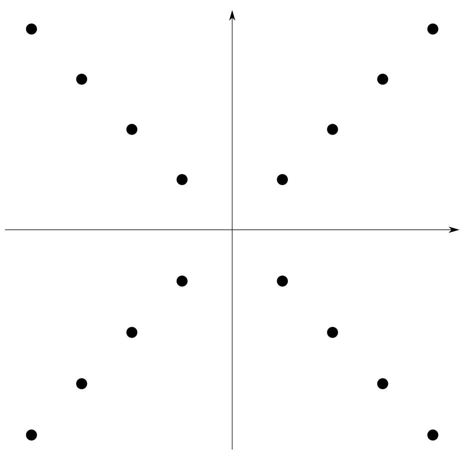
<u>8*th* order</u>

$$\frac{\partial^2 f}{\partial x \partial y} = \frac{8064 s_1^{\square}(x) - 1008 s_2^{\square}(x) + 128 s_3^{\square}(x) - 9 s_4^{\square}(x)}{20160h^2} + \mathcal{O}(h^8) \tag{3.16}$$


(a) 2nd-order mixed partial stencil (eq. 3.14)


(b) 4th-order mixed partial stencil (eq. 3.14)


(c) 6th-order mixed partial stencil (eq. 3.15)


(d) 8th-order mixed partial stencil (eq. 3.16)

3.1.5 The combined stencils

When all three types of approximations to differentials are used for a variable, the total data access pattern is the union of the stencils of each central difference approximation. The first derivative stencils are subsets of the second derivative stencils. Therefore, a full stencil is a combination of the second derivative stencil and the mixed partial stencil of the same order.

The MHD problems are three-dimensional, so the second derivative stencil is drawn in every coordinate direction, and the mixed partial stencil is drawn in every coordinate plane. Note that this means the full stencil has 2-dimensional diagonals but not 3-dimensional ones.

Figure 3.7 shows a two-dimensional slice of the full stencil for a 6th-order accurate simulation. The full three-dimensional stencil is the union of three of these stencils in every coordinate plane. While Astaroth supports other configurations, this stencil is the one that I've used when benchmarking Astaroth.

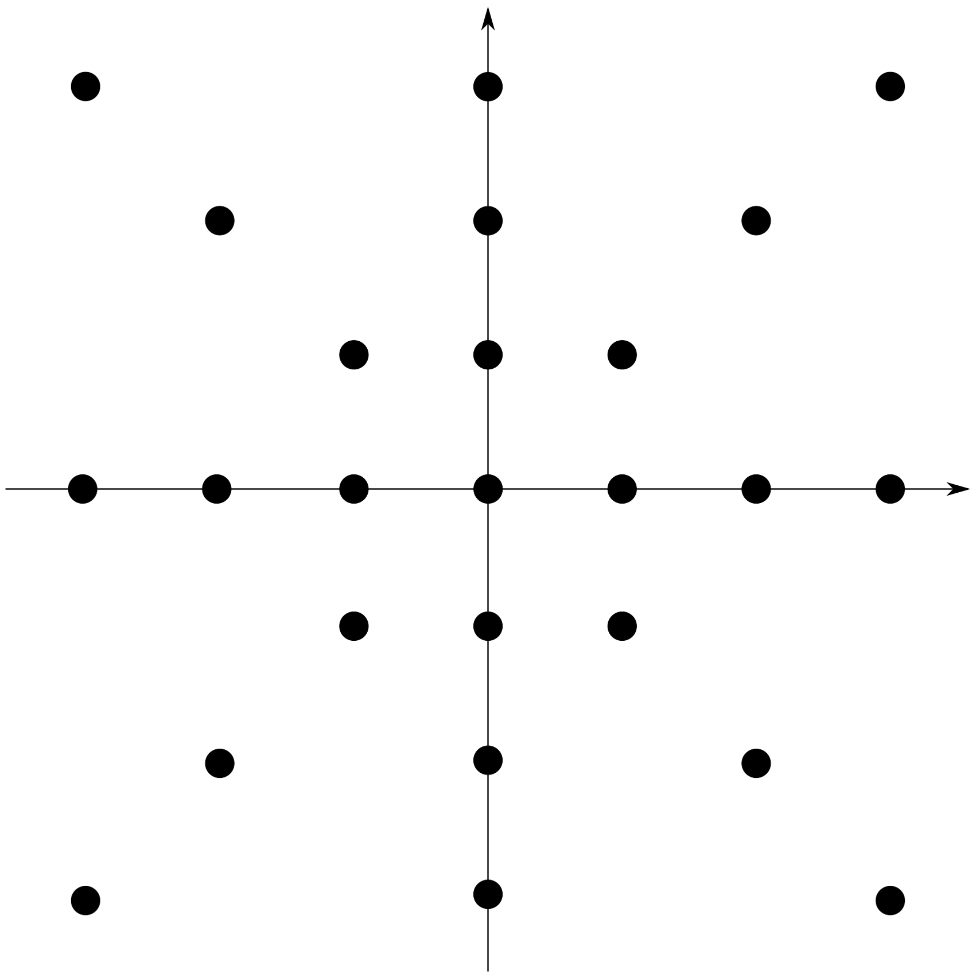


Figure 3.7: A slice of the full 6th-order accurate stencil in a coordinate plane.

3.2 The Runge-Kutta method

Astaroth needs a way to advance the simulation in time, a way to calculate the values at time $\tau + \delta t$ using values at time τ. This is done by adding the time derivative to the values. Since it is a process of applying derivatives to a value over an interval of time, the process is also called *time integration*.

Runge-Kutta methods are one way to integrating differential equations numerically over time. The first such methods were developed by Runge [31] and Kutta [32] at the turn of the 20th century.

A Runge-Kutta method (RK method) is used after all the spatial derivatives have been used to solve for an approximation of a time derivative $D(f, \delta t)$. In the Runge-Kutta method, the time derivative is calculated multiple times and applied multiple times in order to advance the state of f by a timestep δt, from $f(\tau)$ to $f(\tau + \delta t)$.

A three-step Runge-Kutta scheme

Astaroth uses a modified version of a third-order RK method developed by Williamson's [33, (2)] and the same coefficients Williamson uses in his example (11). Astaroth uses this method because it has been used in the Pencil Code (discussed in [24]).

Algorithm 1 is Williamson's Runge-Kutta scheme [33, (2)].

Algorithm 1 Williamson's original algorithm (2)

```
Given coefficients $\boldsymbol{b} = (b_1, b_2, ..., b_n), \boldsymbol{a} = (0, a_2, ..., a_n)$
Given:
  $f(\tau)$, the initial state
  $\delta t \geq 0$, a timestep
  $D(f, \delta t)$, an approximation of the rate of change $\frac{\partial f(t)}{\partial t}$

$f \leftarrow f(\tau)$
$q \leftarrow 0$
for $i \leftarrow 1...n$ do
  $q \leftarrow a_i q + \delta t D(f, \delta t)$
  $f \leftarrow f + b_i q$
end for
```

This algorithm has been refactored into algorithm 2 for Astaroth, where Pekkilä has used the fact that $q_{i-1} = \frac{f_{i-1} - f_{i-2}}{b_{i-1}}$ to remove the variable q altogether. He instead uses two buffers to store f into, which can be swapped after a calculation has been made. In order to express buffer swaps in the algorithm I've used the c-style dereferencing operator * and address resolution operator &.

Algorithm 2 Astaroth's RK3 algorithm, from Pekkilä [7], equation (3.25)

```
Given coefficients $\boldsymbol{b} = (b_1, b_2, ..., b_n), \boldsymbol{a} = (0, a_2, ..., a_n)$
Given:
  $f(\tau)$, the initial state
  $\delta t \geq 0$, a timestep
  $D(f, \delta t)$, an approximation of the rate of change $\frac{\partial f(t)}{\partial t}$

$(f\_1, f\_2) \leftarrow (f(\tau), f(\tau))$
$(f_c, f_p) \leftarrow (\&f\_1, \&f\_2)$
for $i \leftarrow 1...n$ do
    if i=1 then
        $*f_c \leftarrow *f_p + b_i \delta t D(*f_p, \delta t)$
    else
        $*f_c \leftarrow *f_p + b_i \left( a_i \frac{*f_p - *f_c}{b_{i-1}} + \delta t D(*f_p, \delta t) \right)$
    end if
    $(f_p, f_c) \leftarrow (f_c, f_p)$
end for
```

The coefficients for the algorithm used by Astaroth are taken from Williamson [33, table 1, case 7] and are as follows: $\boldsymbol{a} = \left(0, \frac{-5}{9}, \frac{-153}{128}\right)$, $\boldsymbol{b} = \left(\frac{1}{3}, \frac{15}{16}, \frac{8}{15}\right)$. This makes the algorithm a three-step Runge-Kutta method, because $n = 3$.

Time integration is the outer-most layer of calculation in a simulation. The work of approximating derivatives and solving the governing equations is contained within each Runge-Kutta substep.

To move a simulation forward by a single timestep δt, then, Astaroth needs to perform the following three times: for all points (1) approximate spatial derivatives using finite differences, (2) use the governing equations and the spatial derivatives to solve for an approximation of the time derivative $D(f, \delta t)$, and (3) perform a Runge-Kutta substep to apply the time derivative to the values.

4
Iterative Stencil Loops

An iterative stencil loop (ISL) is a program that applies a stencil operation repeatedly on a discrete grid, e.g., a finite difference method. Other examples include image processing, cellular automata, and numerical methods other than finite differences (such as finite volume methods, finite element methods). Astaroth has been designed specifically for numerical simulations but could in principle be used to implement any ISL design.

4.1 A model for reasoning about stencil solvers

I will now present a model for reasoning about ISLs. The model is loosely based on Andreas Schäfer's model [10, ch. 23]. The main components of the model are:

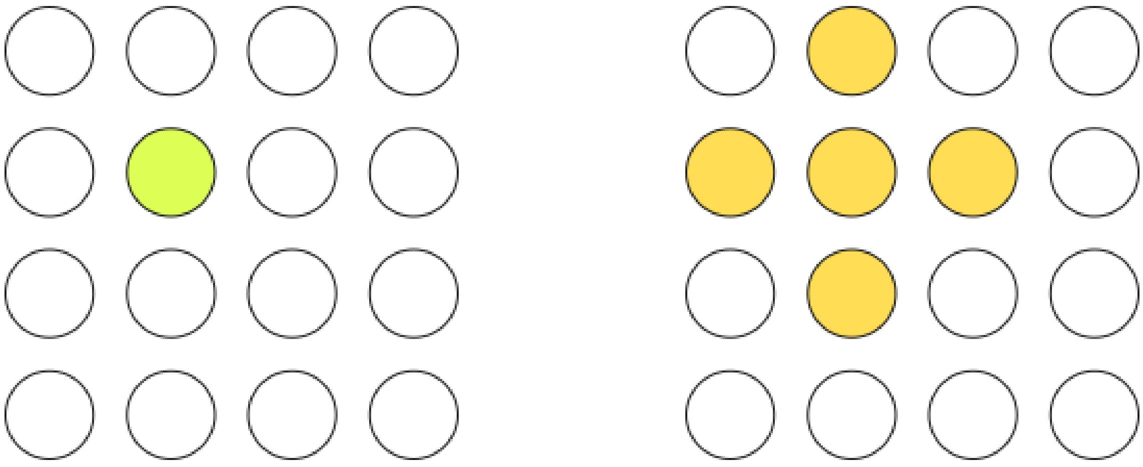
$\mathbf{G}_t$	the fine grid at iteration t. The fine grid is a uniform, structured, n-dimensional array of cell values.
$\mathbf{G'}_t$	the extended fine grid, including additional points around the grid boundary.
$\mathbf{G}_0$	the initial state of the fine grid.
$\mathbf{I_G}$	The index set of the fine grid.
$\mathbf{S}$	a stencil, an ordered set of indices defining an access pattern into the fine grid. The radius r of a stencil $\mathbf{S}$ is the maximal Chebyshev distance between the points in the stencil and its center: $$r_{\mathbf{S}} := \max_{\boldsymbol{s}\in\mathbf{S}} D_{Chebyshev}(\mathbf{0},\boldsymbol{s})$$
T	the transition function. For some cell at position $\boldsymbol{x}$, its value at iteration t is determined by applying T on the values from $t-1$ determined by the stencil: $$\mathbf{G}_t[\boldsymbol{x}] = T\left(\langle \mathbf{G'}_{t-1}[\boldsymbol{x}+\boldsymbol{s}] : \boldsymbol{s}\in\mathbf{S}\rangle\right),\ t>0$$

The list above contains the central objects of the model and some of their properties. For the discretized domain I have used the term *fine* grid in order to differentiate it from the *coarse* grid, which is the grid of subdomains that forms when the fine grid is decomposed.

I will also use the following notation to describe the size of a grid:

$l_i(\mathbf{G})$	the length of the ith dimension of a grid.
$\boldsymbol{l}_\mathbf{G}$	the size of a grid $\boldsymbol{l}_\mathbf{G} = (l_1(\mathbf{G}), ..., l_n(\mathbf{G}))$.

4.2 Updating the fine grid


(a) Highlighted cell c_t at iteration t

(b) Dependencies of c_t in iteration $t-1$

Figure 4.1: A cell and its inputs. The inputs are defined by a 5-point stencil.

The two most important components of the model are the transition function T and the stencil $\mathbf{S}$ which defines the shape of the input to T. At every iteration in an ISL, each grid cell is updated with a value calculated using T. In the case of an MHD simulation, the transition function T consists of applying all the governing equations to update each variable (the velocity field $\vec{u}$, the density ρ, etc.).

For the scope of this thesis, the inputs that produce a value will all be from previous iterations. In other words, all applications of T within an iteration are independent.

For a specific cell, the input to T is called the neighborhood of the cell. Which points are in the neighborhood is defined by the stencil $\mathbf{S}$. For Astaroth, these stencils are defined by the central difference formulae that are used.

Stencils can be categorized by either the number of grid points the stencil covers or by the radius of the stencil. The full 3D stencil presented in the previous chapter (fig. 3.7) is a 55-point stencil of radius 3. The 2D stencil in fig. 4.1 is a 5-point stencil of radius 1. For illustrative purposes, I will use this 5-point stencil for all figures of this chapter.

4.3 Domain decomposition

Let us move on to the problem of running a distributed ISL on an HPC cluster. In order to parallelize the execution of an ISL, the domain (fine grid) is decomposed into subdomains that are then assigned to processes. The subdomains produced in this way are interdependent, since the stencils will reach across subdomain boundaries. This means that processes with neighboring subdomains will need to communicate. I will begin by explaining the method of domain decomposition used by Astaroth. I will then deal with consequences of domain decomposition.

Domain decomposition produces a coarse grid $\mathbf{F}$, a grid consisting of the subdomains. Since we are working with a uniform structured grid, the decomposition problem is quite simple. For an n-dimensional grid we choose P processes $\{p_0, p_1, ..., p_{P-1}\}$ such that we can form the coarse grid $\mathbf{F}$, consisting of P subdomains by factoring P into n positive integers:

$$\prod_{k=1}^{n} a_k = P \tag{4.1}$$

which defines the dimensions of the coarse grid $\boldsymbol{l}_{\mathbf{F}} = (l_1(\mathbf{F}), ..., l_n(\mathbf{F})) = (a_1, ..., a_n)$.

The k-th dimension is cut $a_k - 1$ times, forming a_k regions in each dimension. We then define the index set of the coarse grid $\mathbf{I}_H$ to be the set of tuples in the cartesian product of the regions:

$$\mathbf{I}_H := \prod_{k=1}^{n} \mathbb{Z}_{l_k(\mathbf{F})} \tag{4.2}$$

The factorization is typically done through recursive bisection or as a byproduct of mapping process identifiers to a space-filling curve [34, p. 582], and therefore the factors are usually powers of two[1]. It is, of course, entirely possible to use factors other than powers of two.

More complex grids (e.g., with varying levels of refinement over the domain or non-rectangular geometry) may require more complex decomposition approaches. A good introduction with a bibliography is given in the *Encyclopedia of Parallel Computing* [34, pp. 578–587].

[1]Another reason is that nodes in a cluster typically have processing resources in quantities of powers of two.

Assigning subdomains to processes

Assuming each process is assigned exactly one subdomain, we need a one-to-one map $\boldsymbol{f}$ from 1-dimensional process identifiers to n-dimensional coarse grid coordinates:

$$\boldsymbol{f} : \mathbb{Z}_N \mapsto \mathbf{I}_\mathbf{F} \tag{4.3}$$

Using the map we can refer to processes by their assigned subdomain index:

$$p_{\boldsymbol{f}(i)} := p_i \tag{4.4}$$

Both index-schemes will be used from here on, and they can be distinguished by their typography. I will use p_i for 1-dimensional indices, and $p_{\boldsymbol{\iota}}$ for n-dimensional indices.

This mapping can be any function that fits this signature. E.g., in a torus-based Cray machine, the network topology could directly be used as a map. Another approach — employed in Astaroth — is to use a space-filling curve (SFC) as the map. This approach does limit the coarse grid topology to what the particular SFC being used can fill.

Local grids

The data in a process's subdomain will also be in the form of a regular grid. Since we have assumed each process owns exactly one subdomain, we will call this region the *local domain* or *local grid* $\mathbf{g}_{\boldsymbol{\iota}}$ of a process $p_{\boldsymbol{\iota}}$. For simplicity, all local domains have the same dimensions $\boldsymbol{l}_\mathbf{g} = \boldsymbol{l}_\mathbf{G} \oslash \boldsymbol{l}_\mathbf{F}$, where $\oslash$ is the element-wise division operator,

$$\boldsymbol{l}_\mathbf{g} = \boldsymbol{l}_\mathbf{G} \oslash \boldsymbol{l}_\mathbf{F} = \left(\frac{\boldsymbol{l}_1(\mathbf{G})}{\boldsymbol{l}_1(\mathbf{F})}, \dots, \frac{\boldsymbol{l}_n(\mathbf{G})}{\boldsymbol{l}_n(\mathbf{F})} \right)$$

The local grid re-indexes the global grid, and $\mathbf{g}_{\boldsymbol{\iota}}[\boldsymbol{x}]$ is only defined for data that $p_{\boldsymbol{\iota}}$ owns:

$$\mathbf{g}_{\boldsymbol{\iota}}[\boldsymbol{x}] = \mathbf{G}[\boldsymbol{\iota} \odot \boldsymbol{l}_\mathbf{g} + \boldsymbol{x}], \;\; \mathbf{0} \preceq \boldsymbol{x} \prec \boldsymbol{l}_\mathbf{g} \tag{4.5}$$

where $\odot$ is the element-wise multiplication operator, and $\prec, \preceq$ are defined in terms of coordinate-wise order.

Neighboring subdomains in the coarse grid

A subdomain $\mathbf{g}_{\boldsymbol{\kappa}}$ is in the neighborhood of another subdomain $\mathbf{g}_{\boldsymbol{\iota}}$ if any stencils evaluated in $\mathbf{g}_{\boldsymbol{\iota}}$ extend into $\mathbf{g}_{\boldsymbol{\kappa}}$, as shown in figure 4.2. In the case of a stencil that extends along all coordinate axes and their diagonals, two subdomains are therefore neighbors if the Chebyshev distance between their indices is exactly *1*:

$$N(\mathbf{g}_{\boldsymbol{\iota}}) = \{\mathbf{g}_{\boldsymbol{\kappa}} : D_{Chebyshev}(\boldsymbol{\iota}, \boldsymbol{\kappa}) = 1\} \tag{4.6}$$

The neighborhoods thus formed are n-dimensional Moore neighborhood.

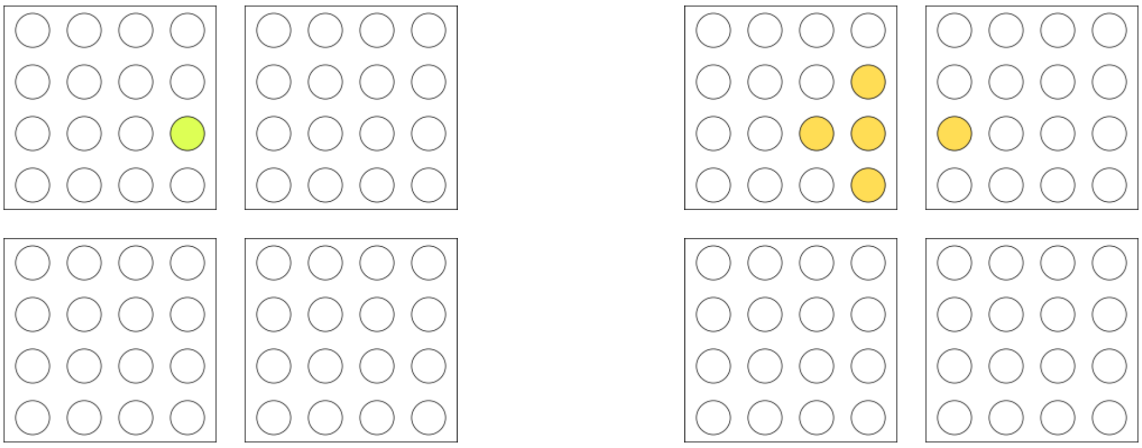

(a) A cell at a partition boundary. (b) The cells dependencies, one of which is remote.

Figure 4.2: The two top subdomains are neighbors, because a cell in the top left subdomain has dependencies in the top right subdomain.

Each neighbor can also be identified by a relative index $\boldsymbol{\iota} - \boldsymbol{\kappa}$. The index set of neighbors can then be defined as:

$$\mathbf{I}_{N(\mathbf{g}_{\boldsymbol{\iota}})} = \{\boldsymbol{\iota} - \boldsymbol{\kappa} : \mathbf{g}_{\boldsymbol{\kappa}} \in N(\mathbf{g}_{\boldsymbol{\iota}})\} \overset{maximally}{=} \{-1, 0, 1\}^n \setminus \mathbf{0}. \tag{4.7}$$

This maximal neighborhood index set turns out to be useful, so let us give it a name:

$$\mathbf{I}_N := \{-1, 0, 1\}^n \setminus \mathbf{0} \tag{4.8}$$

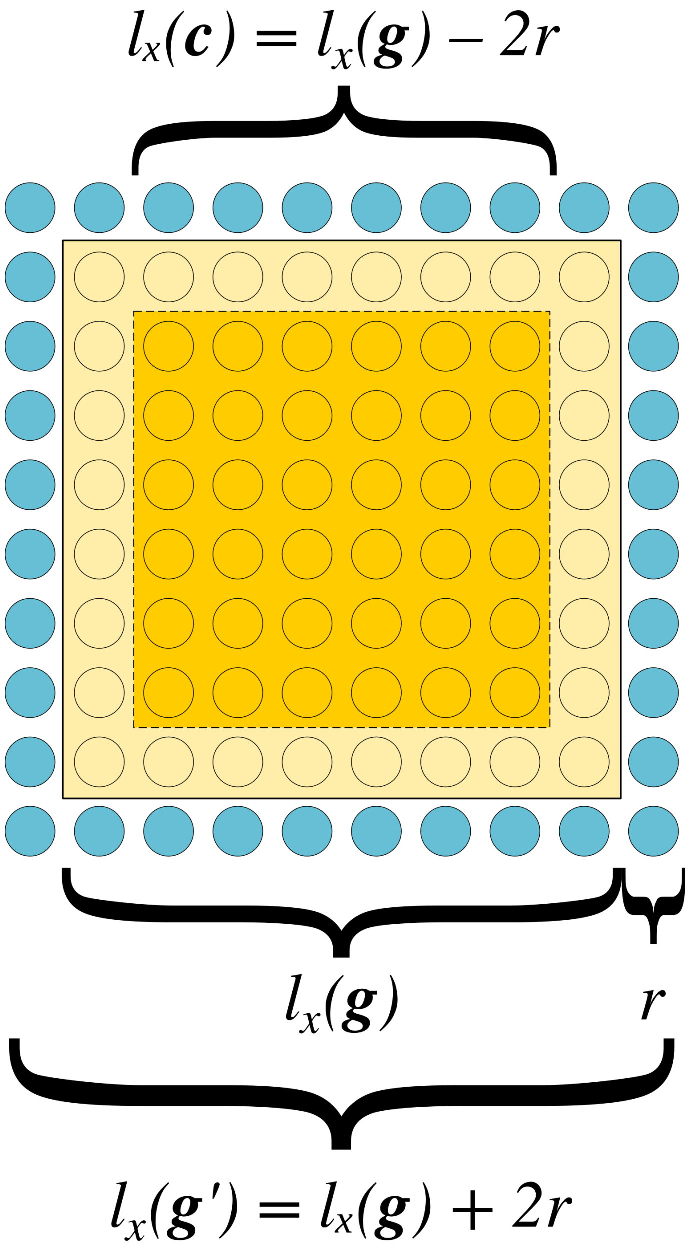
Using this definition, the size of a subdomain's neighborhood is maximally $|N(\mathbf{g}_{\boldsymbol{\iota}})| = 3^n - 1$. The neighborhood of a process may be smaller if the stencil does not extend along every diagonal or if $l_k(\mathbf{F})$ is less than three for some dimension k.

Example: For a 3D case, there are maximally $3^3 - 1 = 26$ neighbors, although some stencils require fewer neighbors. Recall the 55-point stencil that Astaroth uses for 6th-order accurate simulations, fig. 3.7 presented at the end of section 3.1. The stencil has no 3D-diagonals, so there are no neighbors along these diagonals. There are $2^3 = 8$ corners in a cube, and so the exact neighborhood of each process contains only $26 - 8 = 18$ processes in this case.

4.4 The extended grid: the core, the shell, and the halo

Since stencils extend outside the local grid $\mathbf{g}_\iota$ and into neighboring subdomains, we also need the concept of an extended local domain $\mathbf{g}'_\iota$ which contains all points necessary for updating the local grid. $\mathbf{g}'_\iota$ consists of $\mathbf{g}_\iota$ and a *halo* $\mathbf{h}_\iota$ that surrounds $\mathbf{g}_\iota$.

We now see that the local grid $\mathbf{g}_\iota$ has a region that depends on the halo and a region that is independent of it. In other words, the extended grid $\mathbf{g}'$ has a three-layered onion structure, as shown in fig. 4.3.



$l_x(\mathbf{c}) = l_x(\mathbf{g}) - 2r$
$l_x(\mathbf{g})$
r
$l_x(\mathbf{g}') = l_x(\mathbf{g}) + 2r$

Figure 4.3: The extended domain of a process is akin to an onion, consisting of a core (in orange), a shell (in light ocher), and a halo (in blue). The core and shell are part of the local grid, but the halo contains non-local data.

Starting from the inside out, I call the layers:

c the inner core. Contains cells independent of the halo. All cells with distance r or greater from the halo are in the core.

$\mathbf{g} \setminus \mathbf{c}$ the shell of the local domain. All cells in the local domain of a process not within the core are part of the shell. It is the outermost layer of the local grid with a thickness r. These cells all depend on data from neighboring processes.

h the halo (also called a ghost zone). A volume with thickness r outside the local domain of a process, consisting of all non-local data which the shell depends on.

The whole onion is the extended local grid $\mathbf{g}'$, consisting of the local grid and the halo.

$$\mathbf{g}' = \mathbf{g} \cup \mathbf{h}$$

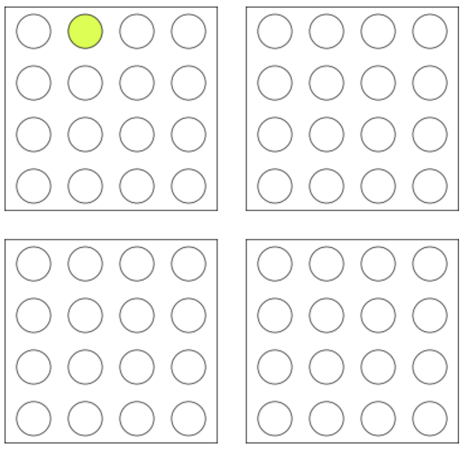
Since the halo is not a part of the local grid, and therefore not locally available, halo data needs to be communicated from other processes. Each iteration in an ISL, a process will receive its halo from other processes. The halo consists of data from the shells of neighboring processes. Symmetrically, a process will need to send out its shell to form the halos of other processes. This process is called *halo exchange*.

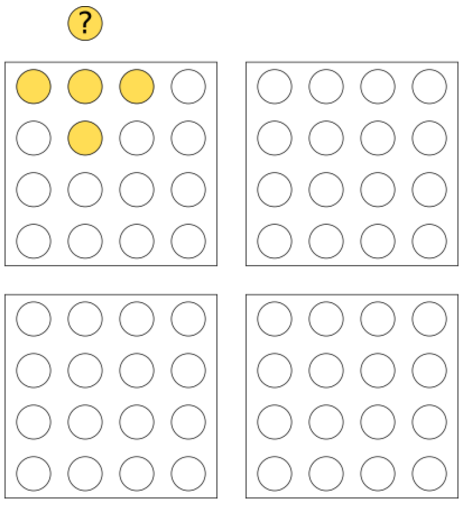
4.5 Boundary conditions

What happens if the stencil extends beyond not only the local domain but the entire global computational domain as well? This region outside the global grid is the halo of the global grid $\mathbf{H}$. The extended global grid $\mathbf{G}'$ consists of the global grid and its halo, $\mathbf{G}' = \mathbf{G} \cup \mathbf{H}$. If $\mathbf{H}$ isn't handled specially, the values of the transition function will be undecidable for those cells. This is a boundary value problem, and it is solved by applying *boundary conditions*.

For ISLs, special-case stencils could be defined for those cells near the perimeter, but this would create a heterogeneous set of tasks. Generally, the same stencil is often used throughout, and $\mathbf{H}$ is populated through some *boundary condition procedure* [35, § 4.9]. The benefit of the latter method is that it simplifies the design of an ISL. It is also more suited for a SIMD (single instruction, multiple data) model of execution, as the same instructions can be applied for all data.

A list of three types of boundary condition procedures follows. This list is just to show that the solution that Astaroth uses (periodic boundary conditions) is not unique. The list is by no means exhaustive.


(a) A cell at the boundary of the grid.



(b) The neighborhood of the cell, extending into **H**.

Figure 4.4: The stencil extends outside the grid for cells at the boundary, making the cells value undecidable. This problem is solved by applying boundary conditions.

Symmetric boundary conditions

Symmetric boundary conditions use mirror symmetry with the boundary plane as the plane of symmetry to generate the missing values. In the domain of fluid dynamics, symmetric boundary conditions correspond to physical walls at the boundary or some symmetry in the field [35, § 4.9.3].

Extrapolation

Extrapolation can also be used to create non-reflecting boundaries. In the domain of fluid dynamics, extrapolated boundary values correspond to outflow boundaries, i.e., we want flows that meet the boundary to carry on forward without reflecting back and disturbing the rest of the system. If the main region of interest in a domain is small, the domain can be shrunk to contain only that region, and outflow boundaries can be used to simulate the rest of the original domain. This saves computer resources [35, § 4.9.2].

Periodic boundary conditions

For periodic boundary conditions, the computational domain wraps around at the boundaries. This method essentially solves the boundary value problem by eliminating the boundary. Simulations run for this thesis use periodic boundary conditions.

Using periodic boundaries, a cell at the bottom boundary of the grid is adjacent to one at the top boundary and ditto for cells at the left and right boundaries, as illustrated in figure 4.5. With this, all boundary conditions can be resolved by halo exchange.

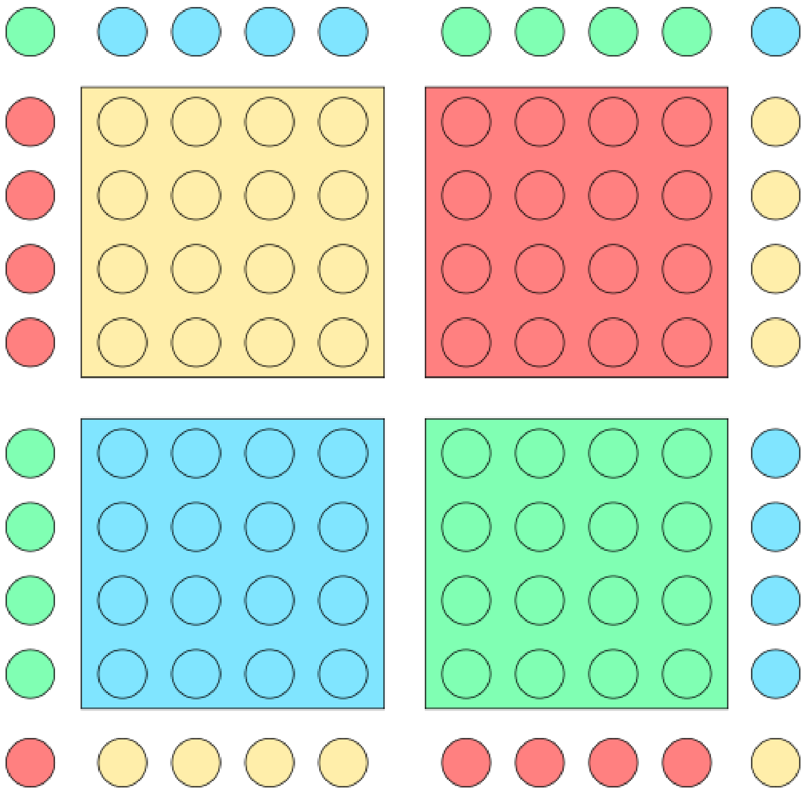


Figure 4.5: Periodic boundaries wrap the domain around itself.

4.6 Dependency analysis

Let us move away from the structure of the grid and on to the structure of work in ISLs.

The calculation of a single cell's values at iteration t depends on values at iteration $t-1$, as shown in figure 4.1. Thus even though all computational tasks in one iteration may be independent of each other, sequences of cell values over time are highly interdependent.

For non-trivial stencils, any proper subset of a grid will only be able to provide data for an even smaller proper subset of itself at the next iteration. In figure 4.6 we see a pyramidal shape forming in time, narrowing as fewer cells have their dependencies fulfilled.

If we consider updating a region of data a task in an ISL, all tasks are highly interdependent with no clear *a priori* order of execution. The only way to manage the scheduling of tasks, then, is to keep track of the dependencies dynamically.

In order to implement the task scheduler, we will need to understand the structure of dependency in an ISL.

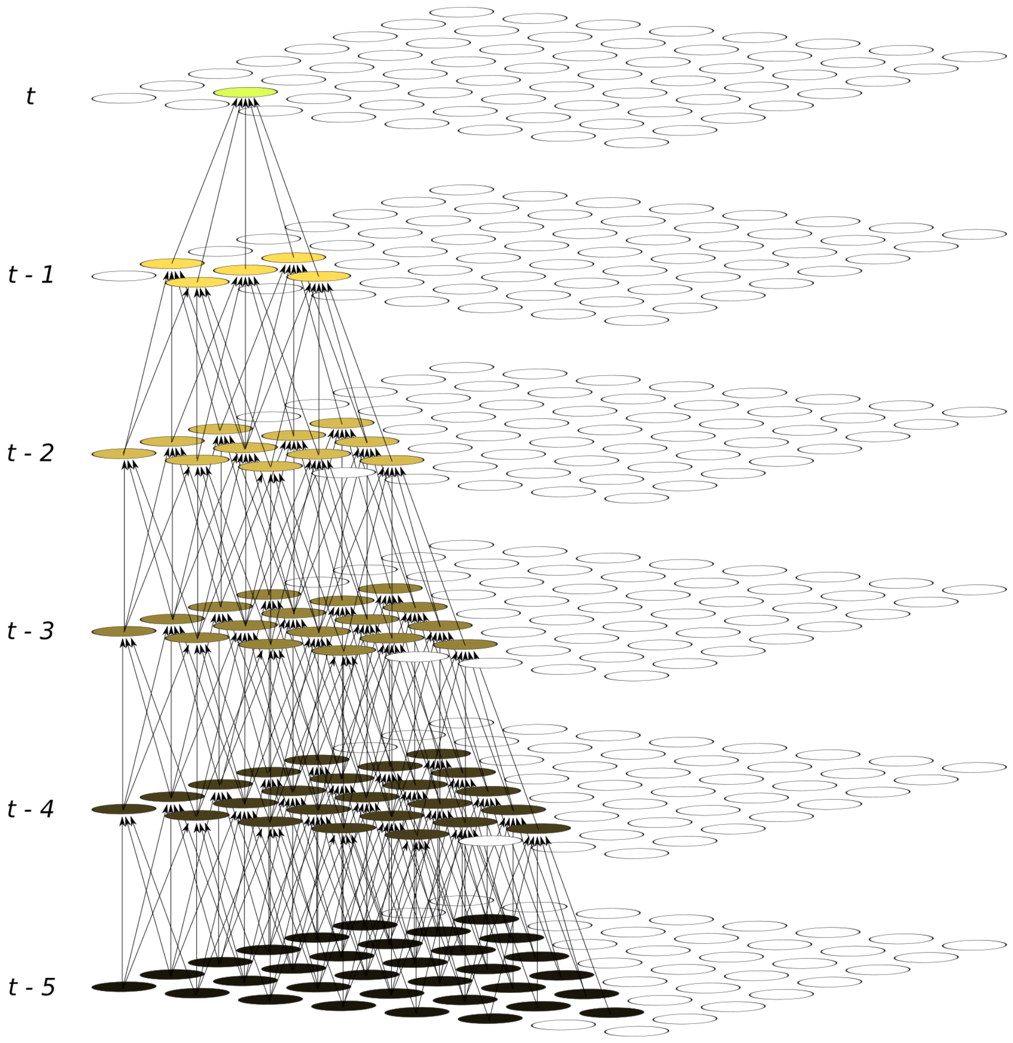


Figure 4.6: The green highlighted cell at iteration t has the pictured dependencies from iterations $t-1$ to $t-5$

Tasks that operate on regions of data

Let us call a unit of work a task. I will now lay out a system for reasoning about dependencies between tasks in an ISL and derive some rules for deciding when a dependency exists between two tasks.

There are two types of tasks that use the data in the shell: exchanging halos and performing a stencil-based calculation. Let us call these two types of tasks halo exchange tasks and compute tasks. Let **B** be the set of halo exchange tasks and **C** the set of compute tasks. I will also use a subscript to denote in which iteration a task is executing. E.g., all halo exchanges at iteration t is the set $\mathbf{B}_t$.

Halo exchange tasks and their regions

The set of halo exchange tasks is simply the set of pairs of messages that needs to be sent and received to construct the halos of a process and its neighbors. Each halo exchange task produces one region in the halo of a process and sends out a region from the shell. I call the set of output regions in the halo the *incoming halo region family* and the set of input regions in the shell the *outgoing halo region family*.

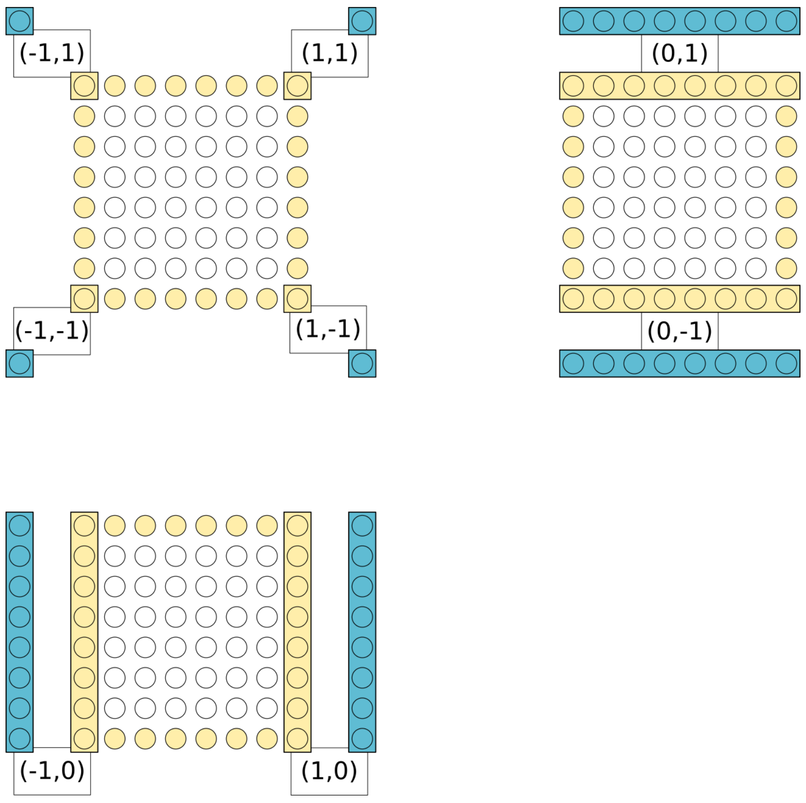


Figure 4.7: Data regions of halo exchange tasks and their respective region ids for a two dimensional grid with a stencil of radius 1. Each pair of outgoing (ocher) and incoming (blue) region with the same id are assigned to one halo exchange task in **B**. Note that the regions of the *outgoing halo region family* (ocher) in the shell overlap.

The halo exchange data regions for a two-dimensional case are illustrated in figure 4.7. Per equation 4.7, a region within a family can be indexed by the set of neighbor indices of a process $\mathbf{I}_N = \{-1,0,1\}^n \setminus \mathbf{0}$, where n is the number of dimensions. A halo exchange task is assigned one index from this index set. That index identifies both the outgoing and incoming regions in respective families.

Compute task regions

The set of compute tasks can in turn be defined in terms of the incoming halo regions as follows. Let us first simplify and consider all points within the radius of a stencil to be within the stencil. We can then divide the grid into regions by identifying cells that share the same set of dependencies on halo regions. This will form the *compute region family*. Each compute task is assigned one compute region.

The compute regions are indexed by a tuple in $\{-1,0,1\}^n$ which contains all non-zero coordinates of the halo regions that a compute region depends on as shown in figures 4.8 and 4.9. This neatly includes the calculation of the inner core (which depends on no region at all) as $\mathbf{C}[\mathbf{0}]$.

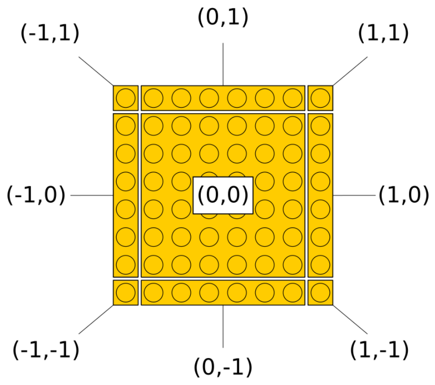


Figure 4.8: Data regions of compute tasks and their respective region ids for a two dimensional grid with a stencil of radius 1.

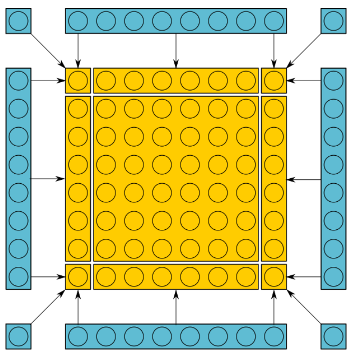


Figure 4.9: Illustrating how the halo regions form the compute regions. Along the edges of the boundary, the cells only depend on one halo region. In the corners, they depend on three halo regions.

Task dependency rules

A compute task depends on nearby incoming halo regions of halo exchange tasks in the same iteration. A compute task also depends on nearby compute regions in the previous iteration. A halo exchange task depends only on the compute tasks that will produce data in its outgoing halo region.

The compute region indices are related to the indices of prerequisite halo exchange tasks. The indices of outgoing halo regions are also related to the compute regions. The indices can therefore be compared to determine if two tasks are dependent.

These are the rules that define all dependencies in this system:

$$\mathbf{C}_t[\boldsymbol{v}] \text{ depends on all tasks } \mathbf{B}_t[\boldsymbol{u}] \quad \text{s.t. } \forall k,\ \boldsymbol{v}_k = \boldsymbol{u}_k \lor \boldsymbol{u}_k = 0 \tag{4.9a}$$

$$\mathbf{B}_t[\boldsymbol{v}] \text{ depends on all tasks } \mathbf{C}_{t-1}[\boldsymbol{u}] \quad \text{s.t. } \forall k,\ \boldsymbol{v}_k = \boldsymbol{u}_k \lor \boldsymbol{v}_k = 0 \tag{4.9b}$$

$$\mathbf{C}_t[\boldsymbol{v}] \text{ depends on all tasks } \mathbf{C}_{t-1}[\boldsymbol{u}] \quad \text{s.t. } \forall k,\ \boldsymbol{v}_k = \boldsymbol{u}_k \lor \boldsymbol{u}_k = 0 \lor \boldsymbol{v}_k = 0 \tag{4.9c}$$

There are a few relationships present in the rules, which we can write down as lemmas:

Lemma 1. $\forall t, \tau : T_t$ *depends on* $T'_{t+c} \iff T_\tau$ *depends on* $T'_{\tau+c}$.

Lemma 2. $\mathbf{C}_t[\boldsymbol{v}]$ *depends on* $\mathbf{B}_t[\boldsymbol{u}] \iff \mathbf{B}_t[\boldsymbol{u}]$ *depends on* $\mathbf{C}_{t-1}[\boldsymbol{v}]$.

Lemma 3. $\mathbf{C}_t[\boldsymbol{v}]$ *depends on* $\mathbf{C}_{t-1}[\boldsymbol{u}] \iff \mathbf{C}_t[\boldsymbol{u}]$ *depends on* $\mathbf{C}_{t-1}[\boldsymbol{v}]$.

It is due to lemma 2 that it is useful to think of sends and receives as two components of the same task. This symmetry only exists if halo exchange is performed in pairs of data regions that are mirrored through the halo exchange interface. In a way, this task dependency symmetry expresses the underlying assumption of symmetric stencils.

The lemmas above can be used to reason about how tasks may interfere with one another. In appendix B (under the assumption that the stencil has the rotational symmetry of a cube) I prove that it is impossible for one task to overwrite the input data of another task. The proof will simplify the design of the task scheduler, because it rules out additional buffer-based dependencies.

Examples and illustrations

An example of how the rules work follows. In a three-dimensional case, the following dependencies exist:

- $\mathbf{B}_t[(1,-1,0)]$ depends on $\mathbf{C}_{t-1}[(1,-1,-1),(1,-1,\ \ 1),(1,-1,\ \ 0)]$
- $\mathbf{C}_t[(1,-1,0)]$ depends on $\mathbf{B}_{t-1}[(1,-1,\ \ 0),(1,\ \ 0,\ \ 0),(0,-1,\ \ 0)]$
- $\mathbf{C}_t[(1,-1,0)]$ depends on $\mathbf{C}_{t-1}\begin{bmatrix}(1, & -1, & -1), & (1, & -1, & 1),\\ (1, & -1, & 0), & (1, & 0, & -1),\\ (1, & 0, & 1), & (0, & -1, & -1),\\ (0, & -1, & 1), & (1, & 0, & 0),\\ (0, & -1, & 0), & (0, & 0, & 0)\end{bmatrix}$

Figures 4.10 and 4.11 illustrate the dependencies between halo exchange and compute tasks in two-dimensional and three-dimensional cases respectively. Dependencies between compute tasks are not shown in the diagrams because the resulting image would be too difficult to read.

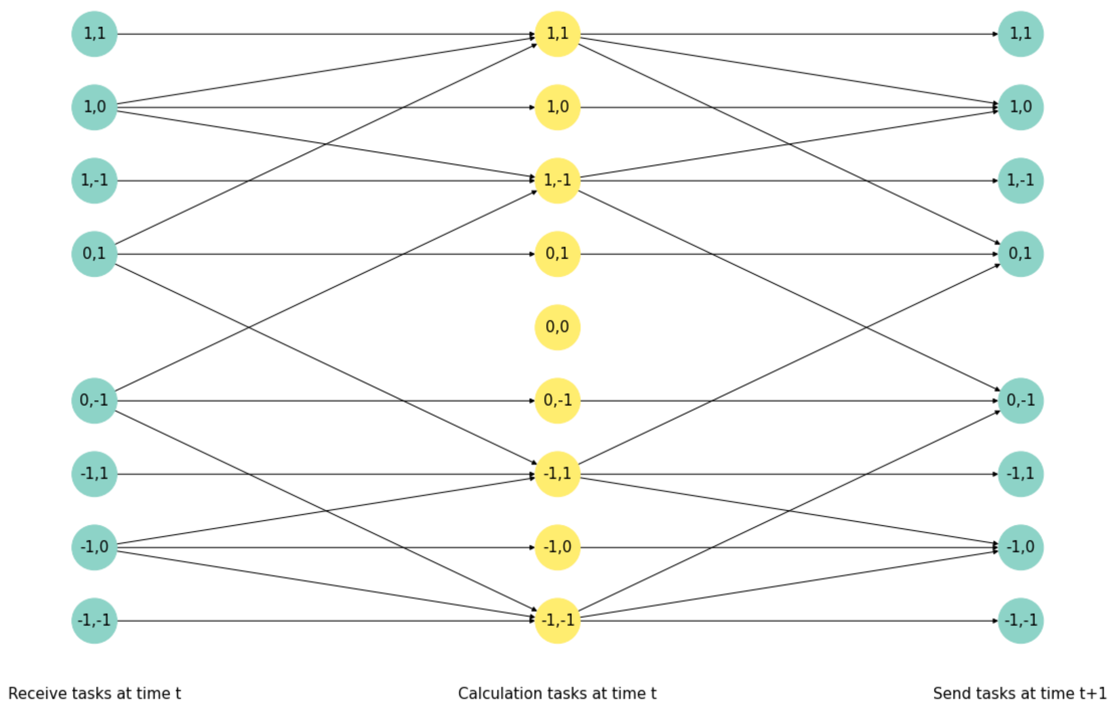


Figure 4.10: Dependencies in a two-dimensional ISL

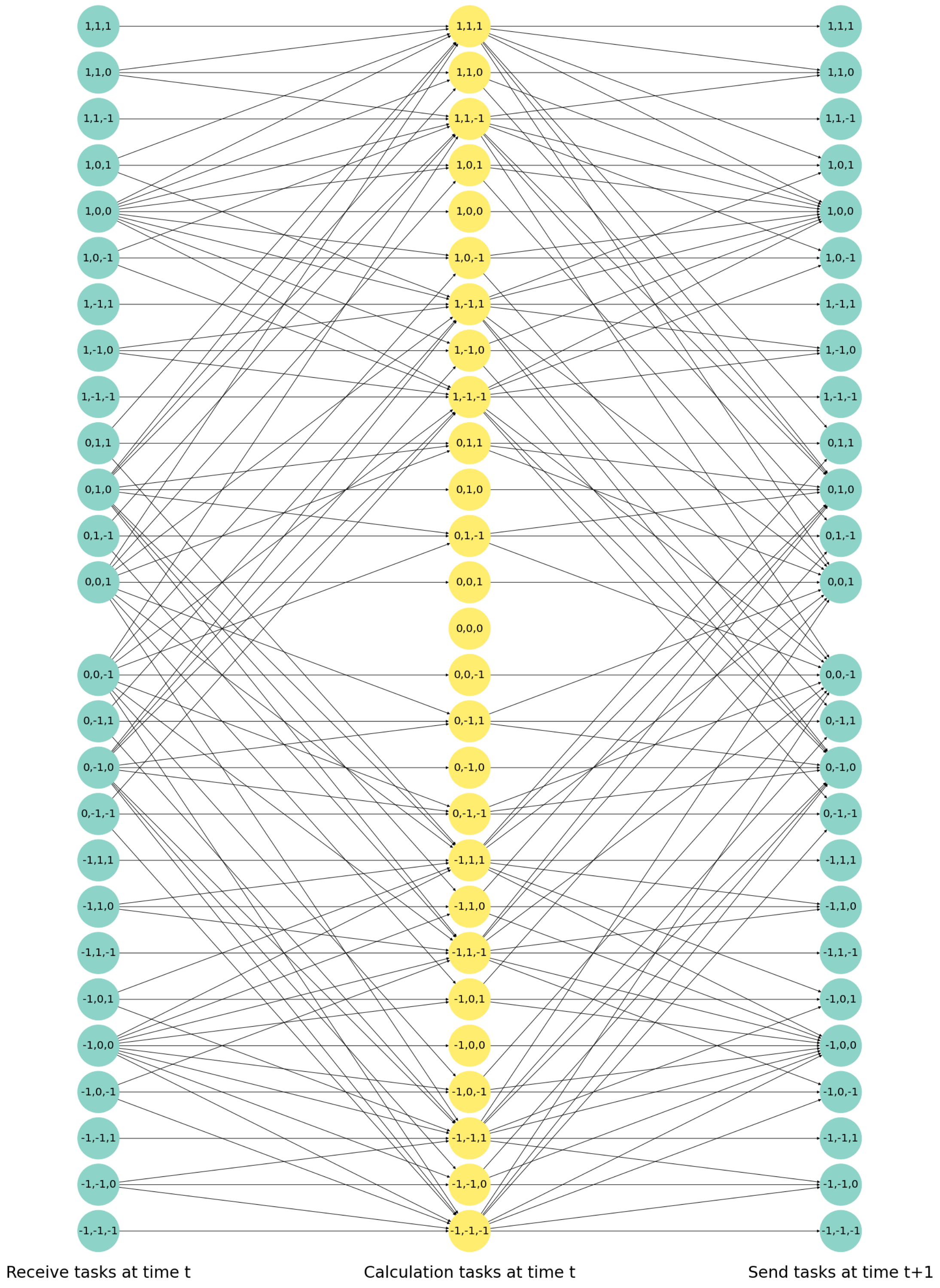


Figure 4.11: Dependencies in a three-dimensional ISL

Part II

Technological Background

When Astaroth is run at scale, it is run on high-performance computing clusters. Each computing environment provides a different communication stack, and Astaroth can be compiled for multiple architectures and with multiple communication libraries.

However, a specific cluster and communication stack has been used to run Astaroth in this thesis and understanding that stack will provide a model for reasoning about communication work.

First, in **chapter 5**, I will describe the application programming interface (API) that Astaroth uses to communicate between processes — Message Passing Interface (MPI).

Then, in **chapter 6**, I will describe the network stack. I will cover relevant details about the interconnection network deployed on the Puhti cluster [36] and techniques for moving data from GPU to GPU, both locally and over the network. I will also discuss the specific libraries that have been used by Astaroth to communicate data between GPUs efficiently.

5
MPI

MPI (Message Passing Interface) is a message passing API standard with many mature implementations, e.g., MPICH [5], MVAPICH [6], and Open MPI [4]. The standardization work began in 1992 [37] as a response to the messy state of message passing APIs at the time, which were largely proprietary and incompatible [38, pp. 10-12]. The standards body in charge of the MPI effort is the MPI Forum [39]. The latest stable standard is MPI 3.1 [40].

This chapter only scratches the surface of MPI. It contains just enough information to understand how MPI relates to Astaroth and the task scheduler. Two textbooks by Gropp, Lusk, et al.: *Using MPI* [38] and *Using Advanced MPI* [41] provide a more in-depth introduction to MPI.

5.1 Message passing

Message passing is a communication model used in distributed programs. In the message-passing model, data is passed between processes in the form of labeled messages of specified sizes with type information attached. Message passing is a two-sided communication process. Two-sided communication requires that both sending and receiving processes explicitly perform an API call in order to transfer the data.

This distinguishes message passing from other distributed programming models (such as global or shared address spaces and one-sided communication), where the communication is either abstracted away or only requires explicit input from one process.

Point-to-point messages

MPI provides both blocking and non-blocking point-to-point communication. MPI messages are labeled by: (1) size, (2) the ids of the participating processes (sender/receiver), (3) a tag (non-unique message id), and (4) a communicator (namespace). The order of messages is also an important implicit part of the metadata of a message.

When a process sends a message, it specifies the length and type of the data, the receiver's id, and a message tag. Before a process receives a message, it needs to specify the length and type of the data, the sender's id, and a message tag. Tags, senders, and receivers can also be defined as wildcards that match any tag, sender or receiver.

In addition to this metadata, both sender and receiver provide the MPI library with pointers to memory buffers. The sender provides the block of data that will be sent, and the receiver provides a buffer where the message contents will be written by the MPI library. The metadata of a message is called the *envelope* and the message content is called the *payload*.

Message matching

A send and receive will be paired by matching the tag, receiver id, and sender id of both API calls. If there are multiple possible matches, the order matters. The n-th send matches the n-th receive call.

The posted receives are held in a Posted Receive Queue (PRQ), waiting to be matched to an incoming message. When an incoming message is received, the PRQ is searched for a match. A performance consideration is that the queue search time is proportional to the PRQ length.

Unexpected messages

The MPI standard says that a process can encounter unexpected messages. This clashes with the two-sided communication process, because a receiving process must specify a buffer that will receive a message's data. If a process receives a message when no matching receive has been posted, the receiver must perform additional work to handle the message. MPI libraries solve this problem by using an unexpected message queue (UMQ) to buffer incoming messages without a destination. Once a receive is posted, the UMQ is searched for a match before the receive is posted to the PRQ [38].

Unexpected messages use more resources than expected ones. By definition, an unexpected message will not match a receive in the PRQ and searching the PRQ will take the maximal time [42]. If the message was sent using an eager protocol, the message content will also need to be temporarily stored somewhere so that it can be copied to its destination when a receive is posted.

As a corollary, a slew of unexpected messages will also impact performance by significantly increasing the time it takes to search the UMQ *when a receive is posted*. This can cause a surprising performance degradation if the size of a process's UMQ scales with the number of processes used to run an application. This was observed by Keller and Graham in 2010 [43] in Oak Ridge National Laboratory's Locally-Self-Consistent Multiple-Scattering simulation [44]. Care should therefore be taken to minimize the number of unexpected messages if performance is important.

One common performance bug is that sends are posted before corresponding receives. The obvious solution is to instead post receives ahead of time. This may create some overhead in terms of additional buffers in memory, but this overhead can be explicitly managed by the programmer and is easier to reason about. This is suggested by the following "Advice to users" given in the MPI standard, in a section on non-blocking communication [40, § 3.7]:

> The message-passing model implies that communication is initiated by the sender. The communication will generally have lower overhead if a receive is already posted when the sender initiates the communication (data can be moved directly to the receive buffer, and there is no need to queue a pending send request). However, a receive operation can complete only after the matching send has occurred. The use of non-blocking receives allows one to achieve lower communication overheads without blocking the receiver while it waits for the send.

There is no limit to *how* early receives may be posted, so they may just as well be posted immediately at initialization. This technique has been used in my treatment of Astaroth.

5.2 Protocols

There are two protocols typically used in MPI implementations. These are not prescribed by the MPI standard but suggested as "Advice to implementors" [40, § 3.4]. The standard suggests the eager protocol for short messages and the rendezvous protocol for longer ones. MPI implementations typically follow this advice. If a message destined for main memory is expected, the eager protocol should always be faster.

When the destination is in GPU memory, however, the situation is not that simple. For GPUs, special protocols need to be implemented, and rendezvous-type protocols will typically always be faster for larger messages (more on this in section 6.2).

Eager protocol

The eager protocol for sending messages sends the payload with the envelope. This requires fewer transactions, but if the message is unexpected on the receiving side, the data will have to be moved into a buffer and later copied into its destination [38, p. 248]. For GPUs, the eager protocol is limited, and a copy is always necessary [45, p. 83].

Rendezvous protocol

The standard describes the rendezvous protocol as follows [40]:

> The sender sends a request-to-send message. The receiver stores this request. When a matching receive is posted, the receiver sends back a permission-to-send message and the sender then sends the message.

This three-way exchange introduces latency and a small bandwidth overhead, but it does neatly avoid buffering of unexpected messages on the receiving end.

Because the receiver acts as the conductor in this protocol, it will not be overwhelmed by incoming messages. Therefore, this sort of protocol is good for situations where communication is constrained (as is the case with GPUs).

5.3 Synchronous vs. Asynchronous communication

Any communication task in MPI must be posted and completed on both sides of the communication. The way this occurs depends on whether the blocking or non-blocking API is used.

In both cases, the send and receive tasks are both posted by calling the appropriate function that starts the process. The functions are *MPI_Send* and *MPI_Recv* in the blocking API; *MPI_Isend* and *MPI_Irecv* in the non-blocking API.

In the blocking case, the function does not return until the message is delivered. Function completion signals work completion.

Non-blocking API calls use an additional parameter of type *MPI_Request* that is used to track the status of the message. The delivery of an asynchronous message can then be confirmed by querying the associated *MPI_Request*. The request can be queried with a call to either *MPI_Wait* or *MPI_Test*.

MPI_Wait will block until the message has arrived. *MPI_Test* will report whether the message has arrived, not blocking execution.

In the baseline version of Astaroth communication is started asynchronously, with *MPI_Isend* or *MPI_Irecv*, but is completed *synchronously*, with *MPI_Wait*. In my treatment, I have switched out all calls to *MPI_Wait* with calls to *MPI_Test*. This ensures that communication never blocks execution.

6
The Communication Stack

The previous chapter concerned the communication API used by Astaroth. This chapter focuses on the concrete communication stack used by Astaroth, the actual mechanisms that move data from one process to another.

The computing environment used in this thesis is CSC's HPC cluster Puhti [36]. Puhti nodes are on an Infiniband network, organized as a fat tree. Puhti nodes on the *gpu* partition are equipped with four Nvidia GPUs and two Mellanox network adapters. Astaroth runs a single process per GPU; four processes per node if all GPUs are utilized.

The data to be communicated resides in GPU memory, which is separate from the main memory of the computer. Astaroth uses Nvidias Compute Unified Device Architecture (CUDA) to program the GPUs. In the CUDA programming model, main memory is called *host memory*.

The data that Astaroth is working on needs to be moved from the memory of one GPU in the cluster to another GPU, potentially on another node. The data transfer from the GPU to the network interface controller is the most challenging. This transfer happens over the PCI Express (PCIe) interconnect on the motherboard, which is the bottleneck.

Direct Memory Access (DMA) allows PCIe devices to communicate with each other without host overhead, and clever solutions using DMA allow for more efficient data transfers on the internal PCIe network. Puhti supports these solutions and provides libraries to interface with them.

Astaroth uses Open MPI [4] as its MPI library, which takes care of MPI semantics. Open MPI in turn uses UCX, a communications library specialized for modern HPC environments. UCX then delegates specific communication tasks to highly specialized communications libraries developed for a certain communication task (e.g., *GDRCopy* performs remote GPU-to-GPU communication).

I will start with the simpler intra-node case before moving on to the networked case. I will then go through the lower-level libraries and up the stack, ending the chapter with a section on Open MPI.

6.1 Intra-node GPU-to-GPU communication

Intra-node communication is typically done through inter-process communication (IPC). CUDA IPC is an efficient mechanism for transferring data between Nvidia GPUs. With CUDA IPC data is moved directly between the memories of two GPUs without the use of staging buffers in main memory [46].

Potluri et al. have created two protocols using CUDA IPC for MPI messages, an eager protocol and a rendezvous protocol [46]. The eager protocol involves an extra copy, while the rendezvous protocol introduces a small IPC message latency. Both are used by UCX.

Eager CUDA IPC protocol

The CUDA IPC-based eager protocol writes messages into pre-allocated *eager buffers* in the receiving GPU's memory. A message header is written to host memory when the write to device memory is finished. The receiving process will match the header against its receive queue, wait for a CUDA IPC event that signals that the write has finished, and move the message from the eager buffer to its final destination buffer.

Waiting for the CUDA IPC event is needed to synchronize the header and message write operations, otherwise the header write might overtake the message write.

Rendezvous CUDA IPC protocol

The CUDA IPC-based rendezvous protocol has the receiving process wait for a Request To Send (RTS) control message to arrive with an IPC handle. Once a posted receive matches an RTS message, the receiving process copies the data straight from the sending GPU's memory to the receiving GPU's memory.

This is slightly different from the rendezvous presented in the MPI chapter, because the receiver performs the transfer, merging the final two steps of the three-way handshake into one.

6.2 InfiniBand networking

The Puhti nodes are connected with InfiniBand(IB), a switched fabric networking system for HPC. The InfiniBand interconnect is the set of hardware that operates the network, and the protocol that runs on it is called the InfiniBand protocol. The hardware supports features provided by the IB protocol, such as flow- and congestion control and Quality-of-Service (QoS) prioritization. In other words, the hardware and protocol are tightly coupled.

Through IB, Puhti also supports OpenFabrics-based Remote Direct Memory Access (RDMA), a way to directly write to and read from the working memory of nodes [1].

The IB protocol uses message queue pairs (QPs), consisting of one send queue and one receive queue. Any two queue pairs form a logical channel that can be used for sending data back and forth. The receive queues can be shared between many QPs on a node to reduce overhead.

If a process wants to communicate with another process, it posts a work request (WR) to a QP, either for sending or receiving. Once the request has been completed, a completion token is posted to a Completion Queue (CQ). A process can either poll the CQ or register to receive events in the CQ to keep track of the status of requests [47, ch. 14],[48].

Offloading

In IB networks, the network cards are called Host Channel Adapters (HCAs). These HCAs — such as the HDR100s on Puhti nodes — are equipped with application-specific integrated circuits (ASICs) that take on communication work that would typically be performed by the operating system running on the processor. The HCA also has direct memory access (DMA) to data in host memory.

In fact, the protocol discussed above is run on the HCA, the QPs and CQs residing in the network adapter's memory [48, § 6.2.2]. A send call made by a communication library goes straight to the HCA, which reduces communication overhead on the host.

RDMA

The HCA needs to map virtual memory addresses it receives in WRs to physical addresses used by user processes in host memory without asking the operating system for advice in order to perform DMA operations. To this end, user processes that need to take advantage of the HCAs DMA capabilities must register host memory regions with the HCA.

Each memory region is assigned a local key and a remote key. Using remote keys, a process can point at remote memory regions across the network. DMA involving remote memory keys is known as remote direct memory access (RDMA).

An RDMA write is a send request posted to the QP with a remote key, and an RDMA read is a receive request posted to a QP with a remote key. The HCA that owns the remote key will respond to a RDMA write request by simply moving the received RDMA data to the memory region pointed at by the remote key. Similarly, for an RDMA read, the responding HCA will send back data from the memory region pointed at by the key in the request [48, § 6.2.2].

[1] https://www.openfabrics.org

In this way, once memory regions have been registered and keys have been distributed to remote processes, a user process can initiate a data transfer without any action from the process at the other end. This means that RDMA writes and reads are one-sided operations.

GPUDirect RDMA

GPUDirect RDMA (GDR) is the term Nvidia and Mellanox use to describe their solution for moving GPU data over IB networks. The GDR mechanism allows the HCA to directly access GPU memory over PCIe the same way it accesses host memory. The goal was to completely bypass the host, transferring data directly between GPUs and HCAs.

However, Potluri et al. found that eager GPU-to-GPU RDMA is too slow, because an extra copy will be needed on the receive side. Copies are so slow on GPUs that any performance gained from using GDR is lost [45, p.83].

Potluri et al. also found that DMA transfers from GPU to HCA are the bottleneck when using GDR. GDR is lower-latency than host-assisted techniques, but the bandwidth of GDR transfers that read GPU memory is a factor of six lower than GDR transfers that only write to GPU memory [45, p. 81].

Based on these results, Potluri et al. created a hybrid rendezvous protocol that switches between the two transfer mechanisms. In this hybrid solution, GDR is only used for very small messages, while larger messages are buffered through host memory for a better-performing solution.

GDRCopy

The first attempt at GDR-based protocols [45] was later improved by the same group the next year [49]. In 2014, Shi et al. created a driver-based transfer mechanism for Host-GPU communication paths that can be used to implement better remote GPU-to-GPU protocols, including eager ones [49]. This protocol, called Fastcopy, registers its own buffers in the PCIe base address register (BAR) of the GPU and uses vector instructions for data movement if available, e.g., Advanced Vector Extensions (AVX) or Intel's Streaming SIMD Extensions (SSE) [49].

Fastcopy is fast when moving data from host memory to device memory but slow when moving data from device memory to host memory. This is because the PCIe interface of Nvidia GPUs has been optimized for writes, which can be bursted through write-combining and started asynchronously with write posting;however, the PCIe interface uses non-prefetchable Memory Mapped IO (MMIO) and therefore cannot burst reads [49]. It seems this design is a legacy from the original use-case of GPUs — frame buffers are updated quite frequently but rarely need to be read back into main memory.

Fastcopy was later renamed to GDRCopy. It should be noted that Fastcopy/GDRCopy is only faster for small messages. UCX switches between different protocols depending on their size and uses GDRCopy only for very small messages.

6.3 UCX

The framework that interacts with Infiniband and provides RDMA capabilities to Astaroth is Unified Communication X (UCX) [50]. It is a standardization effort that injects a layer between MPI and the lower-level network fabric protocols.

UCX uses the GDRCopy library to provide GPUDirect Remote Direct Memory Access (RDMA) support for small messages [49];other eager protocols exist as well. For larger messages, a rendezvous protocol is used. The exact rendezvous protocol can be one of many. The UCX developers recommend a zero-copy, RDMA GET-based protocol, which we have used [51, "Nvidia GPU Support"].

The point at which the switch from one protocol occurs is controlled by threshold variables. We have set the rendezvous threshold at 16KiB based on communication benchmarks[2]. Nearly all messages sent by Astaroth for simulations used in this thesis are so large that the rendezvous protocol is used.

6.4 Open MPI

Finally, the library we interact with directly in Astaroth's source code is Open MPI [4]. Open MPI is designed using a Modular Component Architecture (MCA) in order to support various network stacks, protocols and other system features that are necessary to implement the MPI standard. Open MPI is organized into *frameworks*, each of which defines an interface for *components* which belong to that framework. Open MPI has many components for communication that can be used all at once.

Open MPI uses UCX for GPU-aware MPI over Infiniband. Open MPI has to be configured and built with UCX support to enable the *ucx* component for the *pml* (point-to-point management layer) MCA framework. At runtime, Open MPI can then be configured to use the *ucx* pml component, which replaces the standard OB1 pml[3].

[2]Private communication , Fredrik Robertsén, CSC.

[3]OB1 is not an acronym, it is simply the name of the standard pml component.

Part III

Designing a Task Scheduler for Astaroth

We are ready to discuss the task graph scheduler I have built for Astaroth and how it compares to the baseline.

The first chapter, **chapter 7**, concerns the baseline version of Astaroth, before my treatment. I cover some of the implementation details of the baseline version and show that its runtime has a synchronous, fork-join behavior.

The next chapter, **chapter 8**, concerns my treatment of Astaroth. I first lay out some general arguments for task-based schedulers in HPC and give a specification of my task-based treatment. Then, I cover the design of the task scheduling system and how its runtime behaves.

Chapter 9 contains a performance comparison, where we will see that the task scheduler has better performance than the baseline version for communication-constrained workloads.

Chapter 10 contains a discussion of the results and ideas for future work.

7
Baseline Astaroth

This chapter is based on the git commit *3804e72* in the *mpi-paper-benchmarks-2020-09* branch of Astaroth's online git repository [27]. For a detailed account of this version of Astaroth, see "Scalable communication for high-order stencil computations using CUDA-aware MPI" by Pekkilä et al. (in review) [8].

Astaroth is a framework that consists of (1) a domain-specific language (DSL) for writing compute kernels, (2) a distributed runtime environment for running the kernels on GPUs, and (3) a C++ API for interacting with the runtime. I will focus on the runtime, since that is what I have worked on.

The Astaroth runtime is distributed, consisting of multiple processes each assigned to one GPU. Each iteration, all processes exchange halos with neighboring processes and calculate new values in their local domain.Simulations are progressed using a 3-step Runge-Kutta method[1]. One timestep in a simulation is three iterations of halo exchange and stencil calculation.

I will first explain how data is arranged at runtime. I will then cover the implementation of stencil calculations and halo exchanges, before finally moving on to the synchronous behavior of Astaroth's runtime.

7.1 Implementation details

Simulation data

The entire extended local grid is one big three-dimensional array in the GPU's memory. For the sake of this chapter, I will call this the *grid*.

Several variables — or $fields$ — are defined for each cell in the grid. The variables are stored in separate buffers, called *vertex buffers*. The individual vertex buffers are stored in a *vertex buffer array* which contains all variables over the entire grid.

Each vertex buffer is double buffered. One buffer is used for input, the other for output. The input and output buffers are swapped between iterations. The datatype of

[1]See section 3.2, algorithm 2

scalar fields is *AcReal*, which is either a float or double, depending on the configuration. Vector fields are implemented as a set of three scalar fields.

Data is also communicated between processes during halo exchange. Outgoing halo messages are copied from their regions in the shell and packed into message buffers that will be sent to neighbors. Symmetrically, incoming halo messages arrive into data buffers that are unpacked into regions in the halo. Message data is stored in *PackedData* and *CommData* data structures.

Stencil calculation (compute kernels)

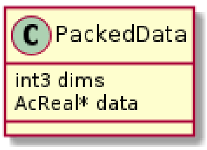
The function *acDeviceIntegrateSubstep* does stencil calculation. *acDeviceIntegrateSubstep* performs one step of the 3-step RK algorithm for a specified data region in the grid. Internally, the function calls a compiled DSL kernel (a CUDA kernel). As an example, the source code in listing 7.1 computes the result of the inner core.

```
const int3 m1 = (int3){2*NGHOST, 2*NGHOST, 2*NGHOST};
const int3 m2 = nn;
acDeviceIntegrateSubstep(device, STREAM_16, isubstep, m1, m2, dt);
```

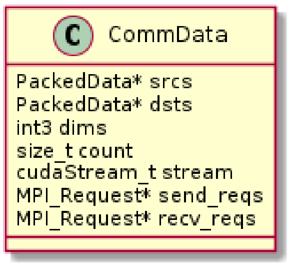
Listing 7.1: commit 3804e72, src/core/device.cc, lines 1450-1452, [27]. The coordinates $m1$ and $m2$ determine the data region that will be operated on, in this case the core. *isubstep* identifies the Runge-Kutta substep.

Halo exchange

Baseline Astaroth uses two data structures for communicating data from one process to another, *PackedData* and *CommData*. A *PackedData* contains a single message buffer. CommData is a structure used to manage multiple *PackedData*s. In the UML diagrams below (fig. 7.1), the type *int3* is a structure of three integers.



(a) PackedData



(b) CommData

Figure 7.1: Data structures used to communicate in baseline Astaroth

These data structures do not contain information about which data regions are stored in the buffers or where and how to communicate the data. This information is stored separately, in a set of coordinates that identify the data regions in the local grid. The coordinates are called b_0s.

A b_0 is an *int3* that points to the coordinate-wise least corner of the data region contained by a *PackedData*. Each *PackedData* contains packed data from a cuboid region of the grid which starts at some b_0 and extends to $b_0 + dims$. The b_0s are also used to determine which process a message buffer is sent to or received from.

CommData

A *CommData* contains every *PackedData* of a particular shape ($dims$). Therefore, there are (1) three *CommData*s for regions at the facets of the shell and halo, (2) three *CommData*s for regions along the edges of the shell and halo, and (3) one *CommData* for regions at the corners of the shell and halo. Together, the *CommData*s contain both the *incoming halo region family* and the *outgoing halo region family*. See section 4.6.

Each *CommData* has a corresponding array of b_0s, identifying each region of data that is contained in the *CommData*. The internal API for exchanging halos consists of a set of functions, shown in listing 7.2. As we can see, the functions take both a *CommData* and an array of *b0s*.

```
void
acPackCommData(const Device device, const int3* b0s,
               CommData* data);

void
acUnpackCommData(const Device device, const int3* b0s,
                 CommData* data);

AcResult
acTransferCommData(const Device device, const int3* b0s,
                   CommData* data);
```

Listing 7.2: commit 3804e72, src/core/device.cc, lines 1028-1101, [27].

During each iteration, for each *CommData* and its corresponding b_0 array: (1) the outgoing *PackedData*s contained in the *CommData* are prepared with a call to *acPack-CommData*; (2) the data is communicated (sent and received) with a call to *acTransfer-CommData*; and (3) the incoming *PackedData*s are unpacked into the halo with a call to *acUnpackCommData*.

7.2 The runtime

We are now ready to understand the Astaroth runtime. Algorithm 3 describes the runtime of the baseline version of Astaroth.

The compute tasks **C** from the ISL chapter do not correspond exactly to each call to *acDeviceIntegrateSubstep* but they perform the same work. The halo exchange tasks **B** do however roughly correspond to the *PackedData*.

Because the *CommDatas* contain the *PackedData*s and the compute work is performed in one go, I have chosen to represent the work in terms of **B** and **C**, since this makes the algorithm easier to compare to my treatment.

Algorithm 3 Baseline Astaroth's runtime

```
Inputs: An initial state $\mathbf{G}_0$, a map $f$, and a transition function $T$

Given $N$ processes $\{p_0, ...., p_{N-1}\}$:
Find a decomposition $\boldsymbol{d}$ for $N$ processes
$\mathbf{G} \leftarrow \mathbf{G}_0$
Assign subgrids $\mathbf{g}_0, ..., \mathbf{g}_{N-1}$ to processes according to $\boldsymbol{d}$ and map $\boldsymbol{f}$
for all distributed processes $p_i$, in parallel do
    for $timestep \leftarrow 1...M$ do

        for $s \leftarrow 1...3$ do
            MPI_Barrier( MPI_WORLD_COMM )
            $\mathbf{g}'_i \leftarrow T(\mathbf{g}'_i, \mathbf{C}[\mathbf{0}], s)$
            for all $b \in \mathbf{B}$ do
                Pack data from $\mathbf{g}'_i$ into $out_b$
                Determine neighbor $p_{ne}$
                MPI_Irecv($p_{ne}, in_b$)
                MPI_Isend($p_{ne}, out_b$)
            end for
            MPI_Waitall($\mathbf{B}$)
            for all $b \in \mathbf{B}$ do
                Unpack data from $in_h$ into $\mathbf{g}'{}_i$
            end for
            for all $c \in \mathbf{C}$ do
                $\mathbf{g}_i \leftarrow T(\mathbf{g}'_i, c, s)$
            end for
            Swap buffers
            Synchronize CUDA device
        end for

    end for
end for
```

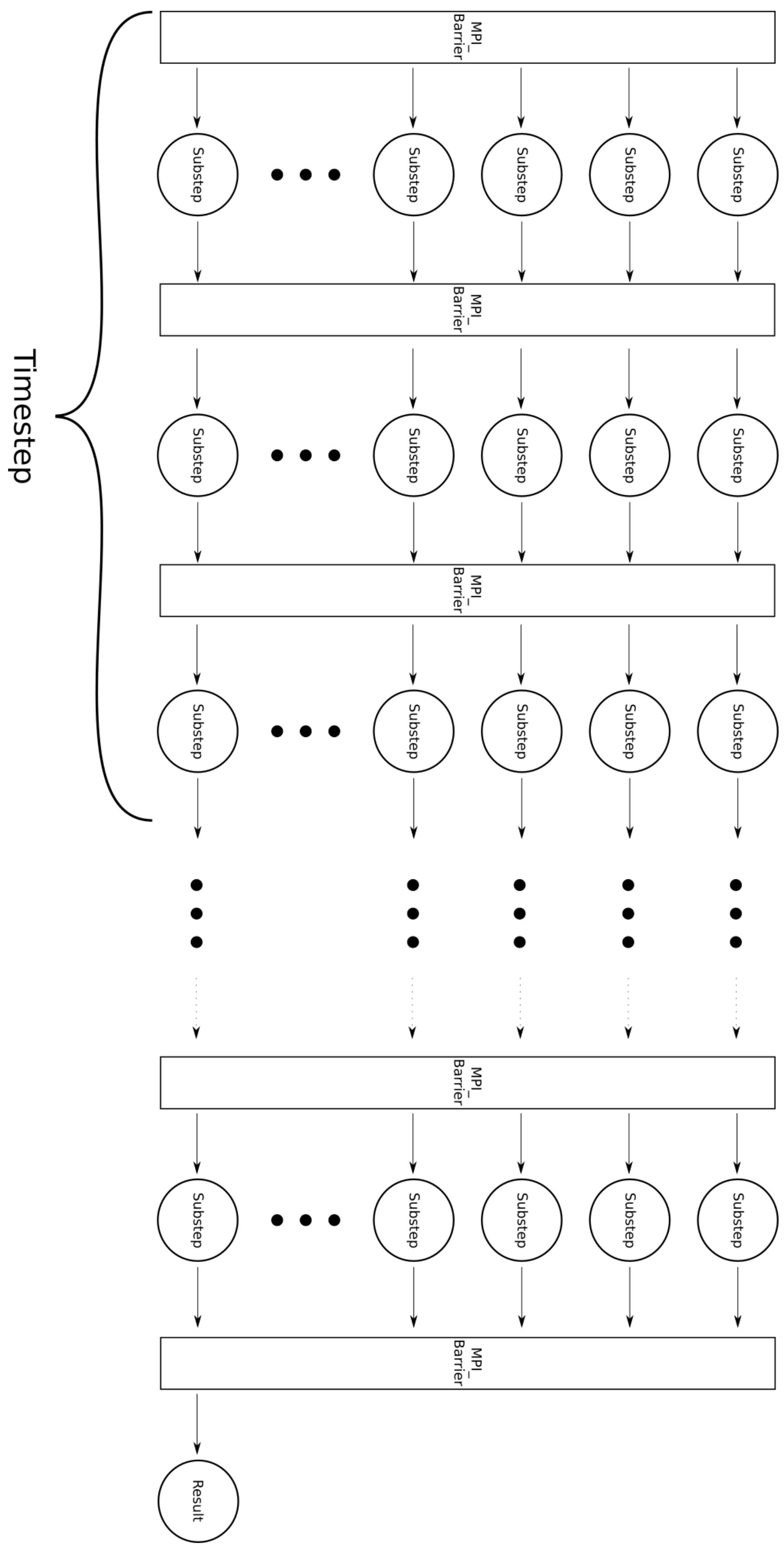


Figure 7.2: Dependency graph for a whole simulation. The barriers cause fork-join (synchronous) behavior.

The algorithm is synchronous

Because the RK-scheme used is a three-step scheme, a single timestep consists of three iterations of calculation and halo exchange. The entire runtime of a simulation can, however, consist of an arbitrary number of iterations. In figure 7.2, I have drawn a dependency graph that illustrates such a case.

Let us consider, then, an Astaroth runtime that runs for an arbitrary number of iterations. If we take a closer look at the algorithm, we can see that it exhibits fork-join behavior. That is to say, all processes need to reach the *MPI_Barrier* before a single one of them can proceed. This means that the performance of an entire distributed iteration is limited by the performance of the slowest process.

This effect will be exacerbated both if the number of iterations is increased and if the number of processes is increased. This synchronous behavior therefore constrains the scalability of Astaroth.

The global synchronizations (*MPI_Barrier*s) in baseline Astaroth are in place to ensure that data dependencies have been satisfied before program execution continues. A global synchronization, however, is a little overzealous, as it blocks until *all* dependencies are satisfied. That is to say, even if progress would be possible for a subset of stencil calculations or halo exchanges, all execution is blocked until the synchronization is complete.

There is, however, an alternative to global synchronization. We can use a system to keep track of individual dependencies and schedule tasks as their dependencies become satisfied. This is the approach I have taken in my treatment, which is the topic of the next chapter.

8
A Task Graph Scheduler for Astaroth

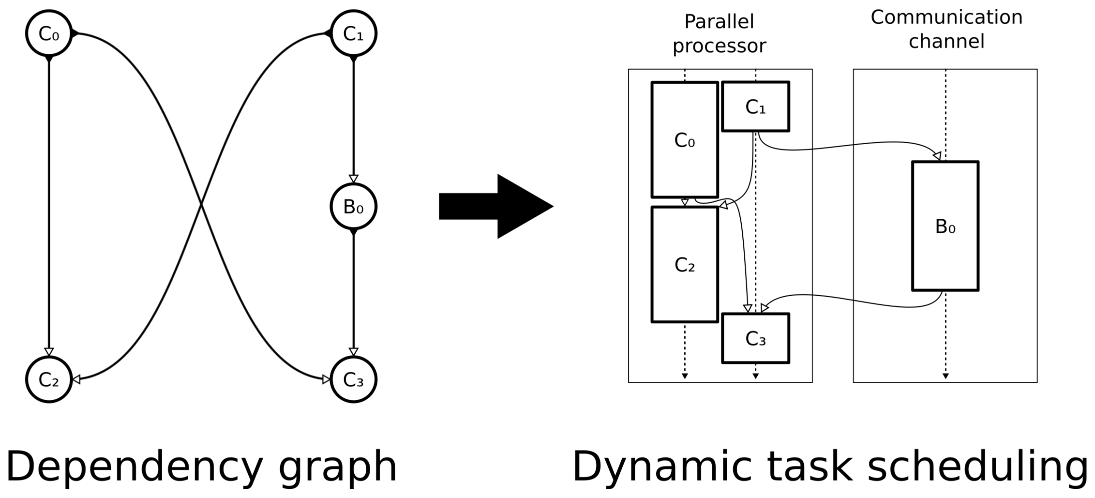


Figure 8.1: A conceptual diagram illustrating how the task graph scheduler works.

In this chapter, I will present an integrated task graph scheduler that I've developed for Astaroth. The version of the task scheduler covered in this section is in commit *56bb4b3* in Astaroth's online repository [27].

Stencil calculation and halo exchange work has been encapsulated into tasks. At initialization time, a dependency graph consisting of these tasks is built, based on the rules derived in chapter 4. Then, once running, tasks are dynamically scheduled for execution when their prerequisite tasks have finished. At no point is a global synchronization necessary. The idea is captured by the diagram in figure 8.1.

Removing global synchronization has two effects. Most importantly, it eliminates waiting time. A more subtle effect is that the workload is more balanced.

Why is the workload more balanced in an asynchronous schedule? A global synchronization causes work to queue up in front of resources in a burst. These bursts lead to a pattern of fluctuation between starvation and excess load. By scheduling tasks greedily, their execution periods are more evenly distributed over the runtime of a simulation, which means that the mean queue length L is decreased. Since the effective arrival rate λ doesn't decrease, then by Little's Law, $W = \frac{L}{\lambda}$, this must decrease the mean queueing time W [52].

I will begin the chapter with an argument for why HPC applications benefit from task-scheduling systems (specifically at large scales). I will then move on to describe the specification and design of the task scheduler I've built for Astaroth.

8.1 The case for task schedulers

In their 2018 paper, "A taxonomy of task-based parallel programming technologies for high-performance computing", Thoman et al. define a task as follows [16, p. 2]:

> A **task** is a sequence of instructions within a program that can be processed concurrently with other tasks in the same program. The interleaved execution of tasks may be constrained by control- and data-flow dependencies.

That is to say, a task is a convenient element of work that can be scheduled, queried, and run concurrently with other tasks. If we want to be able to dynamically schedule work in a program, something like a task scheduler is necessary.

But why are task schedulers relevant to HPC? When running a program on a large cluster, the wall clock times of runs vary significantly. The main source of variance is the network, which is a shared resource with varying load. As we increase the number of parallel processes used by a distributed program, the maximum wall clock time — the processing time of the slowest process (or task) — will tend to increase[1].

This increases the cost of global synchronization, since all processing elements must wait for the slowest one. It is clear that allowing processes to make progress asynchronously will benefit performance in systems with multiple processes. Some task schedulers even include another feature that addresses this problem: dynamic load balancing, moving tasks from slower processing element to faster ones [54].

Now consider that any sufficiently successful program will eventually be ported to a new cluster, where certain tasks are faster and others slower. This is true even if the software stays within an organization, as clusters are phased out and replaced. On a new cluster, most old knowledge about the relative performance of tasks is no longer applicable, and the schedule must be retuned.

We cannot statically determine a schedule that will always be the fastest. The optimal schedule is a function of the cluster architecture and the system load: there is no single optimal schedule for a set of dependent tasks.

If we can find the optimal task schedule for any system load and cluster architecture, there are two major benefits to reap: (1) performance will be adaptive and will respond

[1]This is a result from extreme value theory. If we scale up indefinitely, the distribution of the maximum wall clock time will eventually approach a generalized extreme value distribution, a very fat-tailed distribution [53].

better to changes in system load and transient network errors, (2) as long as the application can be functionally ported to a new system, the performance of the system will be portable as well (assuming no catastrophic performance degradation effects outside of scheduling). This second property is called performance portability [55].

Unfortunately, it has been shown that it is impossible for an online scheduler to always find the best schedule [56]. Nevertheless, we can adjust our aspirations and simply search for a method of finding a *good-enough* schedule for any system load and cluster architecture.

Several current HPC libraries and runtime systems are built around the idea of dynamic task scheduling. Examples of such systems are OCR [14], Legion [13], and HPX [54].

Task-based approaches have a proven track record of improving the performance of HPC applications. Examples include PLASMA, a sofware package for linear algebra problems [57]; SWIFT, a smooth particle hydrodynamics (SPH) solver [58]; and Octo-Tiger, a self-gravitating astrophysical fluid simulation system using adaptive mesh refinement (AMR) [15]. Of these three, Octo-Tiger resembles Astaroth the most, as it is also a stencil solver.

8.2 A specification of Astaroth's task scheduler

The task scheduler I've built is integrated into Astaroth and is tightly coupled to Astaroth. In Thoman et al.'s taxonomy [16], my task scheduler would be classified as: (1) representing dependencies in an arbitrary graph through explicit declaration, (2) mapping work explicitly, and (3) scheduling tasks dynamically (but with static processor assignment).

A short functional specification of the task scheduler follows.

Multistage tasks Tasks may have multiple stages. A multistage task consists of a pipeline of subtasks that operate on the same data, with no external dependencies. Tasks can be queried to check which stage they are at.

Asynchronous progress The scheduler can query a task for the completion of the current stage and advance the task to the next stage. This requires no synchronization.

Dynamic scheduling The scheduler can schedule tasks in an arbitrary order at run-time.

Task representation The work done by Astaroth can be represented as tasks, so that the scheduler can interface with them. In other words, we need to be able to create tasks that execute compute kernels and exchange halos.

The dependency graph Dependencies between tasks can be represented as a graph. Once running, tasks will signal dependent tasks that a dependency has been satisfied. If all dependencies of a task are satisfied, the scheduler can schedule the task for execution.

Independent tasks Tasks are able to execute independently from one another, except for explicit dependencies.

Iteration The same tasks and dependencies can be reused for many iterations. Only one iteration of a task may be active at any time. Dependencies are interpreted relative to the iteration that is currently executed. If this results in a task interpreting that it depends on a task from before the start of execution, the dependency is considered satisfied by definition.

8.3 Task objects

There are two types of tasks: *compute tasks* and *halo exchange tasks*. The tasks have a common interface (fig. 8.2), which allows the scheduler to interact with them.

Tasks have multiple *stages* and transition from one stage to the next. The stages are implemented as states in a state machine. All tasks start off in a special waiting state. They can only be transitioned out of this state once all their dependencies have been satisfied.

Transitions between subsequent (non-waiting) stages happens once a stage has been completed. The completion of stages is detected by polling the resources that the stage has been using.

A task that reaches its final stage will loop back to the waiting state. Once a task loops back to the waiting state, it has finished one iteration. This means that the dependency that the task represents has been satisfied and all its dependents are notified of this.

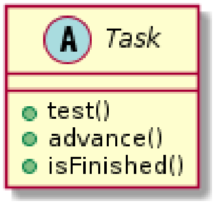


Figure 8.2: A task's scheduling interface

Algorithm 4 Astaroth's runtime using the task scheduler

```
Inputs: An initial state $\mathbf{G}_0$, a map $f$, and a transition function $T$

Given $N$ processes $\{p_0, ...., p_{N-1}\}$:
Find a decomposition $\boldsymbol{d}$ for $N$ processes
$\mathbf{G} \leftarrow \mathbf{G}_0$
Assign subgrids $\mathbf{g}_0, ..., \mathbf{g}_{N-1}$ to processes according to $\boldsymbol{d}$ and map $f$
for all distributed processes $p_i$, in parallel do
    Build the task graph $\mathbf{W}$
    for $timestep = 1$ to $M$ do
        for all $w \in \mathbf{W}$ do
            $w.loop\_counter \leftarrow 0$
        end for
        repeat
            $ready \leftarrow$ true
            for all $w \in \mathbf{W}$ do
                update($w$)
                $ready \leftarrow w.isFinished() \wedge ready$
            end for
        until $ready$
    end for
end for

procedure update(w) $\equiv$
Inputs: A task $w$. The two member functions: test and advance, are specific to the task type

if $w.isFinished()$ then
    return
end if
if $w.state =$ waiting then
    $t \leftarrow w.dependency\_counter.$is_finished$(w.loop\_counter)$
else
    $t \leftarrow w.$test$()$
end if
if $t$ then
    $w.$advance$()$
    if $w.state =$ waiting then
        for all $d \in w.dependents$ do
            $d.$notify$()$
            $w.loop\_counter \leftarrow w.loop\_counter + 1$
        end for
    end if
end if
```

8.4 The runtime

Algorithm 4 describes the runtime of Astaroth, using the task scheduler. The central loop of the algorithm is the actual task scheduler. The algorithm runs a simulation with a three-step Runge-Kutta scheme for some number of timesteps M.

8.5 The task scheduler

The scheduler will poll the tasks in a round robin busy loop and immediately transition them forward if possible. A task is polled by calling its *test* method, and stage transitions are triggered by a call to a task's *advance* method. The simulation will thus make progress where possible, not necessarily optimally — but greedily.

This type of scheduler is known as a *list scheduling algorithm*. In 1966, Graham [59] showed that list scheduling algorithms produce schedules that are at least half as fast as the optimal schedule [2].

8.6 Internal task state

Each task is assigned an output region. The regions id identifies the task.

Each task has a loop counter which is incremented when the task loops back to the waiting state. When the loop counter reaches the end of the loop, the task's *isFinished* method will return true.

Each task has a dependency counter which is incremented when the task gets a notification from a prerequisite task. Each task also keeps a list of all its dependents, so that they can be notified when the task finishes an iteration. These data members are the representation of the dependency graph in the scheduling system. They are populated at initialization time.

Double buffering

Astaroth uses double buffered resources that it swaps between iterations, e.g., VBAs. Tasks keep their own references to all double buffered resources. Each task can then swap their own buffers, which eliminates a dependency on shared state.

In appendix B, I prove that this double buffering is enough because the task dependencies include all possible buffer dependencies.

[2]The actual bound is $2 - \frac{1}{n}$ times the optimal schedule time, where n is the number of processors.

8.7 Compute tasks

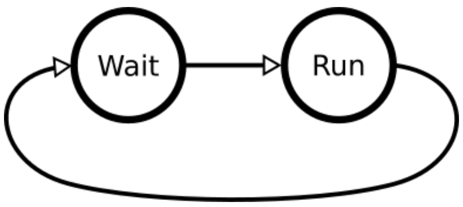


Figure 8.3: The state machine of a compute task.

The compute task is simple and has only one active stage, *Run*, as illustrated by figure 8.3. The *Run* stage simply wraps a call to a GPU kernel that performs a stencil calculation over a region of data.

The GPU kernel needs the coordinates of the output region and the Runge-Kutta substep currently being executed (one of 0, 1, or 2). The output region is a data member of the task, and the Runge-Kutta substep is calculated from the loop counter's value.

Each compute task has a dedicated CUDA stream. A dedicated CUDA stream allows compute tasks to run concurrently. It also allows the *Run* stage to be polled by a call to *cudaStreamQuery*.

8.8 Halo exchange tasks

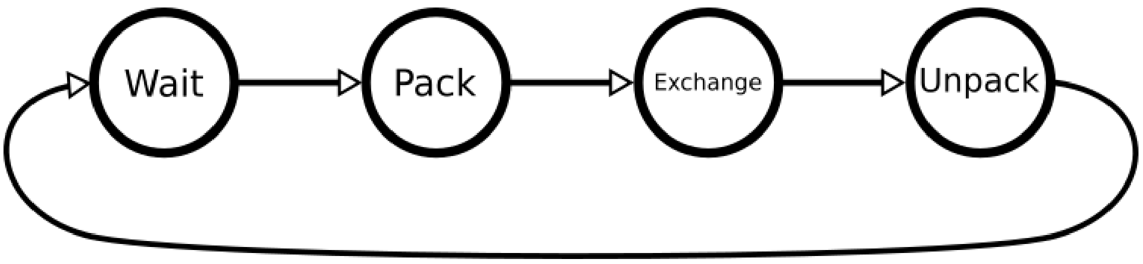


Figure 8.4: The state machine of a halo exchange task

A halo exchange task is more complicated than a compute task. A halo exchange task has multiple stages and consists of two separate operations: a send and receive.

Both sending and receiving is grouped into one task because it is more practical. The grouping makes dependencies between halo exchange tasks and compute tasks symmetric [3].

A halo exchange task has three stages: *Pack*, *Exchange*, and *Unpack*.

[3]See the chapter on ISLs, section 4.6.

In the *Pack* stage, outgoing data is packed from the grid into a buffer. In the *Exchange* stage, an outgoing MPI message is sent and an incoming MPI message is received. In the *Unpack* stage, data is unpacked from the received MPI message buffer into the grid.

While all tasks swap VBAs between iterations, halo exchange tasks also manage message buffers and MPI requests. Message buffers and MPI requests are paired up. The pairs are managed as a double buffered swap chain, swapped between iterations. A halo exchange task both sends and receives MPI messages, so there are two of these swap chains per task.

Polling is handled differently depending on the stage. The *Pack* and *Unpack* stages are performed on the GPU and are polled by calling *cudaStreamQuery*. For the *Exchange* stage, the MPI receive request is polled with *MPI_Test*. Send requests are not completed[4].

Receive optimization

The halo exchange tasks include a special optimization, separate from the scheduler.

As long as a matching MPI receive request has been posted locally, an incoming message will be placed in it once it arrives. However, if there is no matching receive request, an incoming message is placed in an *unexpected message queue* which incurs an overhead (as discussed in the MPI chapter, section 5.1).

There is a simple way to remove this inefficiency, we simply post the receive as early as possible. The first receive request is posted immediately at initialization time. Subsequent receive requests are posted every time a receive request is completed by an incoming message. In this way, there is always a receive buffer waiting for incoming messages that the MPI runtime can write to.

8.9 Task iteration

Running asynchronously for only one iteration would not have a large impact on performance, since Astaroth would then be forced to synchronize after every iteration. The task scheduler therefore supports iteration. At the moment, the new version of Astaroth performs three iterations at a time and then synchronizes, because this is the number of substeps in the Runge-Kutta scheme used by Astaroth.

We can think of this as each task running its own for-loop and coordinating its state with the other tasks. The for-loop is declared by calling the *setIterationsParams* method of a task.

This initializes the state of each task's internal loop counter. A task's loop counter is

[4]Until they are reused in the swap chain, at which point they are guaranteed to have arrived.

used to coordinate dependency notification between tasks and to determine when the task has finished all of its iterations.

8.10 Dependencies

A dependency is a relationship between a prerequisite task and a dependent task. The dependencies are based on the rules derived in the ISL chapter, section 4.6.

Dependencies between tasks are static, they do not change from iteration to iteration. Dependencies can therefore be registered at initialization and are reused for every iteration of the simulation.

Some dependencies will extend across iterations (see fig 8.5). This information needs to be included in the dependency representation. I use an offset, which has a value of either 0 (same iteration) or 1 (one iteration earlier).

All tasks keep a list of their dependents. The offset is kept with the dependent. When a task finishes an iteration, it notifies its dependents and includes the offset in the notification.

All tasks also have a dependency counter. The dependency counter keeps track of dependencies for multiple iterations of the task. When a task receives a notification of a satisfied dependency, the offset is used to work out which iteration the dependency was for, and the dependency counter is incremented. Once the dependency counter reaches its target count for an iteration, the task is ready to start its next iteration.

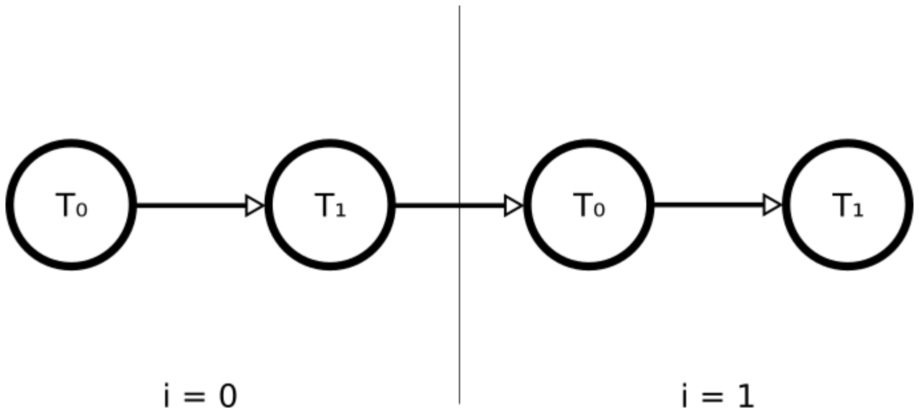


Figure 8.5: Example of a dependency crossing iteration boundaries. The dependency of T_1 on T_0 is in the same iteration; that dependency therefore has an offset of 0. The dependency of T_0 on T_1 is across iterations; that dependency therefore has an offset of 1.

9
Performance and Scalability Comparison

I have compared the performance of baseline Astaroth (commit *3804e72*) with the performance of the task scheduler (commit *56bb4b3*). The benchmark is one timestep of a magnetohydrodynamics simulation. The simulation uses all the governing equations from the magnetohydrodynamics chapter, 6th-order accurate central difference approximations, and the three step Runge-Kutta method. The simulation solves for two three-dimensional vector fields, velocity $\vec{u}$ and magnetic vector potential $\vec{A}$; and two scalar fields, logarithmic density $\ln\rho$ and specific entropy s. There are therefore eight vertex buffers altogether.

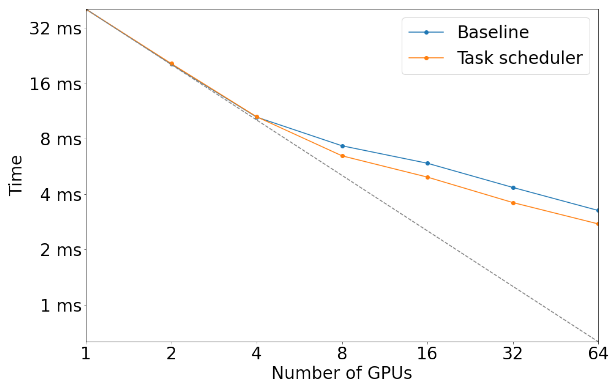
Benchmarks are measured using an instrumented program with a high precision timer. The program runs the benchmark a large number of times[1]. The program reports the minimum, mean, 90th percentile, and maximum values of the distribution of wall clock times. I have used the 90th percentile value as the metric in my results.

Scaling was measured from a single GPU up to 64 GPUs. Strong scaling benchmarks were run on grids of size 256^3, 512^3, and 1024^3. Weak scaling benchmarks were run on grids of size 128^3, 256^3, and 512^3.

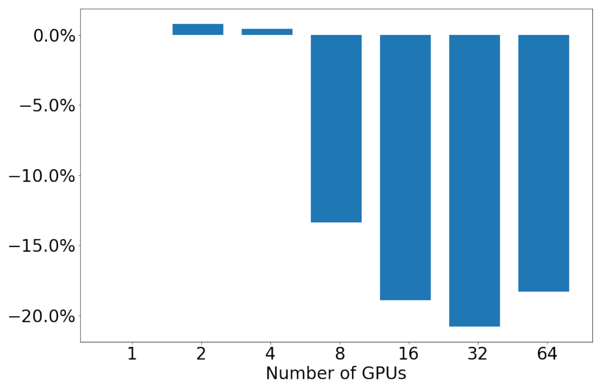
9.1 Benchmark environment

The benchmarks were run on the *gpu* partition of CSC's Puhti cluster [36]. The partition consists of 80 SuperServer 1029GQ-TVRT nodes in a fat tree topology. The nodes connect to the network via dual-rail Mellanox ConnectX-6 InfiniBand HDR100 MT28908 NICs. The chipset is Intel PCH C621 (Lewisburg). Each node has two CPUs (Intel Xeon Gold 6230 Cascade Lake) and four GPUs (Tesla V100-SXM2-32GB GV100GL). All benchmarks were run with Open MPI version 4.0.3, UCX version 1.8, and GDRCopy version 2.1.

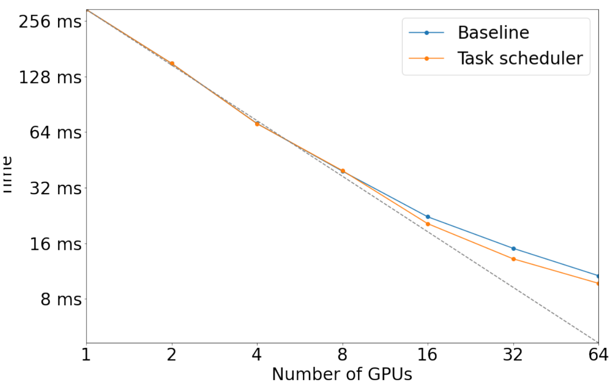
[1]This is the same benchmark program that was used by Pekkilä to produce the results in [8]. You can find it in the Astaroth git repo[27] (either commit), the filepath is "samples/benchmark/".



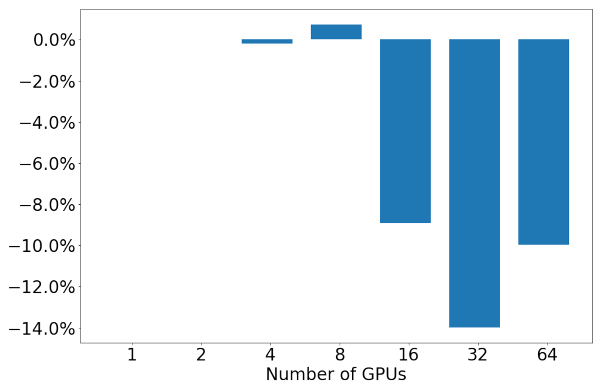
(a) Wall clock times for strong scaling of a 256^3 grid.



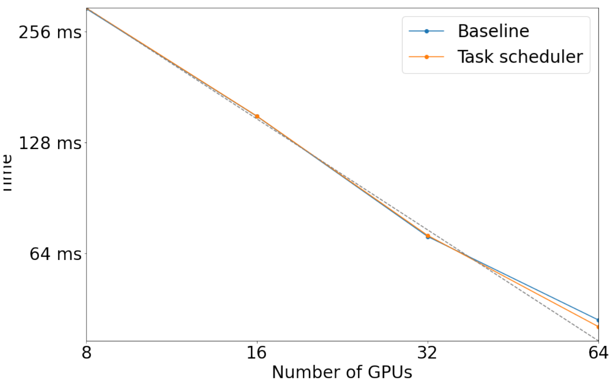
(b) Change in wall clock time relative to baseline. 256^3 grid.



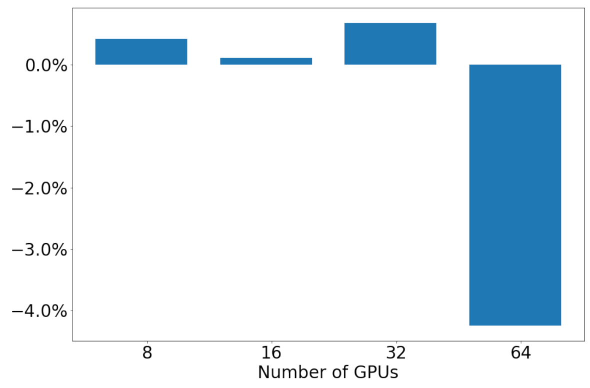
(c) Wall clock times for strong scaling of a 512^3 grid.



(d) Change in wall clock time relative to baseline. 512^3 grid.



(e) Wall clock times for strong scaling of a 1024^3 grid.



(f) Change in wall clock time relative to baseline. 1024^3 grid.

Figure 9.1: Strong scaling results. Values are based on the $90th$ percentile of benchmark measurements. The 1024^3 grid is too big to fit into the memory of four GPUs, which is why strong scaling measurements start at eight GPUs for that grid size.

9.2 Strong scaling results

Strong scaling measures how performance evolves when the total work remains constant and we increase the number of GPUs. The strong scaling results are shown in figure 9.1. Note that the graphs have a log-log scale.

By isolating the communication and computation, we can reason about when the system performs better. I have compared the strong scaling of isolated communication and computation in figure 9.2. Here we can see that the task scheduler starts performing better when communication becomes the performance constraint.

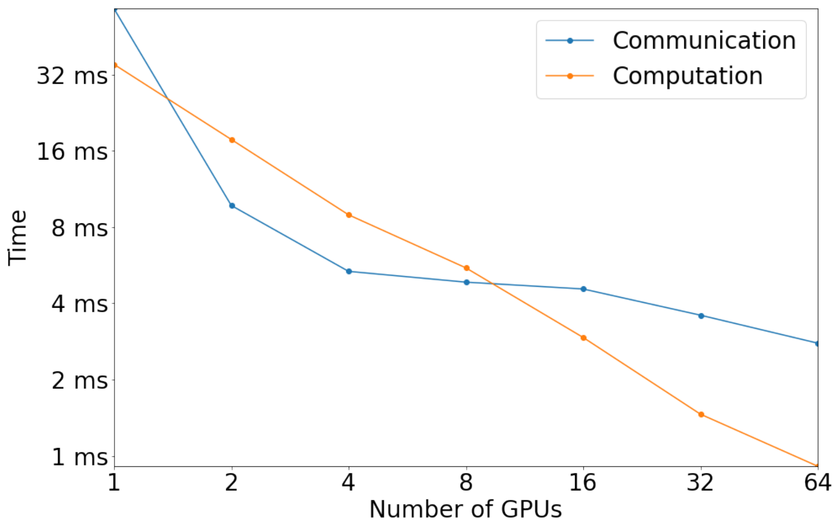


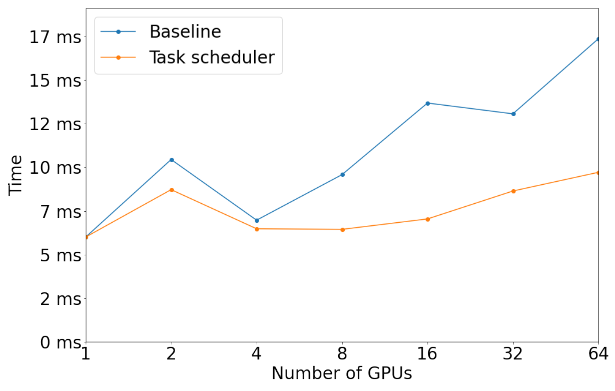
Figure 9.2: Strong scaling of communication and computation. 256^3 grid.

We see that the performance of the task scheduler is up to 20% better than the baseline when communication performance constrains the overall performance. When compute performance is the constraint, overall performance is roughly equal.

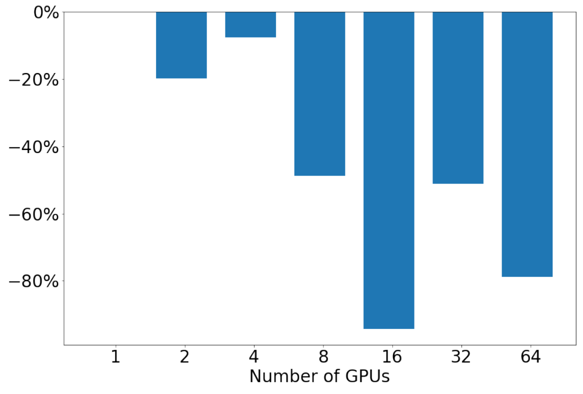
9.3 Weak scaling results

Weak scaling measures how performance evolves when the work per GPU remains constant and we increase the number of GPUs – the total work is proportional to the number of GPUs. The weak scaling results are shown in figure 9.1 on the next page. The y axis is linear in the weak scaling figure.

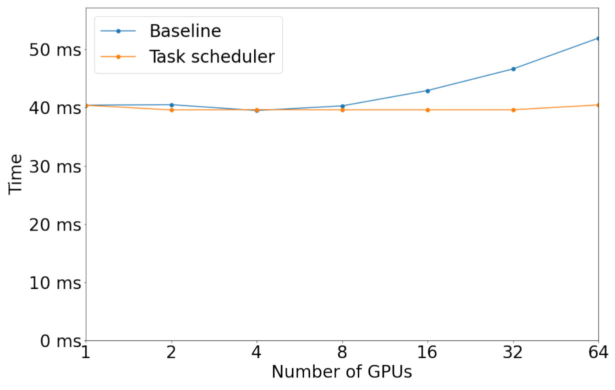
Again, we see that the improvement is more pronounced when the operational intensity is low.



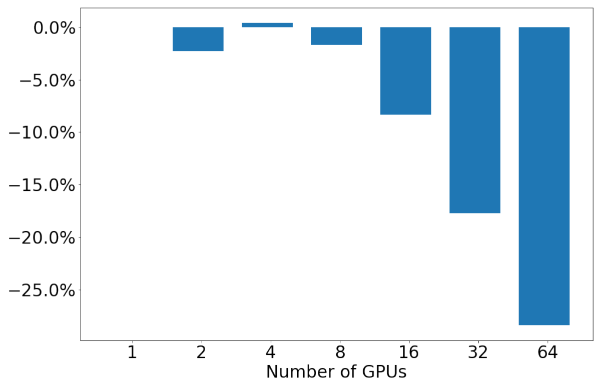
(a) Wall clock times for weak scaling of a 128^3 grid.



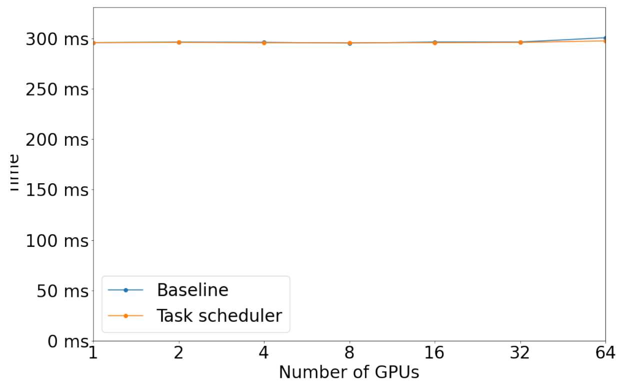
(b) Change in wall clock time relative to baseline. 128^3 grid.



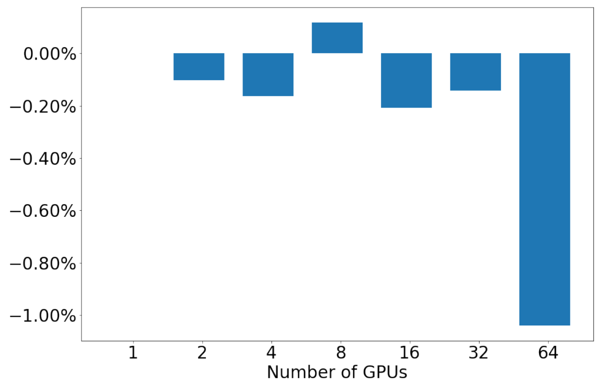
(c) Wall clock times for weak scaling of a 256^3 grid.



(d) Change in wall clock time relative to baseline. 256^3 grid.



(e) Wall clock times for weak scaling of a 512^3 grid.



(f) Change in wall clock time relative to baseline. 512^3 grid.

Figure 9.3: Weak scaling results. Values are based on the $90th$ percentile of benchmark measurements.

At a local grid size of 128^3, when communication severely constrains performance, the baseline scales poorly. The task scheduler is not perfect either, but weak scaling is much better using the new version, up to 80% better.

At a grid size of 256^3, we see that the baseline seems to develop a smaller scaling overhead, while the task scheduler has near-perfect weak scaling to 64 GPUs.

At a grid size of 512^3, both versions have near-perfect weak scaling to 64 GPUs.

10 Discussion

Unfortunately, we are only able to scale up to 64 GPUs on Puhti. There is, however, some evidence that the task scheduler scales better than baseline Astaroth: the fact that the task scheduler performs better for cases where communication is the performance constraint. The weak scaling of the task scheduler is also better than that of the baseline.

As we saw in figure 9.2, communication scales at a slower rate than computation: communication is the scalability bottleneck. Because the task scheduler improves the performance of the scalability bottleneck, the overall scalability of Astaroth should also be improved.

Why has the task scheduler improved the performance of Astaroth specifically for communication-constrained workloads? I believe the reason for this is that the performance of communication is much more variable than the performance of computation.

The performance of each individual compute task is nearly constant. An optimal schedule for compute tasks likely does not change at all from run to run. Therefore, a static schedule is suitable for compute-constrained cases[1].

When the critical path consists of tasks whose performance is less predictable, however, the optimal schedule varies from run to run. This is why a static schedule is ill-suited for communication-constrained runs, the dynamic task graph scheduler performing better in these cases.

By generalizing the argument that the task scheduler performs better when the critical path is not static, we can reason about the performance portability of the task scheduler. I speculate that the task scheduler is likely to have better performance portability than the static schedule, because the optimal schedule will vary as a function of the underlying hardware.

[1]If the schedule is run on clusters where that specific schedule performs well.

10.1 Further work

Performance portability

I would like to test the hypothesis of performance portability by running the two versions of Astaroth on other clusters. I am particularly interested in how they would perform on CSC's soon-to-be-inaugurated LUMI cluster [60]. It would be a boon to the Astaroth team if the task scheduler reduced the cost of porting Astaroth to new clusters.

Automatic dependencies

The current version of the task scheduler uses region ids to generate dependencies according to the rules derived in the ISL chapter (sec. 4.6). This limits the sets of tasks that can be run on Astaroth. The assumption of alternating computation and communication is baked into this method.

Future versions of the task scheduler could use the geometry of task regions to generate dependencies automatically. This would allow for multiple kernels to be run before communicating.

If the compute kernels only touch a subset of the simulated fields, then some kernels may be independent of each other. Future versions of the task scheduler could use information generated by the DSL compiler to reason about variable-specific dependencies between tasks.

CPU tasks

It is important to note that the task scheduler runs on a single thread and is designed for task stages that can be launched asynchronously. If the main scheduling thread must block while waiting for the task to complete, the scheduler would not work as intended. Luckily, both CUDA calls and MPI calls offload the work from the CPU. If they didn't, asynchronous interfaces would have to be created specially. Offloading is also important, since it keeps the scheduling thread busy.

In the future, the task scheduler could be used to write out simulation data for analysis. This would probably involve some CPU task. If so, offloading to another core would have the same effect.

Astaroth's API design constrains performance

The interface of Astaroth is written such that each timestep is performed by a separate call to the same function. This limits the number of asynchronous iterations that the task scheduler can run to three (or however many steps there are in the time integration scheme).

A future interface may take the number of timesteps to run as an argument and run for that many timesteps. The task scheduler would easily support this change. This interface would result in fewer synchronizations and likely better performance.

11
Summary in Swedish

Schemaläggning av uppgiftsgrafer i det astrofysiska simulationsverktyget Astaroth

Beräkningsforskare studerar komplexa fenomen genom att utföra numeriska experiment i datorsimulationer. Som exempel studerar beräkningsastrofysiker astrofysisk magnetism genom att simulera strömningen av magnetiskt material i stora rymdkroppar såsom stjärnor. Denna intersektion av strömningsdynamik och magnetism kallas magnetohydrodynamik (MHD).

Speciellt när det gäller direkt numerisk simulering (DNS) av turbulenta strömningsproblem innehåller datorsimulationer en väldig mängd beräkningsarbete [18]. Det kan ta många veckor att köra simulationer på ett kluster av specialutrustade datorer. Många högeffektiva datorberäkningskluster använder i dagens läge parallellprocessorer (grafikprocessorer) och högeffektiv nätverksteknologi för att accelerera beräkningsarbetet.

Beräkningsarbetet är distribuerat över flera datorer i klustret, och processer i skilda noder behöver kommunicera med varandra. Kommunikationen sker via standardkommunikationsgränssnittet MPI (eng. Message Passing Interface) [40]. Kommunikationsarbetet utgör ofta en flaskhals vid stora arbetsmängder.

Astaroth [27] är ett programvarupaket som beräkningsfysiker använder för att 1) definiera beräkningskärnor i ett domänspecifikt programmeringsspråk och 2) köra dessa beräkningskärnor på datorkluster med grafikprocessorer. Astaroth baserar sig på Fortran-baserade simulationsbiblioteket *Pencil Code* [22].

Tidigare versioner av Astaroth har redan bra prestanda vid låga processantal [8], men kommunikationsprestandan försämras när processantalet ökar. Min målsättning för detta diplomarbete var att skapa en version av Astaroth vars prestanda är skalbar till flera processer. Eftersom kommunikationsarbetet utgör en flaskhals har jag fokuserat på att förbättra Astaroths kommunikationsprestanda.

MHD-simulationer styrs av ett antal partiella differentialekvationer som beskriver tillståndet av varje punkt i simulationsdomänen [26]. De simulationer som har modellerats med hjälp av Astaroth använder en differensmetod för att approximera derivatorna i differentialekvationerna. Domänen har diskretiserats, transformerats till ett regelbundet rutnät, och för varje punkt i rutnätet approximeras värdet av derivatorna med hjälp av en differenskvot av värden i närliggande punkter. Beräkningen utförs sedan över hela rutnätet samtidigt. Det behövs många steg av beräkning, eftersom vi simulerar utvecklingen av ett strömningsfält över en tidsperiod.

När simulationensarbetet distribueras till flera processer uppdelas beräkningsdomänen så att vardera process utför beräkningen för en viss dataregion i domänen. För randpunkter i processberäkningsregionerna utsträcker sig differensmetodens databehov till gränsande beräkningsregioner. Därför behövs ett kommunikationssteg mellan varje beräkningssteg.

I kommunikationssteget överför varje process randdata till processer vars beräkningsregioner gränsar till den sändande processens beräkningsregion. Mottagna randdata formar en datakrans (eng. halo) runt beräkningsregionen av en process. Kommunikationssteget kallas därför kransutbyte[1] (eng. halo exchange).

Kransutbytet består alltså av en mängd kommunikationsuppgifter. Uppgifterna utförs över ett nätverk, och tiden det tar att utföra en uppgift varierar enligt nätverkets tillstånd. På grund av denna variation är det svårt att bestämma ett optimalt schema för arbetsuppgifterna.

I den tidigare versionen av Astaroth utfördes arbetet alltid i samma ordning, på följande vis: Nya värden för den inre delen av domänen, som inte beror av kransdatan, beräknas samtidigt som kransutbytet utförs. När kransutbytet är färdigt beräknas även nya värden för randregionerna. Sedan väntar Astaroth på att all beräkning är slutförd och börjar med nästa iteration [8].

Denna algoritm utnyttjar inte all beräknings- och kommunikationskapacitet, eftersom den innehåller två globala synkroniseringar per iteration. När Astaroth synkroniserar blockeras exekvering av programmet tills alla arbetsprocesser har nått synkronisationspunkten.

Synkronisering säkerställer att databeroendena av efterkommande uppgifter är tillfredsställda innan exekveringen fortsätter. Men en global synkronisering är mer än vad behövs. Vissa uppgifters databeroenden tillfredsställs innan synkroniseringen är slutförd.

För att åtgärda detta har jag uppdelat processberäkningsregioner i uppgiftsregioner och bestämt beroendeförhållanden dem emellan. Med den informationen kan vi bygga upp en uppgiftsgraf. Uppgiftsgrafen beskriver allt arbete som behöver utföras i en simulation, och hur uppgifterna beror av varann.

I min prestandabehandling av Astaroth har jag sedan skrivit en schemaläggningsalgoritm. I algoritmen pollas uppgifterna för att bestämma om de är slutförda, och denna

[1]Skribentens egen översättning.

information används för att reda ut vilka uppgifters databeroenden är tillfredsställda. När en uppgifts databeroenden är tillfredsställda påbörjas uppgiften direkt. Schemaläggningsalgoritmen är alltså en så kallad girig algoritm.

Jag har jämfört prestandan av versionen av Astaroth som använder schemaläggningsalgoritmen och versionen av Astaroth med ett statiskt schema. I prestandajämförelsen framkommer det att schemaläggningsalgoritmen som baserar sig på uppgiftsgrafen har bättre skalbarhet än versionen med statiskt schema. Skillnaden är märkvärd och beskrivs bäst av figurerna 9.1 och 9.3.

I figur 9.1 har jag jämfört versionernas *starka skalbarhet*, d.v.s. hur prestandan utvecklas när den totala arbetsmängden förblir konstant och antalet processer ökas. I figur 9.3 har jag jämfört versionernas *svaga skalbarhet*, d.v.s. hur prestandan utvecklas när arbetsmängden per process förblir konstant och antalet processer ökas.

Både de svaga och starka skalbarhetsegenskaperna är bättre hos versionen med uppgiftsgrafsbaserad schemaläggning. Prestandan verkar förbättras mer ju större del av det totala arbetet består av kommunikation. Jag tror att orsaken till detta är den större variansen i kommunikationsprestanda jämfört med beräkningsprestanda.

Vidare forskning: prestandamätning på LUMI

Prestandamätningarna kördes på datorklustret Puhti [36]. På Puhti var det inte möjligt att använda fler än 64 grafikprocessorer åt gången. CSCs kommande datorkluster LUMI [60] borde ha flera storleksordningar mer beräkningskapacitet än vad Puhti har. På LUMI är det alltså möjligt att göra en mycket utförligare skalbarhetsanalys.

Dessutom har grafikprocessorerna på LUMI olika tillverkare och arkitektur än Puhtis grafikprocessorer[2]. LUMI utgör alltså även ett tillfälle att analysera portabiliteten av Astaroth och schemaläggningsalgoritmen.

Vidareutveckling: stöd för mer generella uppgiftsgrafer

Analysen av beroendeförhållanden antar för tillfället alternerande beräkning och kommunikation. Det finns vissa simulationstyper som utför lokala beräkningar vars resultat inte behöver kommuniceras. I sådana fall behövs ett nytt sätt att generera uppgiftsgrafen.

Ett sätt att göra det är att direkt jämföra inputregioner och outputregioner av uppgifterna variabelvis. Så gör exempelvis programvarupaketet Legion [13]. För detta behövs information om vilka input- och outputvariabler varje kompilerad beräkningskärna har. Då behövs ändringar i kompilatorn för Astaroths domänspecifika programmeringsspråk.

[2]Puhti använder Nvidias Volta-grafikprocessorer [36], LUMI kommer att använda AMDs Instinct-grafikprocessorer [60].

References

[1] M. S. Väisälä, J. Pekkilä, M. J. Käpylä, M. Rheinhardt, H. Shang, and R. Krasnopolsky, “Interaction of Large- and Small-scale Dynamos in Isotropic Turbulent Flows from GPU-accelerated Simulations,” *The Astrophysical Journal*, vol. 907, no. 2, 83, p. 83, Feb. 2021. arXiv: `2012.08758`.

[2] J. Pekkilä, M. S. Väisälä, M. Käpylä, P. J. Käpylä, and O. Anjum, “Methods for compressible fluid simulation on GPUs using high-order finite differences,” *Comput. Phys. Commun.*, vol. 217, pp. 11–22, 2017. [Online]. Available: `https://arxiv.org/abs/1707.08900`.

[3] S. H. Fuller, L. A. Barroso, R. P. Colwell, W. J. Dally, D. Dobberpuhl, P. Dubey, M. D. Hill, M. Horowitz, D. Kirk, M. Lam, *et al.*, “The future of computing performance: Game over or next level?,” 2011.

[4] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, “Open MPI: Goals, concept, and design of a next generation MPI implementation,” in *Proceedings, 11th European PVM/MPI Users’ Group Meeting*, Budapest, Hungary, Sep. 2004, pp. 97–104.

[5] W. Gropp, “MPICH2: a new start for MPI implementations,” in *European Parallel Virtual Machine/Message Passing Interface Users’ Group Meeting*, Springer, 2002, pp. 7–7.

[6] D. K. Panda, H. Subramoni, C.-H. Chu, and M. Bayatpour, “The MVAPICH project: Transforming research into high-performance MPI library for HPC community,” *Journal of Computational Science*, p. 101 208, 2020.

[7] J. Pekkilä, “Astaroth: A library for stencil computations on graphics processing units,” Master’s thesis, Aalto University School of Science, Espoo, Finland, 2019.

[8] J. Pekkilä, M. S. Väisälä, M. J. Käpylä, M. Rheinhardt, and O. Lappi, 2021. arXiv: `2103.01597 [cs.DC]`.

[9] S. Williams, A. Waterman, and D. Patterson, “Roofline: An insightful visual performance model for multicore architectures,” *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, Apr. 2009, ISSN: 0001-0782.

[10] D. Fey, *Grid-Computing: Eine Basistechnologie für Computational Science*. Springer-Verlag, 2010.

[11] A. Schäfer and D. Fey, “Libgeodecomp: A grid-enabled library for geometric decomposition codes,” in *Proceedings of the 15th European PVM/MPI Users’ Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Dublin, Ireland: Springer-Verlag, 2008, pp. 285–294, ISBN: 978-3-540-87474-4.

[12] E. G. Coffman Jr. and J. L. Bruno, *Computer and Job-shop Scheduling Theory*. Wiley-Interscience, 1976.

[13] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, “Legion: Expressing locality and independence with logical regions,” in *SC Conference on High Performance Computing Networking, Storage and Analysis, SC ’12, Salt Lake City, UT, USA - November 11 - 15, 2012*, J. K. Hollingsworth, Ed., IEEE/ACM, 2012, p. 66. [Online]. Available: `https://doi.org/10.1109/SC.2012.71`.

[14] T. G. Mattson, R. Cledat, V. Cavé, V. Sarkar, Z. Budimlic, S. Chatterjee, J. B. Fryman, I. Ganev, R. Knauerhase, M. Lee, B. Meister, B. Nickerson, N. Pepperling, B. Seshasayee, S. Tasirlar, J. Teller, and N. Vrvilo, “The open community runtime: A runtime system for extreme scale computing,” in *2016 IEEE High Performance Extreme Computing Conference, HPEC 2016, Waltham, MA, USA, September 13-15, 2016*, IEEE, 2016, pp. 1–7. [Online]. Available: `https://doi.org/10.1109/HPEC.2016.7761580`.

[15] D. Pfander, G. Daiß, D. Marcello, H. Kaiser, and D. Pflüger, “Accelerating octo-tiger: Stellar mergers on intel knights landing with HPX,” in *Proceedings of the International Workshop on OpenCL, IWOCL 2018, Oxford, United Kingdom, May 14-16, 2018*, S. McIntosh-Smith and B. Bergen, Eds., ACM, 2018, 19:1–19:8. [Online]. Available: `https://doi.org/10.1145/3204919.3204938`.

[16] P. Thoman, K. Dichev, T. Heller, R. Iakymchuk, X. Aguilar, K. Hasanov, P. Gschwandtner, P. Lemarinier, S. Markidis, H. Jordan, T. Fahringer, K. Katrinis, E. Laure, and D. S. Nikolopoulos, “A taxonomy of task-based parallel programming technologies for high-performance computing,” *J. Supercomput.*, vol. 74, no. 4, pp. 1422–1434, 2018. [Online]. Available: `https://doi.org/10.1007/s11227-018-2238-4`.

[17] L. Landau and E. Lifshitz, *Theoretical Physics, vol. 6, Fluid Mechanics*. Pergamon Press, 1987.

[18] P. A. Davidson, *Turbulence: an introduction for scientists and engineers*. Oxford university press, 2015.

[19] F. Krause, "The cosmic dynamo: From $t = -\infty$ to cowling's theorem: A review on history," in *Symposium-International Astronomical Union*, Cambridge University Press, vol. 157, 1993, pp. 487–499.

[20] A. Brandenburg and K. Subramanian, "Astrophysical magnetic fields and nonlinear dynamo theory," *Physics Reports*, vol. 417, no. 1-4, pp. 1–209, 2005.

[21] E. N. Parker, "Hydromagnetic dynamo models.," *The Astrophysical Journal*, vol. 122, p. 293, 1955.

[22] A. Brandenburg, on behalf of the Pencil Code Collaboration, *Pencil Code*, version v2020.08.06, Zenodo, 2020.

[23] Nordic Institute for Theoretical Physics, *The Pencil Code: A high-order MPI code for MHD turbulence. user's and reference manual*, 2021.

[24] A. Brandenburg, "Computational aspects of astrophysical MHD and turbulence," *Advances in Nonlinear Dynamos*, The Fluid Mechanics of Astrophysics and Geophysics, vol. 9, no. 0, pp. 269–344, Apr. 2003.

[25] M. S. Väisälä, "Magnetic phenomena of the interstellar medium in theory and observation," PhD thesis, University of Helsinki, Helsinki, Finland, 2017. [Online]. Available: `http://urn.fi/URN:ISBN:978-951-51-2778-5`.

[26] P. A. Davidson, *An introduction to magnetohydrodynamics*, 2002.

[27] *Astaroth Repository*, ReSoLVE Center of Excellence, Espoo, Finland, 2020. [Online]. Available: `https://bitbucket.org/jpekkila/astaroth`.

[28] R. J. LeVeque, *Finite Difference Methods for Ordinary and Partial Differential Equations: Steady-State and Time-Dependent Problems*. Society for Industrial and Applied Mathematics, 2007.

[29] B. Fornberg, "Classroom note: Calculation of weights in finite difference formulas," *SIAM Rev.*, vol. 40, no. 3, pp. 685–691, 1998.

[30] M. Abramowitz and I. Stegun, *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*, ser. Applied Mathematics Series. US National Bureau of Standards). New York, NY, USA: Dover, 1968.

[31] C. Runge, "Ueber die numerische auflösung von differentialgleichungen," *Mathematische Annalen*, no. 46, pp. 167–178, 1895.

[32] W. Kutta, "Beitrag zur naherungsweisen integration von differentialgleichungen," *Zeitschrift für Mathematik und Physik*, no. 46, pp. 435–453, 1901.

[33] J. Williamson, "Low-storage Runge-Kutta schemes," *Journal of Computational Physics*, vol. 35, no. 1, pp. 48–56, Mar. 1980, ISSN: 0021-9991.

[34] D. A. Padua, Ed., *Encyclopedia of Parallel Computing*. Springer, 2011, ISBN: 978-0-387-09765-7.

[35] R. J. LeVeque, D. Mihalas, E. A. Dorfi, and E. Müller, *Computational Methods for Astrophysical Fluid Flow*. Springer Science & Business Media, 1998, vol. 27.

[36] CSC – IT Center for Science Ltd. (2021). Puhti service description, [Online]. Available: `https://a3s.fi/puhti-service-documents/Puhti-servicedescription.pdf` (visited on 10/03/2021).

[37] The MPI Forum, "MPI: a message passing interface," in *Proceedings Supercomputing '93, Portland, Oregon, USA, November 15-19, 1993*, ACM, 1993, pp. 878–883. [Online]. Available: `https://doi.org/10.1145/169627.169855`.

[38] W. D. Gropp, E. L. Lusk, and A. Skjellum, *Using MPI - Portable Parallel Programming with the Message-Passing Interface, 3rd Edition*, ser. Scientific and engineering computation. MIT Press, 2014, ISBN: 978-0-262-52739-2.

[39] *The MPI Forum*. [Online]. Available: `https://www.mpi-forum.org`.

[40] The MPI Forum, *Message Passing Interface Standard 3.1*. 2015.

[41] W. Gropp, T. Hoefler, R. Thakur, and E. Lusk, *Using advanced MPI: Modern features of the message-passing interface*, ser. Scientific and engineering computation. MIT Press, 2014.

[42] R. Brightwell and K. D. Underwood, "An analysis of NIC resource usage for offloading MPI," in *18th International Parallel and Distributed Processing Symposium (IPDPS 2004), CD-ROM / Abstracts Proceedings, 26-30 April 2004, Santa Fe, New Mexico, USA*, IEEE Computer Society, 2004.

[43] R. Keller and R. L. Graham, "Characteristics of the unexpected message queue of MPI applications," in *Recent Advances in the Message Passing Interface - 17th European MPI Users' Group Meeting, EuroMPI 2010, Stuttgart, Germany, September 12-15, 2010. Proceedings*, R. Keller, E. Gabriel, M. M. Resch, and J. J. Dongarra, Eds., ser. Lecture Notes in Computer Science, vol. 6305, Springer, 2010, pp. 179–188.

[44] Y. Wang, G. Stocks, W. Shelton, D. Nicholson, Z. Szotek, and W. Temmerman, "Order-n multiple scattering approach to electronic structure calculations," *Physical review letters*, vol. 75, pp. 2867–2870, Nov. 1995.

[45] S. Potluri, K. Hamidouche, A. Venkatesh, D. Bureddy, and D. K. Panda, "Efficient inter-node MPI communication using GPUDirect RDMA for InfiniBand clusters with NVIDIA GPUs," in *42nd International Conference on Parallel Processing, ICPP 2013, Lyon, France, October 1-4, 2013*, IEEE Computer Society, 2013, pp. 80–89.

[46] S. Potluri, H. Wang, D. Bureddy, A. K. Singh, C. Rosales, and D. K. Panda, "Optimizing MPI communication on multi-GPU systems using CUDA inter-process communication," in *26th IEEE International Parallel and Distributed Processing Symposium Workshops & PhD Forum, IPDPS 2012, Shanghai, China, May 21-25, 2012*, IEEE Computer Society, 2012, pp. 1848–1857.

[47] R. Rosen, *Linux kernel networking: Implementation and theory*. Apress, 2014.

[48] D. Viswanath, *Scientific Programming and Computer Architecture*. The MIT Press, 2017. [Online]. Available: `https://divakarvi.github.io/bk-spca/spca.html`.

[49] R. Shi, S. Potluri, K. Hamidouche, J. L. Perkins, M. Li, D. Rossetti, and D. K. Panda, "Designing efficient small message transfer mechanism for inter-node MPI communication on infiniband GPU clusters," in *21st International Conference on High Performance Computing, HiPC 2014, Goa, India, December 17-20, 2014*, IEEE Computer Society, 2014, pp. 1–10.

[50] P. Shamis, M. G. Venkata, M. G. Lopez, M. B. Baker, O. R. Hernandez, Y. Itigin, M. Dubman, G. Shainer, R. L. Graham, L. Liss, Y. Shahar, S. Potluri, D. Rossetti, D. Becker, D. Poole, C. Lamb, S. Kumar, C. B. Stunkel, G. Bosilca, and A. Bouteiller, "UCX: an open source framework for HPC network APIs and beyond," in *23rd IEEE Annual Symposium on High-Performance Interconnects, HOTI 2015, Santa Clara, CA, USA, August 26-28, 2015*, IEEE Computer Society, 2015, pp. 40–43.

[51] Open Unified Communication X, *UCX github repository wiki*. [Online]. Available: `https://github.com/openucx/ucx/wiki` (visited on 11/03/2021).

[52] J. D. Little, "A proof for the queuing formula: $L = \lambda W$," *Operations research*, vol. 9, no. 3, pp. 383–387, 1961.

[53] R. A. Fisher and L. H. C. Tippett, "Limiting forms of the frequency distribution of the largest or smallest member of a sample," in *Mathematical Proceedings of the Cambridge Philosophical Society*, Cambridge University Press, vol. 24, 1928, pp. 180–190.

[54] H. Kaiser, P. Diehl, A. S. Lemoine, B. Lelbach, P. Amini, A. Berge, J. Biddiscombe, S. R. Brandt, N. Gupta, T. Heller, K. A. Huck, Z. Khatami, A. Kheirkhahan, A. Reverdell, S. Shirzad, M. Simberg, B. Wagle, W. Wei, and T. Zhang, "HPX - the C++ standard library for parallelism and concurrency," *J. Open Source Softw.*, vol. 5, no. 53, p. 2352, 2020. [Online]. Available: `https://doi.org/10.21105/joss.02352`.

[55] S. J. Pennycook, J. D. Sewall, and V. W. Lee, "Implications of a metric for performance portability," *Future Gener. Comput. Syst.*, vol. 92, pp. 947–958, 2019. [Online]. Available: `https://doi.org/10.1016/j.future.2017.08.007`.

[56] K. S. Hong and J. Y. Leung, "On-line scheduling of real-time tasks," in *Proceedings of the 9th IEEE Real-Time Systems Symposium (RTSS '88), December 6-8, 1988, Huntsville, Alabama, USA*, IEEE Computer Society, 1988, pp. 244–250.

[57] J. Kurzak, H. Ltaief, J. J. Dongarra, and R. M. Badia, "Scheduling dense linear algebra operations on multicore processors," *Concurr. Comput. Pract. Exp.*, vol. 22, no. 1, pp. 15–44, 2010. [Online]. Available: `https://doi.org/10.1002/cpe.1467`.

[58] M. Schaller, P. Gonnet, A. B. G. Chalk, and P. W. Draper, "SWIFT: using task-based parallelism, fully asynchronous communication, and graph partition-based domain decomposition for strong scaling on more than 100, 000 cores," *CoRR*, vol. abs/1606.02738, 2016. arXiv: `1606.02738`. [Online]. Available: `http://arxiv.org/abs/1606.02738`.

[59] R. L. Graham, "Bounds for certain multiprocessing anomalies," *Bell system technical journal*, vol. 45, no. 9, pp. 1563–1581, 1966.

[60] CSC – IT Center for Science Ltd. (2020). One of the world's mightiest supercomputers, lumi, will lift european research and competitiveness to a new level and promotes green transition, [Online]. Available: `https://www.lumi-supercomputer.eu/lumi-one-of-the-worlds-mightiest-supercomputers/` (visited on 22/03/2021).

Appendix A: A simple method for deriving central difference coefficients

In this appendix I demonstrate a method using Richardson extrapolation to derive the coefficients for a five-point central difference approximation.

To get a 4th-order approximation of the first derivative, we define $f'(x)$ using a series expansion of central difference quotients with two step-sizes, h and $2h$. This produces a system of equations:

$$f'(x) = \begin{cases} D(f,h) & = \frac{f(x+h)-f(x-h)}{2h} + ah^2 + \mathcal{O}(h^4) \\ D(f,2h) & = \frac{f(x+2h)-f(x-2h)}{4h} + 4ah^2 + \mathcal{O}(h^4) \end{cases}$$

where $a = -\frac{f^{(3)}(x)}{3!}$.

We can then solve a very simple linear equation and get the two $\mathcal{O}(h^2)$ terms to cancel out and obtain a difference quotient expression for the derivative:

$$f'(x) = \frac{4}{3}\left(D(f,h) - \frac{D(f,2h)}{4}\right) = \frac{4}{3}\begin{pmatrix} \dfrac{f(x+h)-f(x-h)}{2h} & +\cancel{ah^2} + \mathcal{O}(h^4) \\ -\dfrac{f(x+2h)-f(x-2h)}{16h} & -\cancel{ah^2} + \mathcal{O}(h^4) \end{pmatrix}$$

$$= \frac{f(x-2h) - 8f(x-h) + 8f(x+h) - f(x+2h)}{12h} + \mathcal{O}(h^4)$$

General method

Higher-order approximations can be obtained using the same process. In order to keep stencils manageable and applicable to regular grids we take unit steps of h to sample new points. This increases the stencil radius r by 1 for every improvement in approximation. We can then obtain a generic truncation error $\mathcal{O}(h^{2r})$ by solving for the coefficients c_i in a simple system of linear equations:

$$\begin{cases} \sum_{i=1}^{r-1} i^2 c_i = 0 \\ \sum_{i=1}^{r-1} i^4 c_i = 0 \\ \dots \\ \sum_{i=1}^{r-1} i^{2(r-1)} c_i = 0 \end{cases}$$

The approximation can then be obtained by:

$$f'(x) = \sum_{i=1}^{r} c_i \left(\sum_{k=1}^{r} c_k D(f, kh) \right) + \mathcal{O}(h^{2r})$$

Appendix B: Proof that buffer dependencies are satisfied by task dependencies

This proof assumes symmetric stencils and the partitioning of work into tasks according to the scheme presented in section 4.6. Recall the lemmas from section 4.6:

Lemma 1. $\forall t, \tau : T_t$ *depends on* $T'_{t+c} \iff T_\tau$ *depends on* $T'_{\tau+c}$.

Lemma 2. $\mathbf{C}_t[\boldsymbol{v}]$ *depends on* $\mathbf{B}_t[\boldsymbol{u}] \iff \mathbf{B}_t[\boldsymbol{u}]$ *depends on* $\mathbf{C}_{t-1}[\boldsymbol{v}]$.

Lemma 3. $\mathbf{C}_t[\boldsymbol{v}]$ *depends on* $\mathbf{C}_{t-1}[\boldsymbol{u}] \iff \mathbf{C}_t[\boldsymbol{u}]$ *depends on* $\mathbf{C}_{t-1}[\boldsymbol{v}]$.

Remembering that halo exchange tasks do not need *input data* from the halo, there are three possible cases where a task would disturb another: (1) a halo exchange task disturbs a compute task, (2) a compute task disturbs a halo exchange task, and (3) a compute task disturbs a compute task.

The three cases are quite similar and the proof for all three begins the same way, which I will write down as another lemma:

Lemma 4. *If* T_k *disturbs another task* T'_j *by overwriting its inputs, then* T'_j *depends on* T_i, $i \leq j \leq k \wedge i < k$.

Proof. Assume task T_k overwrites a region of data in a buffer, which disturbs a task T'_j by overwriting its inputs in the buffer, $j \leq k$.

If the task T_j is disturbed by T_k writing to the buffer, then T_j depends on the data in the output region of T_k in the buffer.

Because task output regions are disjoint and inputs and outputs are swapped between iterations, the task that is responsible for writing the data to the buffer is the same task T at an earlier iteration: T_i s.t. $i < k \wedge i \leq j$.

Thus, T'_j depends on T_i. □

Disturbances between halo exchange tasks and compute tasks can be lumped into one case. I prove below that halo exchange tasks do not disturb compute tasks, but the proof applies the other way round as well. The proof relies on lemma 2 and is not valid when the assumptions underlying the lemma are not true (e.g., symmetric stencils).

Theorem 1. *In an ISL with symmetric stencils, output from a halo exchange task* $\mathbf{B}_k[\boldsymbol{u}]$ *cannot overwrite the inputs to a compute task* $\mathbf{C}_j[\boldsymbol{v}]$*(and vv.)*

Proof. Assume that $\mathbf{B}_k[\boldsymbol{u}]$ disturbs a compute task $\mathbf{C}_j[\boldsymbol{v}]$, $j \le k$.

Due to lemma 4, $\mathbf{C}_j[\boldsymbol{v}]$ depends on $\mathbf{B}_i[\boldsymbol{u}]$.

Due to lemmas 2 and 1, $\mathbf{B}_k[\boldsymbol{u}]$ depends on all tasks $\mathbf{C}_i[\boldsymbol{v}],,...,\mathbf{C}_k[\boldsymbol{v}]$ either directly or as part of a chain of dependencies.

Since $\mathbf{C}_i[\boldsymbol{v}]$ is a prerequisite task of $\mathbf{B}_k[\boldsymbol{u}]$, $\mathbf{B}_k[\boldsymbol{u}]$ cannot disturb $\mathbf{C}_i[\boldsymbol{v}]$ by overwriting its inputs. □

The proof of compute tasks not disturbing other compute tasks is similar. The proof relies on lemma 3 and is not valid when the assumptions underlying the lemma are not true (e.g symmetric stencils).

Theorem 2. *In an ISL with symmetric stencils, output from one compute task* $\mathbf{C}_k[\boldsymbol{u}]$ *cannot overwrite the inputs to another compute task* $\mathbf{C}_j[\boldsymbol{v}]$

Proof. Assume that $\mathbf{C}_k[\boldsymbol{u}]$ disturbs a compute task $\mathbf{C}_j[\boldsymbol{v}]$, $k < j$.

Due to lemma 4, $\mathbf{C}_j[\boldsymbol{v}]$ depends on $\mathbf{C}_i[\boldsymbol{u}]$, $i < j$. Because all compute-compute dependencies are from one iteration to the next, we know that $i = j - 1$. Therefore, $\mathbf{C}_j[\boldsymbol{v}]$ depends on $\mathbf{C}_{j-1}[\boldsymbol{u}]$

Due to lemmas 3 and 1, $\mathbf{C}_k[\boldsymbol{u}]$ depends on all tasks $\mathbf{C}_{k-1}[\boldsymbol{v}],...,\mathbf{C}_j[\boldsymbol{v}]$ either directly or as part of a chain of dependencies.

Since $\mathbf{C}_j[\boldsymbol{v}]$ is a prerequisite task of $\mathbf{C}_k[\boldsymbol{u}]$, $\mathbf{C}_k[\boldsymbol{u}]$ cannot disturb $\mathbf{C}_j[\boldsymbol{v}]$ by overwriting its inputs. □

Glossary

Physics glossary

Adiabatic
Occurring without loss or gain of heat.

Advection
The transport of a quantity in a fluid by the bulk motion of the fluid.

Convection
The transport of a quantity in a fluid by the overall effects of forces and the gradients of the fluid's material properties (e.g. density).

Electromagnetics
A field of physics dedicated to the electromagnetic force.

Laminar flow
An orderly flow regime, the fluid flows in layers with little mixing.

Massive body
A body that has mass.

Mechanics
A field of physics dedicated to the interactions between massive bodies and forces applied on them.

Multiphysics
A simulation paradigm where multiple physics models are applied to the same simulation data.

Newtonian fluid
A fluid in which any viscous stress is proportional to the rate of strain due to flow.

Poloidal
The direction on the surface of a torus which goes through the hole in the torus.

Thermodynamics
A field of physics dedicated to the relationships between heat, work, and energy.

Toroidal
The direction on the surface of a torus along circles centered on the hole in the torus.

Turbulent flow
A disorderly flow regime, the fluid mixes violently and forms swirling vortices of excess kinetic energy.

Viscosity
A property of a fluid. It characterizes the flow-resistance of a fluid. Higher viscosity $\rightarrow$ lower flow rates; lower viscosity $\rightarrow$ higher flow rates.

Computer glossary

API Application programming interface, the interface through which one software component interacts with another.

Asynchronous
Execution that is not synchronous. Events serve to coordinate execution instead of top-down control.

CPU
Central processing unit, a computer component that executes computer programs. Most CPUs today consist of multiple processor cores on a single chip.

CPU core, processor core
A single embedded processing unit on a multi-core CPU die. A single core is a serial processing element, multiple cores are needed for parallel execution.

DSL Domain specific language, a specialized language purpose-built for domain specialists.

GPU
Graphics processing unit, a computer component that is used to draw graphics on a screen. Due to their parallel-processing capabilities (they have many cores), GPUs are well suited as accelerators for heavy computations.

Interconnect
Short for interconnection network or interconnection network system. A tight-knit

Fork-join

A model of execution where many execution threads are launched, which do work and then wait for each other to reach a synchronization point.

PCIe

PCI Express. A standard hardware interface and motherboard interconnect.

Runtime

1. Short for runtime system. A software component that manages the execution of a program.
2. The time it takes to execute a program.

Schedule

The order in which units of work are executed on a computer.

Scheduler

A software component that produces a schedule.

SIMD

Single instruction, multiple data. Multiple processing elements execute the same instruction but the inputs to each are different.

Synchronous execution

Execution of operations in unison. computer communication network that connects processors to resources. E.g., an Infiniband interconnect that connects nodes in an HPC cluster to each other or a PCIe interconnect that connects resources on a motherboard to each other.