UNIVERSITY OF NOVI SAD
FACULTY OF SCIENCES
DEPARTMENT OF
MATHEMATICS AND INFORMATICS

Vladimir Lončar

Hybrid parallel algorithms for solving nonlinear Schrödinger equation

— PhD thesis —

Advisors:
Dr. Antun Balaž
Dr. Srđan Škrbić

Novi Sad, 2017

Contents

Acknowledgements

First and foremost I offer my sincerest gratitude to my advisors, Dr. Antun Balaž and Dr. Srđan Škrbić, for supporting me during these past six years. Antun has set an example of excellence as a researcher, mentor, instructor, and role model. His patience, guidance, encouragement and advice have made my PhD experience more productive and stimulating, and I thank him for the great effort he put into training me in the scientific field. Srđan has been supportive throughout my thesis and has given me the freedom to pursue my goals without objection. I especially thank him for his support during my transition to Scientific Computing Laboratory.

I also have to thank the members of my PhD committee, Professors Dragan Mašulović, Nataša Krejić and Miljko Satarić for their helpful advice and invaluable feedback.

I would like to give special thanks to my colleagues from the Scientific Computing Laboratory, for all productive discussions we had, for all the laughs we shared over lunch, and for the great time I have spent with you.

This thesis is written in the Scientific Computing Laboratory, Center for the Study of Complex Systems of the Institute of Physics Belgrade and is supported by the Ministry of Education, Science, and Technological Development of the Republic of Serbia under project ON171017. Numerical simulations were run on the PARADOX supercomputing facility at the Scientific Computing Laboratory of the Institute of Physics Belgrade.

Lastly, I would like to thank my family and friends for all their love and encouragement. For my parents who raised me with a love of science and supported me in all my pursuits. For my friends for providing support and friendship that I needed. Thank you.

Abstract

Numerical methods and algorithms for solving of partial differential equations, especially parallel algorithms, are an important research topic, given the very broad applicability range in all areas of science. Rapid advances of computer technology open up new possibilities for development of faster algorithms and numerical simulations of higher resolution. This is achieved through parallelization at different levels that practically all current computers support.

In this thesis we develop parallel algorithms for solving one kind of partial differential equations known as nonlinear Schrödinger equation (NLSE) with a convolution integral kernel. Equations of this type arise in many fields of physics such as nonlinear optics, plasma physics and physics of ultracold atoms, as well as economics and quantitative finance. We focus on a special type of NLSE, the dipolar Gross-Pitaevskii equation (GPE), which characterizes the behavior of ultracold atoms in the state of Bose-Einstein condensation.

We present novel parallel algorithms for numerically solving GPE for a wide range of modern parallel computing platforms, from shared memory systems and dedicated hardware accelerators in the form of graphics processing units (GPUs), to heterogeneous computer clusters. For shared memory systems, we provide an algorithm and implementation targeting multi-core processors using OpenMP. We also extend the algorithm to GPUs using CUDA toolkit and combine the OpenMP and CUDA approaches into a hybrid, heterogeneous algorithm that is capable of utilizing all available resources on a single computer.

Given the inherent memory limitation a single computer has, we develop a distributed memory algorithm based on Message Passing Interface (MPI) and previous shared memory approaches. To maximize the performance of hybrid implementations, we optimize the parameters governing the distribution of data and workload using a genetic algorithm. Visualization of the increased volume of output data, enabled by the efficiency of newly developed algorithms, represents a challenge in itself. To address this, we integrate the implementations with the state-of-the-art visualization tool (VisIt), and use it to study two use-cases which demonstrate how the developed programs can be applied to simulate real-world systems.

Sažetak

Numerički metodi i algoritmi za rešavanje parcijalnih diferencijalnih jednačina, naročito paralelni algoritmi, predstavljaju izuzetno značajnu oblast istraživanja, uzimajući u obzir veoma široku primenljivost u svim oblastima nauke. Veliki napredak informacione tehnologije otvara nove mogućnosti za razvoj bržih algoritama i numeričkih simulacija visoke rezolucije. Ovo se ostvaruje kroz paralelizaciju na različitim nivoima koju poseduju praktično svi moderni računari.

U ovoj tezi razvijeni su paralelni algoritmi za rešavanje jedne vrste parcijalnih diferencijalnih jednačina poznate kao nelinearna Šredingerova jednačina sa integralnim konvolucionim kernelom. Jednačine ovog tipa se javljaju u raznim oblastima fizike poput nelinearne optike, fizike plazme i fizike ultrahladnih atoma, kao i u ekonomiji i kvantitativnim finansijama. Teza se bavi posebnim oblikom nelinearne Šredingerove jednačine, Gros-Pitaevski jednačinom sa dipol-dipol interakcionim članom, koja karakteriše ponašanje ultrahladnih atoma u stanju Boze-Ajnštajn kondenzacije.

U tezi su predstavljeni novi paralelni algoritmi za numeričko rešavanje Gros-Pitaevski jednačine za širok spektar modernih računarskih platformi, od sistema sa deljenom memorijom i specijalizovanih hardverskih akceleratora u obliku grafičkih procesora, do heterogenih računarskih klastera. Za sisteme sa deljenom memorijom, razvijen je algoritam i implementacija namenjena višejezgarnim centralnim procesorima korišćenjem OpenMP tehnologije. Ovaj algoritam je proširen tako da radi i u okruženju grafičkih procesora korišćenjem CUDA alata, a takođe je razvijen i predstavljen hibridni, heterogeni algoritam koji kombinuje OpenMP i CUDA pristupe i koji je u stanju da iskoristi sve raspoložive resurse jednog računara.

Imajući u vidu inherentna ograničenja raspoložive memorije koju pojedinačan računar poseduje, razvijen je i algoritam za sisteme sa distribuiranom memorijom zasnovan na Message Passing Interface tehnologiji i prethodnim algoritmima za sisteme sa deljenom memorijom. Da bi se maksimalizovale performanse razvijenih hibridnih implementacija, parametri koji određuju raspodelu podataka i računskog opterećenja su optimizovani korišćenjem genetskog algoritma. Poseban izazov je vizualizacija povećane količine izlaznih podataka, koji nastaju kao rezultat efikasnosti novorazvijenih algoritama. Ovo je u tezi rešeno kroz integraciju implementacija sa najsavremenijim alatom za vizualizaciju (VisIt), što je omogućilo proučavanje dva primera koji pokazuju kako razvijeni programi mogu da se iskoriste za simulacije realnih sistema.

Chapter 1

Introduction

The study of dynamical evolution of various physical systems is essential for their understanding and represents a key ingredient for possible applications. It is still debated how deterministic nature is, but the majority of models we use to describe systems in nature are deterministic. This means that we use first and second order differential equations to model their dynamic behavior. Even quantum systems, although notorious for their probabilistic behavior, are described in the deterministic mathematical framework and their evolution can be cast into the form of second order differential equations. Apart from very simple models, systems with multiple degrees of freedom are usually described by partial differential equations. At the most fundamental level, these equations are linear and the superposition principle applies. However, due to large numbers of degrees of freedom for realistic systems, the complexity of the corresponding systems of equations is prohibitively large to address either analytically or numerically. Therefore, the effective models with much smaller dimensionality are often used, albeit not without introducing additional complexities, such as nonlinearity of the system. This is true for many-body physics problems, especially physics of ultracold atoms, plasma physics and many others. There are however, systems where nonlinearity is inherent, such as nonlinear optics. Whatever the case may be, nonlinearity is always a source of interesting and novel phenomena and has to be carefully studied. As usual, this translates to mathematical complexity in solving of the corresponding nonlinear partial differential equations. Therefore, the development of mathematical methods and computer algorithms for solving of such kind of equations is of very broad interest, not only in physics, but also in applied physics and engineering, astrophysics, chemistry, quantitative finance, etc.

As far as analytical methods are concerned, our options are limited to a small number of exactly solvable models or application of approximative techniques. While the number of such techniques is considerable, many of them have the disadvantage of lacking the proper way to estimate errors associated with the obtained results. Even for methods where such an estimate is possible, only the lowest order calculations can be done within reason, and calculation of higher order corrections usually requires an exponentially growing number of symbolic operations, making the approach intractable. This is the reason why numerical simulations are so widely used today, and they represent the main tool for studying large number of systems in nature, as well as social systems.

Numerical simulations of realistic systems are often computationally very intensive, substantially more so in higher number of dimensions. Therefore, they cannot be performed efficiently on a single, serial computing resource, and instead require parallel processing. Parallel algorithms for solving partial differential equations are an important research topic today, given the fact that practically all current computers support some form of multiprocessing. The development of parallel algorithms for solving a particular type of partial differential equations, nonlinear Schrödinger equation (NLSE) type, represents the main topic of this thesis.

Multi-core central processing units (CPU) form the first level of parallelism available in all modern computers. Individual cores of the CPU may perform tasks in parallel while sharing a single view of data in memory. By extension, several multi-core CPUs may be packaged within a single computer and given access to the shared memory to further increase a system's parallel processing capabilities. Developing parallel algorithms for shared memory systems is relatively easy, however maximizing the efficiency of their implementation is often hindered by the fact that many systems do not provide uniform memory access times, instead relying on non-uniform memory access (NUMA) design. Also, data coherence issues arise when caching is employed within separate cores. Numerous hardware and software technologies exist to enable better utilization and simpler control of shared memory systems, with OpenMP emerging as the most widespread application programming interface for such computing platforms.

Next level of parallelism in this hierarchy is provided by the dedicated hardware accelerators, such as graphics processing units (GPU) and early Intel Xeon Phi coprocessors. These accelerators feature a vastly different hardware architecture to the one found in CPUs, often use their own separate main memory, and come packaged as an add-on for the existing computer. The difference in architecture is reflected in the programming model, whose successful utilization usually requires a significant modification of the underlying shared memory algorithm. In this thesis, we focus on a specific type of accelerator, the Nvidia GPU, which is programmed using CUDA toolkit. Combining parallelism of the CPU with the one found in the GPU to take full advantage of the resources one computer has is challenging. While CPU and GPU have separate memories, they cannot easily be programmed as a distributed memory system, and can instead be considered a heterogeneous computing resource, which requires development of hybrid algorithms to be fully exploited. These algorithms must shuffle data between CPU and GPU efficiently, overlap computation, and ensure correctness of concurrent operations through synchronization.

Combining several computers into a computer cluster forms the final level of parallelism. Even with the rate at which current CPU and GPU technologies advance, a single computer may not have enough memory, processing power, or both, to address the computational requirements of numerical simulations of high resolution. Therefore, computer clusters, comprised of large number of individual computer nodes interconnected via high-speed networking infrastructure, and accompanied by software solutions that allow for the whole system to be used in a parallel manner (e.g., Message Passing Interface, MPI) are created to fill this gap and complete the hierarchy. These systems have distributed memory, meaning that computational tasks on each node of the cluster operate only on data that are local to that node, and access to remote data is obtained through communication with one or more remote computational tasks. Today, each individual cluster node itself represents a shared memory system and the algorithm may or may not exploit this to attain finer level of parallelism granulation. Distributed memory algorithms must take all this into account and provide a data distribution scheme that facilitates efficient communication between computing nodes and maximizes the computation on local data.

Efficient use of such diverse computing systems requires domain-specific knowledge and the development of parallel algorithms that take into consideration the specifics of the hardware the system contains, as well as parallelization of numerical methods used to solve the given problem. We explore all levels of parallelism across the entire hierarchy of modern hardware resources in this thesis.

As mentioned above, parallel algorithms for solving NLSE are of great importance in the physics of ultracold atoms, which studies the properties of matter at very low temperatures, at the nanokelvin level. At these temperatures, rarified gases of alkali metal elements exhibit a phase transition known as Bose-Einstein condensation [1, 2, 3, 4, 5], which produces a new state of quantum matter, characterized by a coherent, collective behavior. Bose-Einstein condensation is a purely quantum effect and does not have a classical counterpart. It is one of macroscopic quantum phenomena, together with superfluidity, superconductivity and lasers. Bose-Einstein condensates (BEC) and ultracold atoms are one of hot research topics because they provide previously unimaginable control over parameters, properties and behavior patterns of the system. In particular, short-range contact interaction strength in such systems can be tuned over many orders of magnitude, and can even switch the sign, i.e., can be made either attractive or repulsive. Furthermore, the dimensionality of the system can also be tuned using the external trapping potential, making possible crossovers from three spatial dimensions (3D) to two (2D) or one (1D). This can be even performed dynamically, by changing the shape of the trapping potential from a spherical one to disc-shaped or cigar-shaped one. BECs have possible applications in quantum computing [6, 7], as well as in quantum simulations of other systems. As of today, they are considered the only feasible Feynman simulators [8].

Despite this versatility, contact interactions in BEC systems, realized through s-wave scattering of atoms or molecules, are extremely short-ranged and usually have very limited effect on the properties of the system. Therefore, in the last decade special attention was devoted to the study of systems in which, alongside the omnipresent contact interactions, atoms or molecules also exhibit long-range dipole-dipole interaction. The dipolar BEC was first realized in 2005 [9] with chromium atoms (^{52}Cr), which possess a small permanent magnetic dipolar moment. Since then, new atomic and molecular species with larger permanent or induced magnetic (^{168}Er, ^{164}Dy) or electric (^{39}K^{87}Rb, ^{87}Rb^{133}Cs, ^{7}Li^{133}Cs) dipolar moments are broadly investigated, and some of them are successfully condensed. Heteronuclear molecular species with large permanent electric dipole moments, which should become experimentally accessible in the next few years, will have orders of magnitude larger dipole-dipole interactions and will enable full exploration of the strong interaction regime. The presence of long-range interactions can lead to interesting new phenomena, such as phases with nontrivial order (ferromagnetic, antiferromagnetic, striped, etc.), and therefore this field is a topic of intensive research efforts by many groups worldwide.

This motivates us to study numerical algorithms for solving of nonlinear partial differential equations used for effective description of dipolar BEC systems. On one hand, it is an interesting problem in physics in itself, with many possible applications and expected new breakthroughs in the near future. On the other hand, it is also a challenging computer science problem that requires development of novel algorithms, parallelization and the use of latest computer technology.

To begin the development of a (parallel) computer algorithm, we start from the mathematical formulation of the problem, which at its core is a many-body physics problem, involving a Hamiltonian with quantum fields, i.e., the formalism of the second quantization. BEC systems are usually realized with dilute atomic or molecular gases and therefore, in first-order approximation, we can neglect quantum fluctuations. Since the temperature is of the order of nanokelvin, we can neglect thermal excitations as well, which leads to the well-known mean-field theory [10, 11], with the field operators replaced by the classical fields, i.e., the single-particle wave function of the system. This effective wave function $\Psi(\mathbf{r};t)$ depends on position $\mathbf{r}=(x,y,z)$ and time t, and satisfies mean-field *Gross-Pitaevskii equation* (GPE), which for the contact interaction case reads

$$i\hbar\frac{\partial}{\partial t}\Psi(\mathbf{r};t)=\left[-\frac{\hbar^2}{2m}\nabla^2+V(\mathbf{r})+gN_{\mathrm{at}}|\Psi(\mathbf{r};t)|^2\right]\Psi(\mathbf{r};t)\,, \tag{1.1}$$

where m is the mass of the atomic or molecular species, $\hbar$ is the reduced Planck constant, N_{at} is the number of particles in the condensate and $V(\mathbf{r})$ is the trapping potential that confines the condensate, usually a harmonic function. As already mentioned, Eq. (1.1) includes only contact interaction, represented by a nonlinear term with the interaction strength g determined by the s-wave scattering length. Without the nonlinear term, Eq. (1.1) represents the usual Schrödinger equation, where the first term on the right-hand side is the kinetic energy, and $V(\mathbf{r})$ is the potential. The cubic term that arises from atomic interactions makes the equation nonlinear, and therefore it is usually called nonlinear Schrödinger equation. It is worth noting that similar equations also arise in other fields of physics and engineering, in different contexts. The stability and existence of solutions to GPE strongly depends on the sign of nonlinearity g. Namely, if g is positive, which corresponds to repulsive interactions, the system is stable and the above equation always has a physically meaningful solution. On the other hand, if g is negative, the system is stable only up to a critical number of particles and beyond this number it collapses.

For atomic and molecular species with dipolar moments we have to take into account the dipolar interaction term, which yields the dipolar GPE in the form

$$i\hbar\frac{\partial}{\partial t}\Psi(\mathbf{r};t)=\left[-\frac{\hbar^2}{2m}\nabla^2+V(\mathbf{r},t)+gN_{\mathrm{at}}|\Psi(\mathbf{r};t)|^2+N_{\mathrm{at}}\int U_{\mathrm{dd}}(\mathbf{r}-\mathbf{r}')|\Psi(\mathbf{r}';t)|^2d\mathbf{r}'\right]\Psi(\mathbf{r};t)\,. \tag{1.2}$$

The dipolar interaction term is given by a convolution integral that contains the position-dependent dipolar potential $U_{\mathrm{dd}}(\mathbf{r}-\mathbf{r}')$. While contact interaction is fixed for a given system, the dipolar interaction strongly depends on the relative orientation and distance of the dipoles, and can be both attractive and repulsive. This greatly affects stability of the system, similarly to the case of pure contact interaction: if the interaction is predominantly attractive, the system is unstable beyond a critical number of particles, while for predominantly repulsive interaction the system is unconditionally stable.

The dynamics of dipolar BEC system is described using the time-dependent GPE presented above. Typically, we start from a given state $\Psi(\mathbf{r},t=0)$, and propagate it in time to obtain the state of the system at some future time t. However, if the system is in the stationary, ground state, its time propagation reduces to just phase rotation, so that it can be written as $\Psi_0(\mathbf{r})e^{-i\mu t/\hbar}$, where $\Psi_0(\mathbf{r})$ is a real-valued function which satisfies the time-independent GPE

$$\mu\Psi_0(\mathbf{r})=\left[-\frac{\hbar^2}{2m}\nabla^2+V(\mathbf{r})+gN_{\mathrm{at}}\Psi_0(\mathbf{r})^2+N_{\mathrm{at}}\int U_{\mathrm{dd}}(\mathbf{r}-\mathbf{r}')|\Psi_0(\mathbf{r}')|^2d\mathbf{r}'\right]\Psi_0(\mathbf{r})\,, \tag{1.3}$$

where μ is the chemical potential of the system which corresponds to the energy of the system in that state. By employing the Wick rotation, the solution of time-independent equation can be obtained formally by turning to the imaginary time, starting from an arbitrary initial state and propagating the wave function until the convergence is reached. Therefore, the development of methods and algorithms for solving of GPE in both real and imaginary time is of practical interest and enables us to fully study the behavior and properties of BEC systems, from calculation of stationary states to dynamical evolution of a given state subject to a particular experimental protocol.

Numerical methods for solving partial differential equations require significant computing resources, in particular in presence of nonlinearities. The nonlinearities such as those in dipolar GPE further increase the complexity of the corresponding algorithms, insofar as BEC systems are often of multi-scale nature. For example, if the system dynamically develops features at different, independent length scales, such as vortices, density waves or collective oscillations, the smallest scale dictates the discretization spacing. Thus, it may turn out that the minimal mesh size, which is necessary for accurate modeling of the system, is too large to be handled by a single processing element. The necessity of employing fine-grained discretization schemes is so prevalent in many areas where equations of the NLSE type are used that the development of parallel algorithms and their implementation represents an important challenge.

Therefore it is not surprising that this challenge was already addressed, and that there are several approaches developed and tailored specifically for solving of GPE. Most of them are inherently serial and focus solely on the equation for the contact interaction only. Algorithms and software implementations for the dipolar GPE are practically missing, in particular parallel algorithms do not exist in public domain or in the literature. This thesis addresses the identified development gap and provides a significant step forward in the area of numerical methods and parallel algorithms for solving equations the NLSE type, of interest for the fields of ultracold atoms, nonlinear optics, and other areas.

Research results presented in this thesis contribute to the fields of scientific computing and parallel programming, as well as complexity modeling and optimization of hybrid algorithms. In particular, the main contribution of this thesis is the development of parallel algorithms for solving of GPE and related equations of the NLSE type. The thesis proposes and implements several parallel algorithms for a range of modern computing architectures:

- Shared memory systems consisting of one or more multi-core CPUs;
- Hardware accelerators in the form of GPUs;
- Heterogeneous systems combining the multi-core CPUs with the GPUs;
- Distributed memory systems - computer clusters with multi-core CPU nodes;
- Distributed memory systems - computer clusters with GPU-enabled nodes;
- Distributed memory systems - computer clusters with multi-core CPU & GPU-enabled nodes.

Development and implementation of the hybrid algorithms that fully overlap the computation on all resources required major research effort, however, tuning them to get the best performance on a specific hardware combination was even more challenging. This is addressed by automating the tuning of hybrid implementations via a genetic algorithm, where a general method was developed such that it can be applied to other heterogeneous algorithms. In addition to the development of the above parallel algorithms, the optimal data distribution scheme for hybrid algorithms targeting heterogeneous systems is also presented. This scheme was selected during the research leading to this thesis and represents another contribution, as can be verified by its use for the hybrid Fourier transform algorithm, which was developed within the thesis as well. The hybrid Fourier transform algorithm is highly efficient as it enables simultaneous computation on both CPU and GPU, and can be reused independently for other purposes, thus representing an important contribution in itself. The thesis concludes its research by investigating and modeling the complexity of the developed parallel algorithms. The proposed models are tested and experimentally verified through a thorough measurement of the performance of developed programs. Finally, the thesis investigates the possibilities for visualization of large-scale simulations in real time and demonstrates its practical implementation on the example of study of vortex formation in dipolar BECs.

Following on from this introductory chapter, we discuss approaches to solving NLSE and specifically GPE along with the description of the numerical algorithm employed by all programs developed as part of this thesis, in Chapter 2. The chapter starts with an overview of both analytical and well as numerical methods for solving NLSE and GPE, accompanied by a survey of available solvers based on the numerical methods (Section 2.1). Next, we present the dimensionless dipolar GPEs in 3D, 2D and 1D (Section 2.2) which we numerically solve in this thesis, before moving on to the description of the semi-implicit split-step Crank-Nicolson numerical method which forms the basis of the algorithms used in the implementations (Section 2.3). In addition to the main method to solve the dipolar GPE, in Section 2.4 we give expressions for several relevant physical quantities that can be calculated from the wave function and used to study properties of BEC systems.

With the main numerical method described, we proceed to the description of the shared memory algorithm in Chapter 3. In Section 3.1 we describe how the numerical method can be used to construct a computer algorithm targeting shared memory systems, followed by a description of the serial implementation in Section 3.2 and parallel implementation in Section 3.3.

The shared memory algorithm can be extended to hardware accelerators like graphics processing units (GPUs), which we demonstrate in Chapter 4. To get a better understanding of how the algorithm can be implemented, we provide a brief overview of the main concepts of the programming on GPU in Section 4.1. This is followed by the description of the implementation of the shared memory algorithm using CUDA in Section 4.2.

Combining the algorithms from Chapter 3 and 4 to produce a new, hybrid, algorithm which simultaneously employs both CPU and GPU is discussed in Chapter 5. The hybrid algorithm is described in Section 5.1, followed by a naive implementation in Section 5.2, where we also note the inherent problems with such approach. The ways to improve it are discussed in Section 5.3.

Given that the single computer may not satisfy the requirements of large-scale simulations in terms of amount of processing power or available memory, in Chapter 6 we show how the algorithms from Chapters 3, 4 and 5 can be extended with MPI to work with distributed memory systems like computer clusters. To enable this, we demonstrate how data are distributed to separate processes in Section 6.1. Working with only a subset of data required changes to the way we perform computation in the main loop of our algorithms, which we describe in Section 6.2. In Section 6.3 we show how distributed memory processing can be used to improve the input and output (I/O) operations of the developed programs.

To get a better understanding of the results of the simulations created with our programs and enable study of the properties of BEC systems, in Chapter 7 we investigate how the programs can be extended to provide easy and efficient visualization of produced data. We show how the programs can be integrated into a visualization tool (Section 7.1), and how the running simulations can be made interactive (Section 7.2).

Chapter 8 is dedicated to the detailed performance evaluation of all developed programs. To extract the best performance out of the hybrid implementations, we developed several optimization techniques which automatically tune the parameters governing the division of work between CPU and GPU, details of which are given in Section 8.1. This is followed by the description of the methodology used for the tests (Section 8.2), and results of the tests along with their modeling are presented in Sections 8.3 and 8.4.

An example of how the programs can be used for research into the behavior of BEC systems is given in Chapter 9. We demonstrate how the real-world experiments can be simulated in Section 9.1, which also serves as the ultimate verification not only of the algorithms and their implementations, but also of the physical model and mean-field approximation applied. In Section 9.2 we show how simulations of potential new phenomena can be performed using our programs on the example of study of the dipole-dipole interaction effects on the critical velocity for the emergence of vortices in a dipolar BEC.

We provide a summary of the work done as part of this thesis and an outlook of future research directions in Chapter 10.

Chapter 2

Methods for solving nonlinear Schrödinger equation

Nonlinear partial differential equations are known to be difficult to solve. The same is true for the GPE and NLSE in general, although their nonlinearity is not that severe. Numerous methods for solving such kind of equations have been developed, each with different advantages and setbacks. In this chapter we discuss relevant methods for solving one form of NLSE known as dipolar GPE and provide details on the numerical method underpinning the algorithms presented in this thesis.

In Section 2.1 we give an overview of well-known methods used to solve GPE and similar NLSE. We cover both analytic approaches and numerical methods, and go over the most popular solvers. The dimensionless form of dipolar GPE in 3D as well as the effective equations of reduced dimensionality are presented in Section 2.2. Our numerical method of choice, split-step semi-implicit Crank-Nicolson method, upon which all algorithms in this thesis are based, is the subject of Section 2.3. Relevant physical quantities which are usually calculated during numerical simulations of imaginary- and real-time propagation of the GPE are discussed in Section 2.4.

2.1 Overview of solutions of NLSE and GPE

While linear differential equations of different kinds can be generally always solved and the superposition principle allows for a construction of solutions satisfying any given initial (or boundary) conditions, this is not the case with nonlinear differential equations, including NLSE and GPE. In general, unless a special method exists for a particular type of nonlinear equations, one has to use approximative approaches, either analytical or numerical. Here we discuss the most relevant methods for NLSE-type of equations.

Perturbative approach, where the unknown function is expressed as a power series in an appropriate small parameter, is one of the most popular analytical methods. In the case of NLSE and GPE, it leads to a hierarchical system of coupled linear equations in the coefficients of the power series. These equations are coupled due to nonlinearity, and usually cannot be solved analytically. The hierarchy has to be cut at a given order of the smallness parameter, which brings an infinitely large system to a finite size and enables its solving. At the same time, cutting the hierarchy introduces an approximation and associated error to the obtained solution.

There are also physics-motivated approaches, where one neglects certain terms in NLSE/GPE if they are known to be sufficiently small. For example, when studying the ground state of a BEC with strong interactions, the kinetic energy is usually small compared to other terms, and can be neglected. This is done in the Thomas-Fermi approximation, which reduces the corresponding time-independent NLSE/GPE into an algebraic equation. This method can be also applied in real-time propagation if kinetic energy is guaranteed to always remain sufficiently small, however this cannot be known in advance and has to be independently verified. Thomas-Fermi approximation can be reliably used only for very strong interactions or in the limit of infinite interaction. In the other limit, when interaction is very weak, NLSE/GPE reduces to the usual Schrödinger equation, whose solutions can be obtained by the methods tailored for this type of equations, including known analytical solutions. One can even combine these methods with the perturbative expansion in the small interaction strength.

Another physics-motivated approach is the variational method, where one assumes certain mathematical form of the solution containing several unknown parameters (ansatz). These parameters are then optimized and their values are obtained (usually) by minimizing the corresponding functional (action or energy, in the language of physics). The approximative solution obtained in this way is the best in a given class of functions, however it is difficult to assess its quality or the associated error. Selection of a suitable ansatz depends on our knowledge of the behavior of the system, i.e., solutions for particular cases or limits, numerical solutions for some values of system parameters, or experimental data. The more we know about the properties of the system, we are able to build this knowledge into an ansatz that describes more reliably the dynamical behavior and possible solutions of the equations we seek to solve.

In addition to the analytic methods, various numerical methods also play an important role, as in many cases they are the only way to study the given physical system in detail. These methods can be roughly categorized in three groups:

- split-step methods,
- finite difference methods and
- spectral methods.

In split-step methods, propagation in time is separated into several substeps, such that each of them can be performed either analytically or numerically. In addition to errors associated with each individual substep, splitting the time propagation also introduces an overall error that has to be taken into account when estimating the total error of the solution. Finite difference methods rely on the discretization of the corresponding equation and solving it as a system of linear algebraic equations. Note that variations of these methods are usually combined with other approaches, in particular with split-step methods. Spectral methods assume that the solution is written in a complete basis of orthonormal special functions, and the equations for the corresponding coefficients are calculated numerically. The choice of a basis is made so that it leads to numerically tractable system of equations, with most popular examples being the Fourier or Laplace decomposition.

In the case of the dipolar GPE, the convolution integral representing the dipole-dipole interaction term is usually calculated via the discrete Fourier transformation (DFT), which has to be combined with the chosen method of solution.

There are several available solvers for NLSE and GPE and we briefly review the most popular ones: BEC-GP [12, 13, 14, 15] and DBEC-GP [16, 17, 18] families of solvers, TS-MPI [19], GPUE [20], NLSEmagic [21], GPELab [22, 23], GSGPEs [24], ATUS-PRO [25] and GPFEM [26]. All of these solvers are publicly available and published in journals such as Computer Physics Communications.

Published serial implementations have been in development since the 2000s, however they are largely superseded by the parallel implementations. One such notable and highly cited Fortran implementation by Muruganandam and Adhikari [12] combines split-step and finite difference methods, and utilizes Crank-Nicolson scheme to produce solvers for 1D, 2D and 3D GPE with contact interaction only. Parallelizations of these solvers have been developed in C with OpenMP [13] and MPI [14], and in Fortran with OpenMP [15]. New branch that includes the dipolar interaction was published in 2015 [16], and we have subsequently parallelized these solvers using CUDA [17], as well as OpenMP and MPI [18]. Parallelizations of the dipolar programs are presented in this thesis in detail. Note that only the solvers in this branch are capable of taking into account the dipolar interaction. As far as other software packages mentioned below are concerned, only one of them (GPELab) has this capability, while all others can solve GPE with the contact interaction only.

One of such parallel solvers is the Trotter-Suzuki-MPI library (TS-MPI) [19]. This library provides a parallel and distributed implementation based on the Trotter-Suzuki algorithm [27]. The library contains kernels for efficient computation on CPU and GPU, as well as a hybrid implementation, allowing for the simultaneous use of CPU and GPU. Additionally, TS-MPI exposes several external application programming interfaces (APIs), and can thus be used from Python or C++. Unfortunately, TS-MPI library can only be used for 1D and 2D problems, and there is no support for 3D systems.

Specialized GPU solvers also exist, an example of which are GPUE [20] and NLSEmagic [21]. GPUE is based on a time-splitting pseudospectral method, implemented in CUDA and targeting only 2D systems. Being implemented in CUDA, it relies on external Python routines to process the output. It supports only a single GPU, and there is no support for multiple GPUs within the same computer or any form of distributed computation. NLSEmagic can be used to simulate the NLSE in 1D, 2D and 3D. It utilizes a fully-explicit fourth-order Runge-Kutta scheme in time and both second- and fourth-order finite difference scheme in space. While implemented in C and CUDA, NLSEmagic is designed to interface with MATLAB. Like GPUE, it also uses only a single GPU.

There are also solvers implemented fully in MATLAB, like GPELab [22, 23] and GSGPEs [24]. GPELab is a toolbox able to solve stationary and dynamics problems in 1D, 2D and 3D, including the contact, dipole-dipole and user-provided interactions, as well as rotation of the system, stochastic effects and multi-components problems. It is based on a semi-implicit backward Euler scheme, a pseudo-spectral approximation and a Krylov subspace method [28]. While very popular due to its versatility, GPELab was shown to have very bad performance when compared to implementations based on compiled languages [29]. The other MATLAB-based implementation, GSGPEs, has fewer features and can only be used for the computation of the ground state of systems. It also relies on spectral decomposition method to perform this task.

Two solvers based on the finite element method are ATUS-PRO [25] and GPFEM [26]. ATUS-PRO calculates the solutions of the stationary and the time-dependent 1D, 2D and 3D GPE with contact interaction. Programs in this package are implemented in C++ by means of the deal.II library, and support running in parallel on distributed memory systems using MPI. Numerical method employed in the solver is based on the fully implicit Crank-Nicolson method, while stationary states are obtained with a constrained Newton-like method. The other finite element-based solver, GPFEM, can be used to compute the stationary solutions of GPE with rotation, in 2D or 3D. It is implemented in FreeFem++, a free finite element toolbox with its own language. An interesting feature of this solver is that it offers two numerical methods for minimization of GP energy for the user to choose from: a steepest descent method based on Sobolev gradients and a minimization algorithm based on the optimization library Ipopt. This toolkit also provides several post-processing tools for tasks such as identification of quantized vortices, that can help in extracting values of physical quantities from the simulations.

As we can see from this overview, many of the previous implementations focus on a specific problem (e.g., computation of the stationary state), or on a specific dimensionality. Only DBEC-GP solver and GPELab are able to address the dipolar interaction, however the first one is serial and the other one is implemented in MATLAB, leading to their poor performance, in particular when treating 3D systems. One of the aims of this thesis is to overcome this problem and to develop and implement parallel algorithms, which are presented in later chapters.

2.2 Dipolar GPE in 3D, 2D and 1D

Many important properties of BEC systems at low temperatures can be successfully described using the mean-field theory, where quantum fluctuations are neglected and where we assume zero temperature, as already mentioned in Chapter 1. If we take into account scattering of the atoms or molecules through the nonlinear contact interaction term, and the dipole-dipole interaction assuming a fixed orientation of the dipoles along z direction, the corresponding mean-field Gross-Pitaevskii equation has a form

$$i\hbar\frac{\partial}{\partial t}\Psi(\mathbf{r};t) = \left[-\frac{\hbar^2}{2m}\nabla^2 + V(\mathbf{r}) + gN_{\text{at}}|\Psi(\mathbf{r};t)|^2 + N_{\text{at}}\int U_{\text{dd}}(\mathbf{r}-\mathbf{r}')|\Psi(\mathbf{r}';t)|^2 d\mathbf{r}'\right]\Psi(\mathbf{r};t)\,, \quad (2.1)$$

where the nonlinearity $g = 4\pi\hbar^2 a_s/m$ is determined by the s-wave scattering length a_s, a quantity used in atomic physics for characterizing the interactions of atoms in the low-energy limit. N_{at} is the total number of condensed particles and $V(\mathbf{r})$ is the trapping potential, usually anisotropic quadratic function of the form

$$V(\mathbf{r}) = \frac{1}{2}m(\omega_x^2 x^2 + \omega_y^2 y^2 + \omega_z^2 z^2)\,, \quad (2.2)$$

where ω_x, ω_y and ω_z are the corresponding frequencies. However, in general, it may be any arbitrary function depending on position and time $V(\mathbf{r}, t)$. The normalization condition of the wave function is given by

$$\int d\mathbf{r}|\Psi(\mathbf{r};t)|^2 = 1\,, \quad (2.3)$$

which ensures that the number of particles in the system is fixed to N_{at} at all times. The boundary conditions the solution has to satisfy read

$$\lim_{\mathbf{r}\to\pm\infty}\Psi(\mathbf{r};t) = 0\,. \quad (2.4)$$

The dipolar interaction potential is given by

$$U_{\mathrm{dd}}(\mathbf{R}) = \frac{C_{\mathrm{dd}}}{4\pi} \frac{1 - 3\cos^2\theta}{|\mathbf{R}|^3}, \tag{2.5}$$

where $\mathbf{R} = r - r'$ determines relative position of the dipoles, θ is the angle between $\mathbf{R}$ and the orientation of dipoles, and $C_{\mathrm{dd}} = \mu_0 \bar{\mu}^2$ for magnetic dipoles and $C_{\mathrm{dd}} = d^2/\varepsilon_0$ for electric dipoles. To compare contact and dipolar interactions, a new length scale is introduced, characterizing the dipole-dipole interaction:

$$a_{\mathrm{dd}} = \frac{C_{\mathrm{dd}} m}{12\pi\hbar^2}, \tag{2.6}$$

which also allows the dipolar GPE to be written in a more compact way.

In order to numerically simulate GPE, we first have to cast it into dimensionless form and to express all quantities through dimensionless variables. To achieve this, we start by choosing a reference frequency $\tilde{\omega}$, which may be equal to one of the trap frequencies, or have some other value. Now we can express the trap frequencies by the corresponding dimensionless quantities, $\omega_x = \gamma\tilde{\omega}$, $\omega_y = \nu\tilde{\omega}$ and $\omega_z = \lambda\tilde{\omega}$. We also use $\tilde{\omega}$ to define a unit of time $1/\tilde{\omega}$ and to introduce the dimensionless time as $\tilde{t} = t\tilde{\omega}$. The corresponding harmonic oscillator length $l = \sqrt{\hbar/m\tilde{\omega}}$ is used as a unit of length, and the dimensionless coordinates are defined as $\tilde{x} = x/l$, $\tilde{y} = y/l$ and $\tilde{z} = z/l$. Finally, we express the wave function in a dimensionless form by $\tilde{\Psi} = l^{3/2}\Psi$. Using this rescaling, GPE (2.1) reduces to

$$\begin{aligned} i\frac{\partial}{\partial \tilde{t}}\tilde{\Psi}(\tilde{\mathbf{r}};\tilde{t}) = \Bigg[& -\frac{1}{2}\tilde{\nabla}^2 + \frac{1}{2}(\gamma^2\tilde{x}^2 + \nu^2\tilde{y}^2 + \lambda^2\tilde{z}^2) + 4\pi\tilde{a}_s N_{\mathrm{at}}|\tilde{\Psi}(\tilde{\mathbf{r}};\tilde{t})|^2 \\ & + 3\tilde{a}_{\mathrm{dd}} N_{\mathrm{at}} \int d\tilde{\mathbf{r}}' V_{\mathrm{dd}}(\tilde{\mathbf{r}} - \tilde{\mathbf{r}}')|\tilde{\Psi}(\tilde{\mathbf{r}}';\tilde{t})|^2 \Bigg] \tilde{\Psi}(\tilde{\mathbf{r}};\tilde{t}), \end{aligned} \tag{2.7}$$

where the dimensionless dipolar interaction potential reads

$$V_{\mathrm{dd}}(\tilde{\mathbf{R}}) = \frac{1 - 3\cos^2\theta}{|\tilde{\mathbf{R}}|^3}, \tag{2.8}$$

and we have introduced the dimensionless scattering length $\tilde{a}_s = a_s/l$ and the dipole-dipole interaction length $\tilde{a}_{\mathrm{dd}} = a_{\mathrm{dd}}/l$.

It is worth noting that the above conversion to dimensionless quantities is not the only way to do so. For example, we can additionally rescale the time so that instead of $\tilde{t}$ we use $2\tilde{t}$. This would make the factor of 1/2 disappear from the kinetic energy and harmonic potential terms, which may be preferred in some cases. If we additionally drop the tilde symbol from all quantities, GPE takes the form

$$\begin{aligned} i\frac{\partial}{\partial t}\Psi(\mathbf{r};t) = \Bigg[& -\nabla^2 + \gamma^2 x^2 + \nu^2 y^2 + \lambda^2 z^2 + g N_{\mathrm{at}}|\Psi(\mathbf{r};t)|^2 \\ & + g_{\mathrm{dd}} N_{\mathrm{at}} \int d\mathbf{r}' V_{\mathrm{dd}}(\mathbf{r} - \mathbf{r}')|\Psi(\mathbf{r}';t)|^2 \Bigg] \Psi(\mathbf{r};t), \end{aligned} \tag{2.9}$$

where the dimensionless nonlinearity is defined as $g = 8\pi a_s/l$ and the dimensionless dipolar interaction strength is $g_{\mathrm{dd}} = 6a_{\mathrm{dd}}/l$.

In many cases, the geometry of the system is such that one can use effective equations with reduced dimensionality. Lower dimensional equations are analytically more complex and less tractable, but their numerical complexity significantly reduces and thus this effective approach is widely used whenever possible.

In disc-shaped external potentials, when one of the trapping frequencies is much larger than the other two, we can effectively use 2D geometry and derive appropriate equations assuming that a BEC system remains confined to a ground state of harmonic oscillator in the third (confining) direction. We can consider two physically distinct cases, when the dipoles are oriented along the confining direction ($\lambda \gg \gamma, \nu$) and when the dipoles are in the 2D plane (e.g., $\nu \gg \gamma, \lambda$). The corresponding dimensionless effective equations have been derived in Ref. [16]. For the first case, when the dipoles are orthogonal to the plane, the wave function is written as $\Psi(\mathbf{r};t) = \Psi_0(z;\lambda)\Psi_{\mathrm{2D}}(\mathbf{r}_{\mathrm{2D}};t)$, where the ground state in z direction is given as

$$\Psi_0(z;\lambda) = \left(\frac{\lambda}{\pi}\right)^{1/4} e^{-\lambda z^2/2}\,, \tag{2.10}$$

and $\mathbf{r}_{\mathrm{2D}} = (x, y)$ is 2D radius vector. Using this substitution, GPE (2.9) reduces to

$$\begin{aligned} i\frac{\partial}{\partial t}\Psi_{\mathrm{2D}}(\mathbf{r}_{\mathrm{2D}};t) = \Bigg[& -\nabla^2_{\mathrm{2D}} + \gamma^2x^2 + \nu^2y^2 + g_{\mathrm{2D}}N_{\mathrm{at}}|\Psi_{\mathrm{2D}}(\mathbf{r}_{\mathrm{2D}};t)|^2 \\ & + g_{\mathrm{dd,2D}}N_{\mathrm{at}}\int\frac{d\mathbf{k}_{\mathrm{2D}}}{(2\pi)^2}e^{-i\mathbf{k}_{\mathrm{2D}}\cdot\mathbf{r}_{\mathrm{2D}}}\tilde{n}_{\mathrm{2D}}(\mathbf{k}_{\mathrm{2D}};t)h_{\mathrm{2D}}\left(\frac{k_{\mathrm{2D}}}{\sqrt{2\lambda}}\right)\Bigg]\Psi_{\mathrm{2D}}(\mathbf{r}_{\mathrm{2D}};t)\,, \end{aligned} \tag{2.11}$$

where ∇_{2D} represents 2D gradient operator, the effective dimensionless interaction strengths are $g_{\mathrm{2D}} = \sqrt{\lambda/(2\pi)}\;8\pi a_s/l$ and $g_{\mathrm{dd,2D}} = \sqrt{\lambda/(2\pi)}\;8\pi a_{\mathrm{dd}}/l$, $\mathbf{k}_{\mathrm{2D}} = (k_x, k_y)$ is the 2D k-vector, and

$$\tilde{n}_{\mathrm{2D}}(\mathbf{k}_{\mathrm{2D}};t) = \int d\mathbf{r}_{\mathrm{2D}}e^{i\mathbf{k}_{\mathrm{2D}}\cdot\mathbf{r}_{\mathrm{2D}}}|\Psi_{\mathrm{2D}}(\mathbf{r}_{\mathrm{2D}};t)|^2\,, \tag{2.12}$$

$$h_{\mathrm{2D}}(\xi) = 2 - 3\sqrt{\pi}\xi\exp(\xi^2)\{1-\mathrm{erf}(\xi)\}\,. \tag{2.13}$$

For the second case, when dipoles lie in the plane, the wave function can be expressed as $\Psi(\mathbf{r};t) = \Psi_0(y;\nu)\Psi_{\mathrm{2D}}(\mathbf{r}_{\mathrm{2D}};t)$, where the ground state is defined by Eq. (2.10) and 2D radius vector is now $\mathbf{r}_{\mathrm{2D}} = (x, z)$. The corresponding effective 2D GPE has a form

$$\begin{aligned} i\frac{\partial}{\partial t}\Psi_{\mathrm{2D}}(\mathbf{r}_{\mathrm{2D}};t) = \Bigg[& -\nabla^2_{\mathrm{2D}} + \gamma^2x^2 + \lambda^2z^2 + g_{\mathrm{2D}}N_{\mathrm{at}}|\Psi_{\mathrm{2D}}(\mathbf{r}_{\mathrm{2D}};t)|^2 \\ & + g_{\mathrm{dd,2D}}N_{\mathrm{at}}\int\frac{d\mathbf{k}_{\mathrm{2D}}}{(2\pi)^2}e^{-i\mathbf{k}_{\mathrm{2D}}\cdot\mathbf{r}_{\mathrm{2D}}}\tilde{n}_{\mathrm{2D}}(\mathbf{k}_{\mathrm{2D}};t)j_{\mathrm{2D}}\left(\frac{k_{\mathrm{2D}}}{\sqrt{2\nu}}, k_z\right)\Bigg]\Psi_{\mathrm{2D}}(\mathbf{r}_{\mathrm{2D}};t)\,, \end{aligned} \tag{2.14}$$

where ∇_{2D} now represents 2D gradient operator in x-z plane, $g_{\mathrm{2D}} = \sqrt{\nu/(2\pi)}\;8\pi a_s/l$ and $g_{\mathrm{dd,2D}} = \sqrt{\nu/(2\pi)}\;8\pi a_{\mathrm{dd}}/l$ are the effective interaction strengths, $\mathbf{k}_{\mathrm{2D}} = (k_x, k_z)$ is the corresponding 2D k-vector, and

$$j_{\mathrm{2D}}(\xi, k_z) = -1 + 3\sqrt{\pi}\frac{k_z^2}{2\nu\xi}\exp(\xi^2)\{1-\mathrm{erf}(\xi)\}\,. \tag{2.15}$$

Similar dimensional reduction can be performed for BEC systems in elongated, cigar-shaped traps [16], where again we can distinguish two cases: when the dipoles are orthogonal to the trap axis (without loss of generality, we consider just the geometry $\nu, \lambda \gg \gamma$) and when they are parallel to the trap axis ($\gamma, \nu \gg \lambda$). In a cigar-shaped trap, a BEC system is assumed to remain confined to a ground state of a harmonic oscillator in two transversal directions (y and z in the first case, and x and y in the second), and we consider the wave function to depend only on one remaining coordinate. In the first case, the wave function is written as $\Psi(\mathbf{r};t) = \Psi_0(y;\nu)\Psi_0(z;\lambda)\Psi_{1\mathrm{D}}(x;t)$, and assuming axial symmetry ($\nu = \lambda$) the effective 1D GPE can be expressed in a closed form

$$i\frac{\partial}{\partial t}\Psi_{1\mathrm{D}}(x;t) = \Bigg[-\frac{\partial^2}{\partial x^2} + \gamma^2 x^2 + g_{1\mathrm{D}} N_{\mathrm{at}} |\Psi_{1\mathrm{D}}(x;t)|^2 + g_{\mathrm{dd},1\mathrm{D}} N_{\mathrm{at}} \int_{-\infty}^{\infty} \frac{dk_x}{2\pi} e^{-ik_x x} \tilde{n}_{1\mathrm{D}}(k_x;t) j_{1\mathrm{D}}\left(\frac{k_x}{\sqrt{2\nu}}\right)\Bigg]\Psi_{1\mathrm{D}}(x;t)\,, \tag{2.16}$$

where interactions strengths are $g_{1\mathrm{D}} = 4\nu a_s/l$ and $g_{\mathrm{dd},1\mathrm{D}} = 4\nu a_{\mathrm{dd}}/l$, and

$$\tilde{n}_{1\mathrm{D}}(k_x;t) = \int_{-\infty}^{\infty} dx e^{ik_x x} |\Psi_{1\mathrm{D}}(x;t)|^2\,, \tag{2.17}$$

$$j_{1\mathrm{D}}(\xi) = \frac{1}{\sqrt{\pi}} \int_{-\infty}^{\infty} dk_y e^{-k_y^2} h_{2\mathrm{D}}\left(\sqrt{\xi^2 + k_y^2}\right)\,. \tag{2.18}$$

In the second case, when dipoles are parallel to the trap axis ($\gamma, \nu \gg \lambda$), the wave function is written as $\Psi(\mathbf{r};t) = \Psi_0(x;\gamma)\Psi_0(y;\nu)\Psi_{1\mathrm{D}}(z;t)$. Again, assuming axial symmetry ($\gamma = \nu$) the closed form of the effective 1D GPE reads

$$i\frac{\partial}{\partial t}\Psi_{1\mathrm{D}}(z;t) = \Bigg[-\frac{\partial^2}{\partial z^2} + \lambda^2 z^2 + g_{1\mathrm{D}} N_{\mathrm{at}} |\Psi_{1\mathrm{D}}(z;t)|^2 + g_{\mathrm{dd},1\mathrm{D}} N_{\mathrm{at}} \int_{-\infty}^{\infty} \frac{dk_z}{2\pi} e^{-ik_z z} \tilde{n}_{1\mathrm{D}}(k_z;t) h_{1\mathrm{D}}\left(\frac{k_z}{\sqrt{2\gamma}}\right)\Bigg]\Psi_{1\mathrm{D}}(z;t)\,, \tag{2.19}$$

where interaction strengths are $g_{1\mathrm{D}} = 4\gamma a_s/l$ and $g_{\mathrm{dd},1\mathrm{D}} = 2\gamma a_{\mathrm{dd}}/l$, and

$$h_{1\mathrm{D}}(\xi) = \int_0^{\infty} du \left[\frac{3\xi^2}{u + \xi^2} - 1\right] e^{-u}\,. \tag{2.20}$$

The above equations in 3D, as well as the effective equations in 2D and 1D are solved by the algorithms and programs presented in this thesis.

2.3 Split-step Crank-Nicolson method for the GPE

With the dimensionless form of GPE introduced in the previous section, we proceed with the description of *split-step semi-implicit Crank-Nicolson method* [30], the approach we use to numerically solve GPE in 3D as well as effective 2D and 1D equations. The basic idea behind this method is to divide the calculation of the solution into multiple steps at two levels. First, time propagation is discretized and the total propagation time T is divided into N time steps, each of length $\Delta = T/N$. At the second level, each time step is divided into several substeps, dealing with different parts of the Hamiltonian governing the time propagation. This second level of splitting uses semi-implicit Crank-Nicolson (CN) method, which we describe in detail here.

We first demonstrate how this method works on a simple example in 1D, before moving on to the solution of full dipolar GPE. Let us consider the equation of the general form

$$i\frac{\partial}{\partial t}\Psi(x;t) = H\Psi(x;t)\,, \tag{2.21}$$

where the Hamiltonian H contains the kinetic term, i.e., the Laplacian, as well as other non-derivative terms, including the potential and nonlinear terms. Time propagation starts from a known initial condition at $t = t_0$ and proceeds in N time steps of the duration Δ. To propagate the wave function in one time step, we split H into two parts, $H = H_1 + H_2$, where H_1 contains all non-derivative terms, while H_2 is the remaining Laplacian. This allows us to make an approximation and split Eq. (2.21) into two parts, which we consider and solve separately, one after the other,

$$i\frac{\partial}{\partial t}\Psi(x;t) = H_1\Psi(x;t)\,, \tag{2.22}$$

$$i\frac{\partial}{\partial t}\Psi(x;t) = H_2\Psi(x;t)\,. \tag{2.23}$$

Equation (2.22) is solved first, with an initial value $\Psi(x;t_0)$ at $t = t_0$, to obtain an intermediate solution at $t = t_0 + \Delta$. This intermediate solution is then used as the initial value for Eq. (2.23), yielding the final solution $\Psi(x;t_0+\Delta)$ at $t = t_0+\Delta$. This procedure then continues in the next time step until we finish time propagation. As we mentioned above, the time splitting is an approximation and introduces an error of the order of Δ^2, which may be neglected when used with small time step Δ. However, such a scheme allows us to treat the larger part of the Hamiltonian exactly, even for arbitrarily large nonlinear terms, as we show below.

Let us denote the wave function at time $t_n = t_0+n\Delta$ by $\Psi^n(x)$ and the corresponding intermediate solution of Eq. (2.22) by $\Psi^{n+1/2}(x)$. Since H_1 does not contain any derivatives, the intermediate solution can be explicitly calculated as

$$\Psi^{n+1/2}(x) = \mathcal{O}_{\text{nd}}(H_1)\Psi^n(x) \equiv e^{-i\Delta H_1}\Psi^n(x)\,, \tag{2.24}$$

where we use $\mathcal{O}_{\text{nd}}(H_1)$ to denote the time evolution operator with respect to H_1 and the suffix "nd", short for "non-derivative", to differentiate it from other similar operators we use.

Next step is to perform the time propagation w.r.t. H_2, by using semi-implicit CN scheme,

$$i\frac{\Psi^{n+1}(x) - \Psi^{n+1/2}(x)}{\Delta} = \frac{1}{2}H_2\left[\Psi^{n+1}(x) + \Psi^{n+1/2}(x)\right]\,, \tag{2.25}$$

where the partial derivative $\partial/\partial t$ is approximated using a two-point formula connecting the values of the wave function in the present and the future time step. On the right-hand side hand, the Laplacian $\partial^2/\partial x^2$ is written in a semi-implicit form, which averages the wave function over the present and the future time step. This averaging makes the scheme unconditionally stable [31, 32].

The solution of this time-discretized equation with the initial condition $\Psi^{n+1/2}(x)$, obtained in the previous substep, yields the wave function $\Psi^{n+1}(x)$ at time t_{n+1}. Formally, it is solved as

$$\Psi^{n+1}(x) = \mathcal{O}_{\text{CN}}(H_2)\Psi^{n+1/2}(x) \equiv \frac{1 - i\Delta H_2/2}{1 + i\Delta H_2/2}\,\Psi^{n+1/2}(x)\,. \tag{2.26}$$

Similarly to the $\mathcal{O}_{\text{nd}}(H_1)$ operator, the $\mathcal{O}_{\text{CN}}(H_2)$ operation denotes the time evolution operator w.r.t. H_2. To actually solve Eq. (2.25), we also have to discretize spatial coordinate and introduce a mesh with N_x equidistant points, separated by a spacing h. Mesh points are defined by $x_i = -L+ih$, where $L = N_xh/2$ determines the size of the spatial interval $[-L,\, L]$ considered to be relevant, i.e., such that the wave function is sufficiently small outside of it and we can safely set it to zero in our calculations. If we denote the wave function at time t_n and position x_i by Ψ_i^n, Eq. (2.25) can be written as

$$i\frac{\Psi_i^{n+1} - \Psi_i^{n+1/2}}{\Delta} = \frac{\Psi_{i+1}^{n+1} - 2\Psi_i^{n+1} + \Psi_{i-1}^{n+1} + \Psi_{i+1}^{n+1/2} - 2\Psi_i^{n+1/2} + \Psi_{i-1}^{n+1/2}}{2h^2}\,, \tag{2.27}$$

where each Laplacian is approximated using a three-point formula. The associated discretization error is of second order in h, the same as in time step Δ, as discussed previously. This makes discretization errors negligible for small enough discretization steps. Equation (2.27) can be written as a series of tridiagonal equations in unknown wave function values at t_{n+1},

$$A_i^- \Psi_{i-1}^{n+1} + A_i^0 \Psi_i^{n+1} + A_i^+ \Psi_{i+1}^{n+1} = B_i \,, \tag{2.28}$$

where the coefficients

$$A_i^- = A_i^+ = -i\frac{\Delta}{2h^2} \,, \tag{2.29}$$

$$A_i^0 = 1 + i\frac{\Delta}{h^2} \,, \tag{2.30}$$

$$B_i = i\frac{\Delta}{2h^2}\left(\Psi_{i+1}^{n+1/2} - 2\Psi_i^{n+1/2} + \Psi_{i-1}^{n+1/2}\right) + \Psi_i^{n+1/2} \,, \tag{2.31}$$

are all known. To solve Eq. (2.28), we use forward recursion relation, and assume the solution in a form

$$\Psi_{i+1}^{n+1} = \alpha_i \Psi_i^{n+1} + \beta_i \,, \tag{2.32}$$

where coefficients α_i and β_i need to be determined. If we substitute Eq. (2.32) into Eq. (2.28) we obtain

$$\Psi_i^{n+1} = \gamma_i \left(A_i^- \Psi_{i-1}^{n+1} + A_i^+ \beta_i - B_i\right) \,, \tag{2.33}$$

where $\gamma_i = -1/(A_i^0 + A_i^+ \alpha_i)$. We can now use the backward recursion relation to obtain the coefficients α_i and β_i,

$$\alpha_i = \gamma_{i+1} A_{i+1}^- \,, \quad \beta_i = \gamma_{i+1}(A_{i+1}^+ \beta_{i+1} - B_{i+1}) \,. \tag{2.34}$$

The strategy to apply the aforementioned recursion relations from Eqs. (2.32)–(2.34) is to perform a backward sweep of the mesh to determine α_i and β_i, for i running from $N_x - 2$ to 0. The initial values on endpoints are chosen to be $\alpha_{N_x-1} = \beta_{N_x-1} = 0$, which ensures that the value of the wave function at the border is always zero. With the coefficients α_i, β_i and γ_i determined, and the boundary condition $\Psi_0^{n+1} = 0$ fixed to provide that the wave function is equal to zero at $x_0 = -L$, we can determine the solution for the entire space range at time t_{n+1} using a forward sweep from Eq. (2.32). This completes the single iteration and procedure continues on until end of propagation is reached.

Note that, apart from unconditional stability and small discretization errors, the above scheme has one additional advantage, namely it conserves the normalization of the wave function. This can be monitored throughout the calculation and used as a criterion to assess the quality and validity of the obtained solution.

The method illustrated above in 1D can be straightforwardly generalized to solving 3D problems. For a full 3D dipolar GPE (2.9), discretization of time is performed in the same way, while spatial coordinates are discretized analogously with N_x, N_y and N_z mesh points in the corresponding direction, and with spacings h_x, h_y and h_z, respectively. Time propagation in each iteration is now divided into four substeps, corresponding to the following parts the Hamiltonian:

$$H_1 = \gamma^2 x^2 + \nu^2 y^2 + \lambda^2 z^2 + gN_{\mathrm{at}}|\Psi(\mathbf{r};t)|^2 + g_{\mathrm{dd}}N_{\mathrm{at}}\int d\mathbf{r}' V_{\mathrm{dd}}(\mathbf{r}-\mathbf{r}')|\Psi(\mathbf{r}';t)|^2 \,, \tag{2.35}$$

$$H_2 = -\frac{\partial^2}{\partial x^2} \,, \quad H_3 = -\frac{\partial^2}{\partial y^2} \,, \quad H_4 = -\frac{\partial^2}{\partial z^2} \,. \tag{2.36}$$

To propagate the wave function w.r.t. H_1, we need to calculate the expression (2.35) at each mesh point. This requires evaluation of the dipolar term given by the convolution integral, which can be written and calculated as

$$\int d\mathbf{r}' V_{\mathrm{dd}}(\mathbf{r}-\mathbf{r}')|\Psi(\mathbf{r}';t)|^2 = \mathcal{F}^{-1}\Big[\mathcal{F}\big[V_{\mathrm{dd}}(\mathbf{r})\big]\mathcal{F}\big[|\Psi(\mathbf{r};t)|^2\big]\Big]\,, \tag{2.37}$$

where $\mathcal{F}$ represents Fourier transform and $\mathcal{F}^{-1}$ its inverse. These can be computed using the standard Fast Fourier Transform (FFT) algorithm. Note that $\mathcal{F}\big[V_{\mathrm{dd}}(\mathbf{r})\big]$ does not change between time steps, and can be computed analytically, yielding

$$\mathcal{F}\big[V_{\mathrm{dd}}(\mathbf{r})\big] = 3\cos^2\theta - 1\,, \tag{2.38}$$

where θ is the angle between the orientation of dipoles and vector $\mathbf{k}$. As we assume that the dipoles are oriented in z direction, $\cos\theta = k_z/k$. This leaves us with two Fourier transforms in each time step and in particular motivates our interest for implementation of this approach on GPUs.

The above method is applicable to real-time dynamics of the system, given that its initial state is known. However, we can extend the method to also enable calculation of the ground state. This is achieved by switching to *imaginary-time* propagation, when real time t is replaced by an imaginary quantity $\tau = it$. Mathematically, this corresponds to a Wick rotation [33] and yields imaginary-time analogue of GPE,

$$-\frac{\partial}{\partial\tau}\Psi(\mathbf{r};\tau) = H\Psi(\mathbf{r};\tau)\,, \tag{2.39}$$

where $H = H_1 + H_2 + H_3 + H_4$ is the same as above. Although such a Wick rotation is just a formal mathematical transformation of GPE and does not have any physical motivation, time propagation of the above equation leads to a solution of the time-independent GPE (1.3),

$$\Psi_0(\mathbf{r}) = \lim_{\tau\to\infty}\Psi(\mathbf{r};\tau)\,, \tag{2.40}$$

provided that the normalization of the wave function is kept equal to unity during the time propagation, as it is not conserved any more. Equation (2.40) follows from the decomposition of the wave function in the eigenbasis of the Hamiltonian for the Schrödinger equation and also holds for GPE [12]. Therefore, to obtain a ground state we start with an arbitrary initial state (usually taken to be a Gaussian) and propagate it in imaginary time until the convergence is achieved. We also note that the ground state of a system described by GPE can always be represented by a real-valued wave function, which can be exploited to obtain the solution faster. Namely, working with real-valued data on a computer is faster than with the complex-valued ones, since all mathematical calculations require less operations. Practically, this works as follows: the time propagation w.r.t. H_1 is performed using a formula analogous to Eq. (2.24),

$$\Psi^{n+1/2} = \mathcal{O}_{\mathrm{nd}}(H_1)\Psi^n \equiv e^{-\Delta H_1}\Psi^n\,, \tag{2.41}$$

while the coefficients of the CN scheme (2.29)–(2.31) are now real-valued and (in 1D) defined as

$$A_i^- = A_i^+ = \frac{\Delta}{2h^2}\,,\quad A_i^0 = 1 - \frac{\Delta}{h^2}\,,\quad B_i = -\frac{\Delta}{2h^2}\left(\Psi_{i+1}^{n+1/2} - 2\Psi_i^{n+1/2} + \Psi_{i-1}^{n+1/2}\right) + \Psi_i^{n+1/2}\,. \tag{2.42}$$

Here Δ represents time step in imaginary time and generalization to 3D is straightforward.

Similar procedure is applied to solve effective GPEs of reduced dimensionality. In 2D, depending on the orientation of dipoles, the Hamiltonian corresponding to GPE (2.11) or (2.14) is written in the split form $H = H_1 + H_2 + H_3$. The non-derivative part H_1 is already expressed in the form suitable for evaluation using FFT, while the Laplacians H_2 and H_3 are addressed using the CN scheme, as described above. Effective 1D GPEs (2.16) and (2.19) are solved in the same way, using the split form of the Hamiltonian $H = H_1 + H_2$. Similarly to 2D case, the non-derivative part H_1 is written in k-space and allows for efficient evaluation using FFT.

With the main numerical method described above, we can define a computer algorithm for real- and imaginary-time propagation in 1D, 2D and 3D. While this is the main goal that will be addressed in subsequent chapters, we first give expressions for calculation of relevant physical quantities in the next section.

2.4 Calculation of relevant physical quantities

Numerical methods presented in this thesis enable us to solve different variants of GPE in a number of physically relevant problems. However, in order to study properties of physical systems described by those equations as well as their dynamical evolution, we have to extract values of relevant physical quantities from the current state of the system, i.e., from the wave function in a given time t. The most useful quantities are chemical potential, energy, and expectation value of the system size. In particular, convergence of these quantities during the imaginary-time propagation is used as a criterion for the convergence of the wave function to the ground state of the system.

Chemical potential μ can be defined for stationary states of a BEC and follows from the time-independent GPE (1.3):

$$\mu = \frac{1}{2}\int d\mathbf{r}\Big[|\nabla\Psi(\mathbf{r})|^2 + (\gamma^2x^2 + \nu^2y^2 + \lambda^2z^2)|\Psi(\mathbf{r})|^2 + gN_{\mathrm{at}}|\Psi(\mathbf{r})|^4 \\ + g_{\mathrm{dd}}N_{\mathrm{at}}\int d\mathbf{r}' V_{\mathrm{dd}}(\mathbf{r}-\mathbf{r}')|\Psi(\mathbf{r}')|^2|\Psi(\mathbf{r})|^2\Big]\,. \tag{2.43}$$

The corresponding energy of the system E is given by a similar expression in which the interaction energy terms contain an additional factor of 1/2:

$$E = \frac{1}{2}\int d\mathbf{r}\Big[|\nabla\Psi(\mathbf{r})|^2 + (\gamma^2x^2 + \nu^2y^2 + \lambda^2z^2)|\Psi(\mathbf{r})|^2 + \frac{1}{2}gN_{\mathrm{at}}|\Psi(\mathbf{r})|^4 \\ + \frac{1}{2}g_{\mathrm{dd}}N_{\mathrm{at}}\int d\mathbf{r}' V_{\mathrm{dd}}(\mathbf{r}-\mathbf{r}')|\Psi(\mathbf{r}')|^2|\Psi(\mathbf{r})|^2\Big]\,. \tag{2.44}$$

The above equations are written for the rescaling of time $\tilde{t} = 2\tilde{\omega}t$, used throughout previous section. If other standard rescaling of time is used, $\tilde{t} = \tilde{\omega}t$, the factor of 1/2 in front of both Eqs. (2.43) and (2.44) should be removed. Note that the dimensionless values of μ and E can be converted back to physical units by multiplying them with $\hbar\tilde{\omega}$.

Similarly, we can define chemical potential and energy for stationary states of effectively 2D and 1D systems. For 2D case when dipoles are oriented perpendicular to the plane, described by GPE (2.11), chemical potential is given by

$$\mu = \frac{1}{2}\int d\mathbf{r}_{2\mathrm{D}}\Bigg[|\nabla_{2\mathrm{D}}\Psi_{2\mathrm{D}}(\mathbf{r}_{2\mathrm{D}})|^2 + (\gamma^2x^2 + \nu^2y^2)|\Psi_{2\mathrm{D}}(\mathbf{r}_{2\mathrm{D}})|^2 + g_{2\mathrm{D}}N_{\mathrm{at}}|\Psi_{2\mathrm{D}}(\mathbf{r}_{2\mathrm{D}})|^4 \\ + g_{\mathrm{dd,2D}}N_{\mathrm{at}}\int \frac{d\mathbf{k}_{2\mathrm{D}}}{(2\pi)^2}e^{-i\mathbf{k}_{2\mathrm{D}}\cdot\mathbf{r}_{2\mathrm{D}}}\tilde{n}_{2\mathrm{D}}(\mathbf{k}_{2\mathrm{D}})h_{2\mathrm{D}}\left(\frac{k_{2\mathrm{D}}}{\sqrt{2\lambda}}\right)|\Psi_{2\mathrm{D}}(\mathbf{r}_{2\mathrm{D}})|^2\Bigg]\,, \tag{2.45}$$

while the energy can calculated as

$$E = \frac{1}{2}\int d\mathbf{r}_{2\mathrm{D}} \Bigg[|\nabla_{2\mathrm{D}}\Psi_{2\mathrm{D}}(\mathbf{r}_{2\mathrm{D}})|^2 + (\gamma^2 x^2 + \nu^2 y^2)|\Psi_{2\mathrm{D}}(\mathbf{r}_{2\mathrm{D}})|^2 + \frac{1}{2} g_{2\mathrm{D}} N_{\mathrm{at}} |\Psi_{2\mathrm{D}}(\mathbf{r}_{2\mathrm{D}})|^4 + \frac{1}{2} g_{\mathrm{dd},2\mathrm{D}} N_{\mathrm{at}} \int \frac{d\mathbf{k}_{2\mathrm{D}}}{(2\pi)^2} e^{-i\mathbf{k}_{2\mathrm{D}}\cdot\mathbf{r}_{2\mathrm{D}}} \tilde{n}_{2\mathrm{D}}(\mathbf{k}_{2\mathrm{D}}) h_{2\mathrm{D}}\left(\frac{k_{2\mathrm{D}}}{\sqrt{2\lambda}}\right) |\Psi_{2\mathrm{D}}(\mathbf{r}_{2\mathrm{D}})|^2 \Bigg]. \tag{2.46}$$

In the second 2D case, when dipoles lie in the plane, the chemical potential and energy are given by

$$\mu = \frac{1}{2}\int d\mathbf{r}_{2\mathrm{D}} \Bigg[|\nabla_{2\mathrm{D}}\Psi_{2\mathrm{D}}(\mathbf{r}_{2\mathrm{D}})|^2 + (\gamma^2 x^2 + \lambda^2 z^2)|\Psi_{2\mathrm{D}}(\mathbf{r}_{2\mathrm{D}})|^2 + g_{2\mathrm{D}} N_{\mathrm{at}} |\Psi_{2\mathrm{D}}(\mathbf{r}_{2\mathrm{D}})|^4 + g_{\mathrm{dd},2\mathrm{D}} N_{\mathrm{at}} \int \frac{d\mathbf{k}_{2\mathrm{D}}}{(2\pi)^2} e^{-i\mathbf{k}_{2\mathrm{D}}\cdot\mathbf{r}_{2\mathrm{D}}} \tilde{n}_{2\mathrm{D}}(\mathbf{k}_{2\mathrm{D}}) j_{2\mathrm{D}}\left(\frac{k_{2\mathrm{D}}}{\sqrt{2\nu}}, k_z\right) |\Psi_{2\mathrm{D}}(\mathbf{r}_{2\mathrm{D}})|^2 \Bigg], \tag{2.47}$$

$$E = \frac{1}{2}\int d\mathbf{r}_{2\mathrm{D}} \Bigg[|\nabla_{2\mathrm{D}}\Psi_{2\mathrm{D}}(\mathbf{r}_{2\mathrm{D}})|^2 + (\gamma^2 x^2 + \lambda^2 z^2)|\Psi_{2\mathrm{D}}(\mathbf{r}_{2\mathrm{D}})|^2 + \frac{1}{2} g_{2\mathrm{D}} N_{\mathrm{at}} |\Psi_{2\mathrm{D}}(\mathbf{r}_{2\mathrm{D}})|^4 + \frac{1}{2} g_{\mathrm{dd},2\mathrm{D}} N_{\mathrm{at}} \int \frac{d\mathbf{k}_{2\mathrm{D}}}{(2\pi)^2} e^{-i\mathbf{k}_{2\mathrm{D}}\cdot\mathbf{r}_{2\mathrm{D}}} \tilde{n}_{2\mathrm{D}}(\mathbf{k}_{2\mathrm{D}}) j_{2\mathrm{D}}\left(\frac{k_{2\mathrm{D}}}{\sqrt{2\nu}}, k_z\right) |\Psi_{2\mathrm{D}}(\mathbf{r}_{2\mathrm{D}})|^2 \Bigg]. \tag{2.48}$$

The same applies to 1D systems described by GPE (2.16), where in the case when dipoles are orthogonal to the trap axis, the chemical potential and energy can be expresses by

$$\mu = \frac{1}{2}\int_{-\infty}^{\infty} dx \Bigg[\left|\frac{d\Psi_{1\mathrm{D}}(x)}{dx}\right|^2 + \gamma^2 x^2 |\Psi_{1\mathrm{D}}(x)|^2 + g_{1\mathrm{D}} N_{\mathrm{at}} |\Psi_{1\mathrm{D}}(x)|^4 + g_{\mathrm{dd},1\mathrm{D}} N_{\mathrm{at}} \int_{-\infty}^{\infty} \frac{dk_x}{2\pi} e^{-ik_x x} \tilde{n}_{1\mathrm{D}}(k_x) j_{1\mathrm{D}}\left(\frac{k_x}{\sqrt{2\nu}}\right) |\Psi_{1\mathrm{D}}(x)|^2 \Bigg], \tag{2.49}$$

$$E = \frac{1}{2}\int_{-\infty}^{\infty} dx \Bigg[\left|\frac{d\Psi_{1\mathrm{D}}(x)}{dx}\right|^2 + \gamma^2 x^2 |\Psi_{1\mathrm{D}}(x)|^2 + \frac{1}{2} g_{1\mathrm{D}} N_{\mathrm{at}} |\Psi_{1\mathrm{D}}(x)|^4 + \frac{1}{2} g_{\mathrm{dd},1\mathrm{D}} N_{\mathrm{at}} \int_{-\infty}^{\infty} \frac{dk_x}{2\pi} e^{-ik_x x} \tilde{n}_{1\mathrm{D}}(k_x) j_{1\mathrm{D}}\left(\frac{k_x}{\sqrt{2\nu}}\right) |\Psi_{1\mathrm{D}}(x)|^2 \Bigg], \tag{2.50}$$

while in the case when dipoles are parallel to the trap axis these quantities are defined by

$$\mu = \frac{1}{2}\int_{-\infty}^{\infty} dz \Bigg[\left|\frac{d\Psi_{1\mathrm{D}}(z)}{dz}\right|^2 + \lambda^2 z^2 |\Psi_{1\mathrm{D}}(z)|^2 + g_{1\mathrm{D}} N_{\mathrm{at}} |\Psi_{1\mathrm{D}}(z)|^4 + g_{\mathrm{dd},1\mathrm{D}} N_{\mathrm{at}} \int_{-\infty}^{\infty} \frac{dk_z}{2\pi} e^{-ik_z z} \tilde{n}_{1\mathrm{D}}(k_z) h_{1\mathrm{D}}\left(\frac{k_z}{\sqrt{2\gamma}}\right) |\Psi_{1\mathrm{D}}(z)|^2 \Bigg], \tag{2.51}$$

$$E = \frac{1}{2}\int_{-\infty}^{\infty} dz \Bigg[\left|\frac{d\Psi_{1\mathrm{D}}(z)}{dz}\right|^2 + \lambda^2 z^2 |\Psi_{1\mathrm{D}}(z)|^2 + \frac{1}{2} g_{1\mathrm{D}} N_{\mathrm{at}} |\Psi_{1\mathrm{D}}(z)|^4 + \frac{1}{2} g_{\mathrm{dd},1\mathrm{D}} N_{\mathrm{at}} \int_{-\infty}^{\infty} \frac{dk_z}{2\pi} e^{-ik_z z} \tilde{n}_{1\mathrm{D}}(k_z) h_{1\mathrm{D}}\left(\frac{k_z}{\sqrt{2\gamma}}\right) |\Psi_{1\mathrm{D}}(z)|^2 \Bigg]. \tag{2.52}$$

Note that both of the above quantities can be generalized and calculated also during real-time propagation, however only time-dependent energy is physically relevant.

Another set of useful quantities is related to the size of the system, which can be calculated for the ground state as well as during dynamical evolution of the system. Typical measure of system's size in a given direction is expectation value of the coordinate square, such as

$$\langle x^2 \rangle = \int d\mathbf{r}\, x^2 |\Psi(\mathbf{r};t)|^2 \,, \quad \langle y^2 \rangle = \int d\mathbf{r}\, y^2 |\Psi(\mathbf{r};t)|^2 \,, \quad \langle z^2 \rangle = \int d\mathbf{r}\, z^2 |\Psi(\mathbf{r};t)|^2 \,, \tag{2.53}$$

and the system size is then estimated as a *root-mean-square* (RMS) of the corresponding expectation value

$$x_{\mathrm{rms}} = \sqrt{\langle x^2 \rangle} \,, \quad y_{\mathrm{rms}} = \sqrt{\langle y^2 \rangle} \,, \quad z_{\mathrm{rms}} = \sqrt{\langle z^2 \rangle} \,. \tag{2.54}$$

The size of the whole system is usually estimated by the quadratic mean of coordinate RMS sizes,

$$r_{\mathrm{rms}} = \sqrt{\langle x^2 \rangle + \langle y^2 \rangle + \langle z^2 \rangle} \,. \tag{2.55}$$

Note that all RMS sizes calculated in this way are dimensionless and can be converted to physical units by multiplying them with the harmonic oscillator length l.

We can define analogous quantities for systems of reduced dimensionality where, for example, coordinate RMS sizes of 2D systems can be expressed by

$$\langle x^2 \rangle = \int d\mathbf{r}_{\mathrm{2D}}\, x^2 |\Psi_{\mathrm{2D}}(\mathbf{r}_{\mathrm{2D}};t)|^2 \,, \quad \langle y^2 \rangle = \int d\mathbf{r}_{\mathrm{2D}}\, y^2 |\Psi_{\mathrm{2D}}(\mathbf{r}_{\mathrm{2D}};t)|^2 \,, \tag{2.56}$$

while for 1D systems the RMS size is given by

$$\langle x^2 \rangle = \int\limits_{-\infty}^{\infty} dx\, x^2 |\Psi_{\mathrm{1D}}(x;t)|^2 \,. \tag{2.57}$$

Of course, the above equations depend on the 2D or 1D system geometry, i.e., on the coordinates used to describe the system.

Finally, in order to visualize the system, one can consider the appropriate density distribution. For 3D systems, the density is given by

$$n(\mathbf{r};t) = |\Psi(\mathbf{r};t)|^2 \,, \tag{2.58}$$

and analogously for 2D and 1D systems. We can also consider effective, lower-dimensional density projections, that can be more easily visualized. For example, for 3D systems we can define the effective 2D and 1D density by integrating out the full 3D density over some of the coordinates,

$$n_{\mathrm{2D}}(\mathbf{r}_{\mathrm{2D}};t) = \int_{-\infty}^{\infty} dz\, |\Psi(\mathbf{r};t)|^2 \,, \quad n_{\mathrm{1D}}(x;t) = \int d\mathbf{r}_{\mathrm{2D}}\, |\Psi(\mathbf{r};t)|^2 \,. \tag{2.59}$$

Similar density projections can be defined for 2D systems as well. All densities and density projections calculated in this way are dimensionless and can be converted to physical units by multiplying them with the appropriate unit (l^{-3} in 3D, l^{-2} in 2D and l^{-1} in 1D).

Chapter 3

Algorithm for shared memory systems

Numerical analysis from Chapter 2 outlines the algorithm we present in this chapter. Similarly to the numerical analysis, the algorithm is based on Refs. [12, 13, 16]. Here we present an improved serial and a new, parallelized version for shared memory computer systems.

Initial algorithm for solving the GPE with contact interaction term using Crank-Nicolson method was published in Ref. [12], which also includes a serial implementation in Fortran. This was followed by the serial and parallel implementations written in C [13]. An extension of the algorithm to include the dipolar interaction term was published in Ref. [16]. The authors provided two serial implementations, in C and Fortran. Their algorithm was used as a starting point for all algorithms and implementations presented in this thesis.

In Section 3.1 we will give an overview of the baseline algorithm for dipolar GPE [16] and discuss options for its parallelization on shared memory systems, followed by the description of the improved serial implementation in Section 3.2. Typical numerical simulations that make use of presented serial implementations of the algorithm are extremely computationally demanding, and therefore the parallelization approach was an obvious route forward, details of which are given in Section 3.3.

3.1 Description of the algorithm

In this section we give a brief description of the algorithm for the application of numerical methods presented in Chapter 2, which is instructive in understanding how to implement it on a shared memory computer.

We start with the description of the discretization scheme used by the algorithm. Let `Nx`, `Ny` and `Nz` be the number of points in each direction, corresponding to the x, y and z directions in one, two or three dimensions (1D, 2D and 3D, respectively). In other words, the number of points in 1D is defined by `Nx`, in 2D by `Nx` and `Ny`, and in 3D by `Nx`, `Ny` and `Nz`. Discretization points in each direction are equidistant, with their spacing defined in dimensionless units by variables `dx`, `dy` and `dz`, illustrated in Figure 3.1.

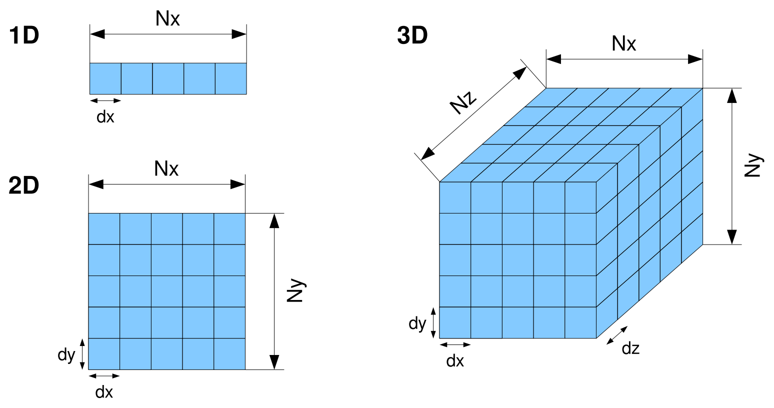


Figure 3.1: Example of 1D, 2D and 3D meshes with `Nx`, `Ny` and `Nz` discretization points in x, y and z direction, respectively, and the corresponding spacings `dx`, `dy` and `dz`.

The mesh determines how the space is discretized, but the actual values of the wave function, which is also discretized by the mesh, need to be stored in a separate array. For convenience, we refer to this array as `psi` variable. This variable is defined as a vector of size `Nx` in 1D, as `Nx` $\times$ `Ny` matrix in 2D and as `Nx` $\times$ `Ny` $\times$ `Nz` tensor in 3D. In real-time propagation, the wave function values are complex, while in imaginary-time propagation they are purely real, which determines the type of `psi` variable.

Time is discretized according to $t_n = t_0 + n\Delta$, with the time step Δ stored in a variable `dt`. Taking into account the time step and the total amount of physical time we wish to simulate, we determine the number of iterations required (n). The main part of the algorithm is a loop where each iteration corresponds to one time step of the propagation. Inside the loop, we update the wave function by propagating it in time with respect to different parts of the Hamiltonian, according to the split-step scheme described in Chapter 2. This is illustrated in Figure 3.2.

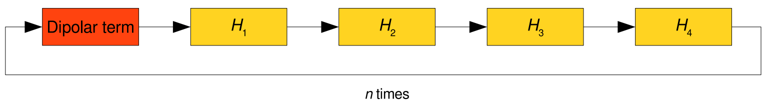


Figure 3.2: Main loop of the real-time propagation algorithm in 3D.

In each step of the main loop, wave function values are computed along the discretization mesh in several smaller steps. First such substep is the calculation of the dipolar interaction term. We have two further substeps in 1D, pertaining to Eq. (2.22) w.r.t. H_1 and Eq. (2.23) w.r.t. H_2. In 2D and 3D we follow a similar procedure and add further substeps for H_3 and H_4. Each substep produces intermediate values of the wave function that are used in the next substep. The order of the substeps dealing with parts of the Hamiltonian with spatial derivatives (H_2, H_3 and H_4) can be arbitrary, a feature useful in hybrid and distributed memory algorithms. Propagation of the wave function w.r.t. H_1 relies on the availability of the dipolar interaction term value, which is computed in the first substep, before we start with the update of the wave function.

The dipolar interaction term is computed according to Eq. (2.37), using the discrete Fourier transform (DFT). DFT of a sequence is commonly implemented by relying on the *Fast Fourier transform* (FFT) algorithm via an external library. From Eq. (2.37) it follows that we first need a

DFT of a sequence consisting of absolute squares of wave function values. The resulting complex-valued sequence is then multiplied by the Fourier transform of the dipolar potential. This transform is a known function (2.38), precomputed for a chosen spatial mesh and stored in an array. Next, by performing the inverse DFT we get the dipolar interaction term stored in the resulting array, which we then use to propagate the wave function w.r.t. H_1. In 2D and 3D, we need multidimensional DFT, which can be computed by the composition of a sequence of 1D DFTs along each direction.

The substep in which the wave function is propagated w.r.t. the Hamiltonian part without spatial derivatives (H_1) proceeds according to Eq. (2.24), by employing further nested loops over mesh points. Part of the H_1 is the trap potential $V(\mathbf{r})$, which, like Fourier transform of the dipolar potential, can be computed only once during the initialization, and then reused. We assume that the trap potential is static in all algorithms described in this thesis. In case of a time-varying trap potential, the algorithm can be modified to update the trap potential before this substep, i.e., at the beginning of each iteration.

The remaining substep, or substeps in 2D and 3D, propagates the wave function by relying on the Crank-Nicolson scheme. We need to perform a backward sweep of the mesh in each direction to determine corresponding coefficients α, β and γ for each direction, according to Eqs. (2.32)–(2.34), followed by a forward sweep to determine the solution for the entire space range. Since the coefficients α and γ for each direction are independent of the current state of the wave function, they can be computed beforehand, during initialization.

The algorithm for imaginary-time propagation is similar to the one discussed above, with the addition of the normalization step at the end of each iteration of the main loop, as illustrated in Figure 3.3. To normalize the wave function, we first need to compute its norm, according to Eq. (2.3), and then to divide the wave function values with that norm.

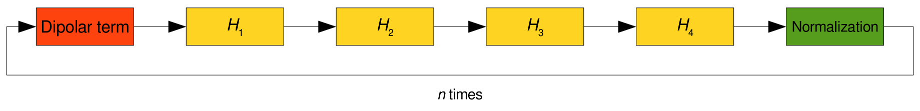


Figure 3.3: Main loop of the imaginary-time propagation algorithm in 3D, which introduces an additional step of normalization.

In Section 2.4, we introduced other physical quantities of interest that are computed from the wave function, namely chemical potential, energy and the RMS of system's physical extent (size), which give us more insight into the state of the simulation and allows us to observe and measure time evolution of a BEC. These quantities do not have to be computed in each iteration. Instead, they can be calculated periodically after a predefined number of iterations.

We see that the algorithm is computationally very demanding, as it requires multiple passes over the entire mesh in each iteration of the main loop. In 2D and 3D, the total number of discretized points sharply increases, making this problem even greater. As we have identified above, several variables can be moved out of the main loop and computed only once, however the remaining calculations are still very demanding, hence the need for parallelization.

The described substeps cannot be executed concurrently, as the intermediate values they produce are used in subsequent substeps. Instead, parallelization of the algorithm can only be achieved by focusing on the mesh loops inside substeps. This is an obvious choice for shared memory systems, as all data stored in the memory is local and accessible to all participating processes, removing the need for data transfers between processes. The loops themselves can easily be parallelized if they do not contain recursive relations. This is true for the substeps dealing with the computation of the dipolar term and propagation w.r.t. H_1, as well as periodical computation of chemical potential, energy and RMS of BEC's size. Substep involving H_2 cannot be easily parallelized in 1D, as it has recursive relations in both backward and forward sweeps. In general, recursive relations can be parallelized using the *scan* algorithm [34] (also known as the generalization of *prefix sum* algorithm [35]), however the implementation complexity of such an algorithm made us discard this approach. In 2D and 3D, we can exploit the fact that the recursive relations appear only in the innermost loops of the substeps involving H_2, H_3 and H_4. Thus, we can achieve parallelization by dividing the work among the processes at the level of the outermost loop.

Given the algorithm and parallelization strategy, in next two sections we present serial and parallel implementations.

3.2 Serial implementation

Basis for the programs presented in this section was originally developed in Fortran and published in 2009 in Ref. [12], however without considering the dipolar interaction term. Fortran and C implementations that take into account both contact and dipolar interaction term have been published in 2015 in Ref. [16] as package *DBEC-GP*, which can be downloaded from the journal repository [36] or alternatively from the local repository [37]. This package consists of 10 Fortran 90/95 programs, as well as 10 corresponding C programs. For each programming language, there is one program for solving full GPE in 3D, two programs for solving the effective 2D GPE and two for solving the effective 1D GPE, implemented separately for imaginary- and real-time propagation. Here we present improved versions of the latter C programs. Note that these programs may not be suitable for long-running simulations, due to their serial nature. They are superseded by all other programs presented in this thesis, which vastly out-perform programs presented in this section. Another issue with serial and other shared memory programs is their inherent limitation on mesh sizes due to memory constraints of a single computing node. This will be addressed in Chapter 6.

Implementation-wise, the main difference between real- and imaginary-time programs is that the wave function is complex in real-time propagation, and real in imaginary-time. Real values are stored in double precision, as single precision type is inadequate for wave function values since the lower precision in computation leads to faster accumulation of numerical errors. In C-based implementations, complex values are stored in `double complex` type, which is part of the C99 standard [38] of the C language. The implication of using complex numbers is that real-time propagation is more demanding both in terms of memory and computation. However, they have a different purpose: imaginary-time versions of programs are used for calculation of the stationary (ground) state of a physical system, which is then typically used as the initial value of the wave function for real-time propagation.

All programs share the similar structure and reuse common routines whenever possible. Here we present the general flow of implementation, with examples from 1D, 2D and 3D programs.

Before the main time-propagation loop can begin, we need to allocate space for the variables, and set them to their initial values. Matrices and tensors used in 2D and 3D programs are allocated as contiguous arrays of required type (real or complex), with a pointer scheme that allows access to elements in 2D or 3D, see Listing 3.1. This allows us to access elements of matrices/tensors in a more intuitive way, e.g., `psi[i][j][k]`, as opposed to computing the 1D index every time (for example, `psi[i * Ny * Nz + j * Nz + k]`). The underlying flat allocation scheme is not required, but has turned out to be very useful during the implementation of GPGPU and heterogeneous programs, described in Chapters 4 and 5.

Listing 3.1: Allocation of a real-valued vector, matrix and tensor. Allocation of complex-valued arrays is analogous.

```c
// Double vector allocation
double *alloc_double_vector(long Nx) {
   double *vector;

   if((vector = (double *) malloc((size_t) (Nx * sizeof(double)))) == NULL) {
      fprintf(stderr, "Failed to allocate memory for the vector.\n"); exit(EXIT_FAILURE);
   }
   return vector;
}
// Double matrix allocation
double **alloc_double_matrix(long Nx, long Ny) {
   long i;
   double **matrix;

   if((matrix = (double **) malloc((size_t) (Nx * sizeof(double *)))) == NULL) {
      fprintf(stderr, "Failed to allocate memory for the matrix.\n"); exit(EXIT_FAILURE);
   }
   if((matrix[0] = (double *) malloc((size_t) (Nx * Ny * sizeof(double)))) == NULL) {
      fprintf(stderr, "Failed to allocate memory for the matrix.\n"); exit(EXIT_FAILURE);
   }
   for(i = 1; i < Nx; i ++)
      matrix[i] = matrix[i - 1] + Ny;

   return matrix;
}
// Double tensor allocation
double ***alloc_double_tensor(long Nx, long Ny, long Nz) {
   long i, j;
   double ***tensor;

   if((tensor = (double ***) malloc((size_t) (Nx * sizeof(double **)))) == NULL) {
      fprintf(stderr, "Failed to allocate memory for the tensor.\n"); exit(EXIT_FAILURE);
   }
   if((tensor[0] = (double **) malloc((size_t) (Nx * Ny * sizeof(double *)))) == NULL) {
      fprintf(stderr, "Failed to allocate memory for the tensor.\n"); exit(EXIT_FAILURE);
   }
   if((tensor[0][0] = (double *) malloc((size_t)(Nx * Ny * Nz * sizeof(double)))) == NULL){
      fprintf(stderr, "Failed to allocate memory for the tensor.\n"); exit(EXIT_FAILURE);
   }
   for(j = 1; j < Ny; j ++)
      tensor[0][j] = tensor[0][j-1] + Nz;
   for(i = 1; i < Nx; i ++) {
      tensor[i] = tensor[i - 1] + Ny;
      tensor[i][0] = tensor[i - 1][0] + Ny * Nz;
      for(j = 1; j < Ny; j ++)
         tensor[i][j] = tensor[i][j - 1] + Nz;
   }

   return tensor;
}
```

Propagation of the wave function w.r.t. different parts of the Hamiltonian has been implemented in separate functions, where function `calcnu` takes into account H_1 part of the Hamiltonian, `calclux` propagates the wave function w.r.t. H_2, `calcluy` corresponds to propagation determined by H_3 (in 2D and 3D programs) and `calcluz` does the same for H_4 (in 3D programs). Calculation of the dipolar interaction term, upon which function `calcnu` relies, has been implemented separately, in function `calcpsidd2`. Afterwards, the value of the wave function is propagated for each mesh point, as shown in Listing 3.2.

Listing 3.2: Loops are employed to propagate the wave function in each mesh point, as illustrated here on a 1D real-time `calcnu` function.

```
void calcnu(double complex *psi, double *psidd2) {
   long i;
   double psi2, psi2lin, psidd2lin, tmp;

   for(i = 0; i < Nx; i ++) {
      psi2 = cabs(psi[i]);
      psi2 *= psi2;
      psi2lin = psi2 * g;
      psidd2lin = psidd2[i] * gd;
      tmp = dt * (pot[i] + psi2lin + psidd2lin);
      psi[i] *= cexp(- I * tmp);
   }
}
```

The function `calcpsidd2` computes the DFT of the input sequence by means of the FFT algorithm. The authors in Ref. [16] relied on the FFTW library [39] for this task, a well-known and widely used FFT library. In their implementation they have used the ordinary DFT of complex data, which takes a complex-valued input and produces a complex-valued output of identical size, a so-called *C2C* transform. This can be improved by observing the fact that in Eq. (2.37) the input of the transform is $|\Psi(\mathbf{r})|^2$, which is purely real, even when $\Psi(\mathbf{r})$ is complex-valued. DFT of a real-valued input produces an output that has Hermitian symmetry, with one half of the output being the complex conjugate of the other half, making it redundant. Many FFT libraries, FFTW included, are able to exploit this symmetry to provide a special real-to-complex (*R2C*) transform. R2C transform is both faster, due to fewer operations involved, and requires less memory, because only half of the output is retained. To use R2C instead of C2C transforms, we have to ensure that the size of output array is sufficient (e.g., `Nx * Ny * (Nz / 2 + 1) * sizeof(double complex)` in 3D), and use the appropriate R2C functions from FFTW (see Listing 3.3). Performing any type of transform with FFTW involves creation and execution of a *plan*, which specifies all details of the transform.

Listing 3.3: Real-to-complex and complex-to-real FFTW calls used in `calcpsidd2` function. Shown here is the 1D variant of the function, used in real-time propagation.

```
// Here we use in-place FFT, so we need to ensure proper padding of input and output arrays
psidd2 = alloc_double_vector((Nx / 2 + 1) * 2);
psidd2fft = (fftw_complex *) psidd2;

// Create plans during initialization
plan_forward = fftw_plan_dft_r2c_1d(Nx, psidd2, psidd2fft, FFTW_MEASURE);
plan_backward = fftw_plan_dft_c2r_1d(Nx, psidd2fft, psidd2, FFTW_MEASURE);
```

```
void calcpsidd2(double complex *psi, double *psidd2, fftw_complex *psidd2fft) {
   long i;
   double tmp;
   // Prepare input for FFT
   for(i = 0; i < Nx; i ++) {
      tmp = cabs(psi[i]);
      psidd2[i] = tmp * tmp;
   }
   // Execute R2C transform
   fftw_execute_dft_r2c(plan_forward, psidd2, psidd2fft);
   // Resulting output array has Nx / 2 + 1 elements
   for(i = 0; i < Nx / 2 + 1; i ++) {
      psidd2fft[i][0] *= potdd[i];
      psidd2fft[i][1] *= potdd[i];
   }
   // Execute C2R transform
   fftw_execute_dft_c2r(plan_backward, psidd2fft, psidd2);
   // Rescale the result of FFT
   for(i = 0; i < Nx; i ++) {
      psidd2[i] /= Nx;
   }
   // Handle boundary case
   psidd2[Nx - 1] = psidd2[0];
}
```

Functions `calclux`, `calcluy` and `calcluz` perform the time propagation with respect to parts of the Hamiltonian involving spatial partial derivatives by solving the tridiagonal system that arises from Crank-Nicolson method. Implementation of forward recursion and backward substitution is shown in Listing 3.4.

Listing 3.4: 3D real-time functions `calclux`, `calcluy` and `calcluz`.

```
// Time propagation with respect to H2 (x-part of the Laplacian).
void calclux(double complex ***psi, double complex *cbeta) {
   long i, j, k;
   double complex c;

   for(j = 0; j < Ny; j ++) {
      for(k = 0; k < Nz; k ++) {
         cbeta[Nx - 2] = psi[Nx - 1][j][k];
         for(i = Nx - 2; i > 0; i --) {
            c = - Ax * psi[i + 1][j][k] + Ax0r * psi[i][j][k] - Ax * psi[i - 1][j][k];
            cbeta[i - 1] = cgammax[i] * (Ax * cbeta[i] - c);
         }
         psi[0][j][k] = 0.;
         for(i = 0; i < Nx - 2; i ++) {
            psi[i + 1][j][k] = calphax[i] * psi[i][j][k] + cbeta[i];
         }
         psi[Nx - 1][j][k] = 0.;
      }
   }
}
// Time propagation with respect to H3 (y-part of the Laplacian).
void calcluy(double complex ***psi, double complex *cbeta) {
   long i, j, k;
   double complex c;

   for(i = 0; i < Nx; i ++) {
      for(k = 0; k < Nz; k ++) {
         cbeta[Ny - 2] = psi[i][Ny - 1][k];
         for(j = Ny - 2; j > 0; j --) {
            c = - Ay * psi[i][j + 1][k] + Ay0r * psi[i][j][k] - Ay * psi[i][j - 1][k];
            cbeta[j - 1] = cgammay[j] * (Ay * cbeta[j] - c);
         }
         psi[i][0][k] = 0.;
         for(j = 0; j < Ny - 2; j ++) {
            psi[i][j + 1][k] = calphay[j] * psi[i][j][k] + cbeta[j];
         }
         psi[i][Ny - 1][k] = 0.;
      }
   }
}
// Time propagation with respect to H4 (z-part of the Laplacian).
void calcluz(double complex ***psi, double complex *cbeta) {
   long i, j, k;
   double complex c;

   for(i = 0; i < Nx; i ++) {
      for(j = 0; j < Ny; j ++) {
         cbeta[Nz - 2] = psi[i][j][Nz - 1];
         for(k = Nz - 2; k > 0; k --) {
            c = - Az * psi[i][j][k + 1] + Az0r * psi[i][j][k] - Az * psi[i][j][k - 1];
            cbeta[k - 1] = cgammaz[k] * (Az * cbeta[k] - c);
         }
         psi[i][j][0] = 0.;
         for(k = 0; k < Nz - 2; k ++) {
            psi[i][j][k + 1] = calphaz[k] * psi[i][j][k] + cbeta[k];
         }
         psi[i][j][Nz - 1] = 0.;
      }
   }
}
```

In Listing 3.4 we observe the pattern employed in higher-dimensional programs. Depending on the spatial direction of the computation we perform, i.e., the function used, nested loops are reordered appropriately so that the innermost one corresponds to the computation direction. This pattern is used in other functions whenever we need to access elements in a matrix/tensor in a specific spatial direction.

In imaginary-time propagation, the algorithm mandates that at the end of every time step an additional substep, normalization of the wave function, is required. To compute the wave function norm, we have to integrate over all space, according to Eq. (2.3). Numerical integration is implemented in function `simpint` using composite Simpson's rule,

$$\int_a^b dx\, f(x) \approx \frac{h}{3} \sum_{i=1}^{n/2} \Big[f(x_{2i-2}) + 4f(x_{2i-1}) + f(x_{2i}) \Big] , \tag{3.1}$$

where n is the number of subintervals, step size h is $h = (b-a)/n$ and $x_i = a + ih$. Computing the norm and subsequent normalization of the wave function is illustrated in Listing 3.5. Note that in real-time propagation the norm of the wave function is preserved, and therefore we do not perform the normalization at the end of every time step. However, keeping track of the norm is still useful as a check of correctness, and we calculate it at the regular intervals. The implementation remains mostly the same, with the last step (performing the normalization of the wave function) omitted in real-time propagation.

Listing 3.5: Numerical integration via composite Simpson's rule and its usage in computation of the norm and normalization of the wave function, as used by the 1D imaginary-time programs. The equivalent real-time function is similar, with the last step removed.

```
// Numerical integration via composite Simpson's rule
double simpint(double h, double *f, long N) {
   int c;
   long i;
   double sum;

   sum = f[0];
   for (i = 1; i < N - 1; i ++) {
      c = 2 + 2 * (i % 2);
      sum += c * f[i];
   }
   sum += f[N - 1];

   return sum * h / 3.;
}

void calcnorm(double *norm, double *psi, double *tmpx) {
   long i;
   double tmp;

   for(i = 0; i < Nx; i ++) {
      tmpx[i] = cabs(psi[i]);
      tmpx[i] *= tmpx[i];
   }

   *norm = sqrt(simpint(dx, tmpx, Nx));
   tmp = 1. / *norm;

   for(i = 0; i < Nx; i ++) {
      psi[i] *= tmp;
   }
}
```

The chemical potential and energy are computed in function `calcmuen`, while RMS size is calculated in function `calcrms`. Chemical potential and energy are computed in the same function because Eqs. (2.43) and (2.44) used to compute these quantities differ only in a coefficient in front of the nonlinear term. Computation of these quantities requires calculation of spatial derivatives of the wave function, which is implemented in function `diff` using Richardson extrapolation formula of the fourth order [40],

$$f'(x) \approx \frac{f(x-2h) - 8f(x-h) + 8f(x+h) - f(x-2h)}{12h}\,. \tag{3.2}$$

Finally, expressions for the chemical potential and energy are integrated using composite Simpson's rule. The implementation of `calcmuen` with the aforementioned calculations in 1d case is illustrated in Listing 3.6.

Listing 3.6: Calculation of chemical potential and energy in 1D imaginary-time `calcmuen` function. Also shown is the numerical differentiation function, based on Richardson extrapolation formula.

```
void calcmuen(double *mu, double *en, double *psi, double *dpsix, double *psidd2, double
    *tmpxi, double *tmpxj) {
   long i;
   double psi2, psi2lin, psidd2lin, dpsi2;
   diff(dx, psi, dpsix, Nx);
   calcpsidd2(psi, psidd2);

   for(i = 0; i < Nx; i ++) {
      psi2 = psi[i] * psi[i];
      psi2lin = psi2 * g;
      psidd2lin = psidd2[i] * gd;
      dpsi2 = dpsix[i] * dpsix[i];
      tmpxi[i] = (pot[i] + psi2lin + psidd2lin) * psi2 + dpsi2;
      tmpxj[i] = (pot[i] + 0.5 * psi2lin + 0.5 * psidd2lin) * psi2 + dpsi2;
   }
   *mu = simpint(dx, tmpxi, Nx);
   *en = simpint(dx, tmpxj, Nx);
}
// Calculation of numerical derivative using Richardson extrapolation formula
void diff(double h, double *f, double *df, long N) {
   long i;

   df[0] = 0.;
   df[1] = (f[2] - f[0]) / (2. * h);

   for(i = 2; i < N - 2; i ++) {
      df[i] = (f[i - 2] - 8. * f[i - 1] + 8. * f[i + 1] - f[i + 2]) / (12. * h);
   }

   df[N - 2] = (f[N - 1] - f[N - 3]) / (2. * h);
   df[N - 1] = 0.;
}
```

In 2D and 3D, partial derivatives need to be calculated for each direction. Previously published implementations [13, 16] allocate separate arrays for each partial derivative, that can consume significant amount of memory if the mesh size is large. By reorganizing the computation of derivatives slightly, we can reuse a single array for all calculations involving partial derivatives. This small change leads to significant reduction in memory usage, especially in 3D programs. For this reason we have relied on this type of memory usage optimizations for all programs presented in this thesis.

Calculation of RMS sizes in function `calcrms` is straightforward, given the simplicity of their definitions (2.53)–(2.57). Relevant portion of the implementation in 1D is shown in Listing 3.7. Implementation in 2D and 3D relies on the concepts we already presented, hence no further details will be given.

Listing 3.7: Computation of the RMS size in 1D in imaginary-time propagation.

```
void calcrms(double *rms, double complex *psi, double *tmpx) {
   long i;
   double psi2;

   for(i = 0; i < Nx; i ++) {
      psi2 = cabs(psi[i]);
      psi2 *= psi2;
      tmpx[i] = x2[i] * psi2;
   }
   *rms = sqrt(simpint(dx, tmpx, Nx));
}
```

In order to use the programs to simulate a given physical system, we need to provide the appropriate input parameters. The parameters of the programs, e.g., the discretization scheme, number of time steps, trap anisotropy coefficients, coefficients of nonlinear terms, interaction strengths, names of output files and others, are specified in the input file. Input files are passed to the programs as runtime arguments, where they are parsed, and the variables obtained in this way are used to initialize the space mesh, the wave function, trap and dipolar potentials, and Crank-Nicolson coefficients. Listing 3.8 shows how initialization of space mesh and associated wave function, trap potential and Crank-Nicolson coefficients is implemented in 1D. Initialization of the dipolar potential, which is not listed here, has a slightly longer implementation and is available in Ref. [16]. In 2D and 3D this step is performed in an analogous way, and is also not given here.

Listing 3.8: Initialization of space mesh and associated wave function, trap potential and Crank-Nicolson coefficients in 1D real-time programs.

```c
// Initialization of the space mesh and the initial wave function.
void initpsi(double complex *psi) {
   long i;
   double cpsi = sqrt(sqrt(pi / vgamma));

   for(i = 0; i < Nx; i ++) {
      x[i] = (i - Nx2) * dx; x2[i] = x[i] * x[i];
   }
   for(i = 0; i < Nx; i ++) {
      psi[i] = exp(- 0.5 * vgamma * x2[i]) / cpsi;
   }
}
// Initialization of the potential.
void initpot() {
   long i;
   double vgamma2 = vgamma * vgamma;

   for(i = 0; i < Nx; i ++) {
      pot[i] = 0.5 * par * (vgamma2 * x2[i]);
   }
}
// Crank-Nicolson scheme coefficients generation.
void gencoef() {
   long i;

   Ax0 = 1. + I * dt / dx2;
   Ax0r = 1. - I * dt / dx2;
   Ax = - 0.5 * I * dt / dx2;

   calphax[Nx - 2] = 0.;
   cgammax[Nx - 2] = - 1. / Ax0;
   for (i = Nx - 2; i > 0; i --) {
      calphax[i - 1] = Ax * cgammax[i];
      cgammax[i - 1] = - 1. / (Ax0 + Ax * calphax[i - 1]);
   }
}
```

With all the relevant variables allocated and initialized to the proper values, the time propagation can now proceed. This is achieved by calling the previously described computation functions in the main loop, as illustrated in Listing 3.9. Note that in the imaginary-time propagation, as mentioned earlier, the normalization has to be performed at the end of every time step. In the current implementation, the number of time iterations is predefined in the input file.

Listing 3.9: Main loop of the time propagation, for 3D imaginary-time programs.

```
//Read input, allocate and initialize variables
...
// Main loop of time propagation
for(i = 0; i < Nrun; i ++) {
   calcpsidd2(psi, psidd2, psidd2fft);
   calcnu(psi, psidd2);
   calclux(psi, cbeta);
   calcluy(psi, cbeta);
   calcluz(psi, cbeta);
   // In imaginary-time propagation, we need additional normalization step
   calcnorm(&norm, psi, tmpxi, tmpyi, tmpzi);
   // Optionally, physical quantities can be computed after certain number of steps
   if (i % 1000 == 0) {
      ... // Call 'calcmuen', 'calcrms' or output functions
   }
}
// Write summary file, clean up and exit
...
```

After the predefined number of time steps, programs can compute physical quantities like chemical potential or energy, or they can write to a file the current density profile (absolute square of the wave function), which can be used for further analysis and/or visualization. Final density profile of the imaginary-time propagation can be used as the initial value for wave function propagation in real time. 2D and 3D programs also provide integrated (effective) 1D density profiles, obtained by integrating the densities over all spatial variables but one. Furthermore, 3D programs can write integrated 2D density profiles, obtained in a similar way. Output mesh can coincide with the mesh used by the program, or can be coarser, controlled by the `outstpx`, `outstpy` and `outstpz` input parameters. Two examples of such output functions are given in Listing 3.10.

Listing 3.10: Examples of output functions from 2D real-time programs that compute and write density and integrated density to a file.

```
void outpsi2xy(double complex **psi, FILE *file) {
   long i, j;

   for(i = 0; i < Nx; i += outstpx) {
      for(j = 0; j < Ny; j += outstpy) {
         fprintf(file, "%8le %8le %8le\n", x[i], y[j], cabs(psi[i][j]) * cabs(psi[i][j]));
      }
      fprintf(file, "\n"); fflush(file);
   }
}

void outdenx(double complex **psi, double *tmp, FILE *file) {
   long i, j;

   for(i = 0; i < Nx; i += outstpx) {
      for(j = 0; j < Ny; j ++) {
           tmp[j] = cabs(psi[i][j]) * cabs(psi[i][j]);
      }
      fprintf(file, "%8le %8le\n", x[i], simpint(dy, tmp, Ny)); fflush(file);
   }
}
```

Additionally, all programs write the summary output file, which contains the information on the input parameters, and the computed norm, chemical potential and energy. A sample summary file is given in Listing 3.11.

Listing 3.11: Example of a summary output file of 3D real-time propagation.

```
 Real time propagation 3D, OPTION = 2

  Number of Atoms N = 5000, Unit of length AHO = 0.00000100 m
  Scattering length a = 100.00*a0, Dipolar ADD = 132.70*a0
  Nonlinearity G_3D = 332.4918, Strength of DDI GD_3D = 105.33272
  Parameters of trap: GAMMA = 0.50, NU = 1.00, LAMBDA = 1.50

 # Space Stp: NX = 128, NY = 96, NZ = 80
  Space Step: DX = 0.200000, DY = 0.200000, DZ = 0.200000
 # Time Stp : NSTP = 0, NPAS = 100, NRUN = 900
   Time Step:  DT = 0.005000
   Dipolar Cut off: R = 10.000

 * Change for dynamics: GPAR = 1.500, GDPAR = 1.000 *

                ---------------------------------------------------------
                  Norm     Chem       Ener/N     <r>       |Psi(0,0,0)|^2
                ---------------------------------------------------------
Initial:          1.0000   5.81621    4.33909    2.54499   0.02162
After NPAS iter.: 1.0000   5.81920    4.33911    2.54358   0.02181
After NRUN iter.: 1.0000   6.93923    5.00916    2.63297   0.01448
                ---------------------------------------------------------

 Clock Time: 290 seconds
 CPU Time: 290 seconds
```

The 10 improved programs developed here use the same naming convention as the original C programs. They can be downloaded from the local repository [37].

3.3 Parallelization on shared memory systems

It can be observed that the implementation described in the previous section is computationally very demanding, especially in 2D and 3D. This results in long execution time when a large discretization scheme is used, which makes the programs less appealing for fine-grained simulations, thus making parallelization a necessity. Parallel implementation on shared memory systems using C language can be achieved in multiple ways, using a wide array of technologies. These range from automatic parallelizations built in compilers [41] to specialized language extensions like Unified Parallel C [42, 43] and Cilk Plus [44]. Given that the parallelization efforts of previous work in Ref. [13] was focused on using OpenMP, which was shown to achieve 80-90% of the ideal speedup, we adopted the same approach.

OpenMP is designed for shared memory systems containing multiple processors/cores. It provides an explicit programming model, based around the concept of using multiple threads to accomplish parallelism. OpenMP is published as an open specification [45], and there are numerous compilers which implement support for it. Usually, a special flag is passed to the compiler (`-fopenmp` in GCC or `-qopenmp` in newer versions of Intel's compiler) to instruct it to parallelize the code using OpenMP.

Parallelization in OpenMP is achieved using the *fork-join* model, where *master* thread spawns (or *forks*) a number of *slave* threads to perform a task in parallel, at the end of which the slave threads *join* back to the master thread. It relies on compiler directives (called *pragmas*) and runtime library routines to achieve this task. Programmers declare parallel regions of the code by using compiler directives. When a master thread encounters such regions, it creates a team of parallel threads that execute the statements in the region. When threads complete all statements in the parallel region, they synchronize and terminate, leaving only the master thread to continue with the serial execution of the program. This flow is illustrated in Figure 3.4. The number of threads can be different for any parallel region, however in most cases the number of threads is equal to the number of processing cores available in the system. Such is the case with the programs presented here.

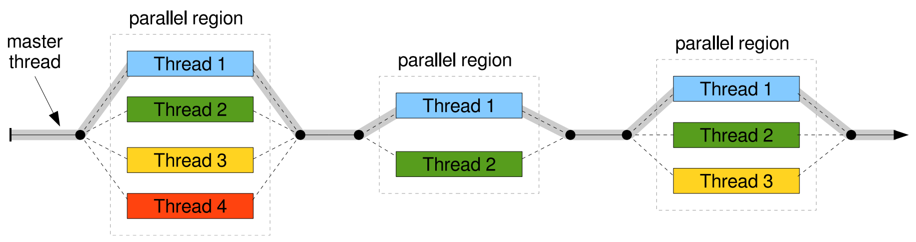


Figure 3.4: The fork-join model, used by OpenMP.

To create parallel regions, we must declare them with the compiler directives. Each OpenMP directive in C begins with the keyword `#pragma omp`, followed by a directive name and its optional clauses, and ends with the new line. When dealing with loops, the declaration of a parallel region can be shortened, as demonstrated in Listing 3.12. While the second approach results in a cleaner code that is easier to follow, the first one provides additional flexibility. We use both approaches in our programs, depending on whether we need to perform any additional work or just parallelize the loop at hand. Special care must be taken to ensure that each thread has a copy of intermediate variables it writes, defined in the surrounding parallel region. Otherwise, the compiler would assume that variable is shared among threads, and each thread would write its own value, overwriting the data of other threads and leading to concurrency problems.

Listing 3.12: OpenMP compiler directives.

```c
#include<omp.h> // Include the appropriate OpenMP header
...
#pragma omp parallel // Spawns a team of parallel threads
{
   #pragma omp for private(i) // Divides loop iterations between the spawned threads
   for(i = 0; i < Nx; i ++) {
      tmpx[i] = psi[i] * psi[i];
   }
}

#pragma omp parallel for private(i) // Shorter, combined parallel worksharing construct
for(i = 0; i < Nx; i ++) {
   tmpx[i] = psi[i] * psi[i];
}
```

Our programs rely on OpenMP to parallelize the most computationally demanding functions, namely `calcnu`, `calclux`, `calcluy`, `calcluz`, `calcpsidd2`, `calcnorm`, `calcmuen` and `calcrms`. Unfortunately, OpenMP currently does not provide any support for parallel input and output (I/O), leaving this task entirely to the programmer. Performing efficient read/write operations on a single file in multi-threaded environment is a complex task, and because of this no effort was made to improve the I/O functions in our programs, and they remain serial.

Since most of the computation in our programs is done in a series of loops, the parallelization is achieved by assigning the portion of each loop to a different thread. In 1D, that approach would take the form presented in Listing 3.13.

Listing 3.13: OpenMP parallelization of 1D loop, demonstrated on real-time `calcrms` function.

```
void calcrms(double *rms, double complex *psi, double *tmpx) {
   long i;
   double psi2;

   #pragma omp parallel for private(i)
   for(i = 0; i < Nx; i ++) {
      psi2 = cabs(psi[i]);
      psi2 *= psi2;
      tmpx[i] = x2[i] * psi2;
   }
   *rms = sqrt(simpint_th(dx, tmpx, Nx));
}
```

However, we can observe that this technique would not always work in 1D programs, because of the recursive relations in `calclux` function. The first loop requires that `cbeta[i]` is computed before `cbeta[i - 1]`, and similarly with the `psi` in the second loop. The *scan* algorithm could be used to parallelize the loops of this function, however improvements over the serial implementation would be visible only for simulations with very large number of discretization points (over a million), which are rarely needed. In multidimensional programs this approach would be even less effective, as the outer loops are independent and are better suited for parallelization than the innermost loop with recursive relations. Since the serial implementation in 1D can be considered to be sufficiently fast on modern hardware, and due to the complexity that parallelization of `calclux` would entail, the focus here is placed on improving 2D and 3D variants of the programs. 1D programs remain only partially parallelized, with all functions except for `calclux` being threaded.

In multidimensional programs, the outer loops are independent and their work can be divided between threads. Listing 3.14 illustrates how the loop of 2D real-time `calcnu` function is parallelized with OpenMP. The OpenMP runtime would divide the work in such a way that each thread would process approximately `Nx / nthreads` elements (with `nthreads` being the number of threads in the parallel region), where each element contains the whole inner loop. During computation, values of several expressions are stored in temporary variables. As mentioned before, each thread must have a private copy of these variables to avoid overwriting data, so we must specify them in the parallel construct. This concept is used in the parallelization of `calcnu`, `calclux`, `calcluy`, `calcluz` functions and portions of the `calcpsidd2` function.

When working with dynamically allocated arrays, for example when writing temporary results to an array, the variables cannot be made private just by declaring them in the parallel region. Instead, we have to extend the allocated array so that each thread has a separate portion. For instance, instead of allocating a vector of `Nx` elements, we allocate a `nthreads` $\times$ `Nx` matrix, and use each row in a different thread. Even though this concept increases the amount of memory programs use, that increase is not significant as the number of threads is relatively low compared to typical

Listing 3.14: OpenMP parallelization of 2D loop, demonstrated on real-time `calcnu` function.

```
void calcnu(double complex **psi, double **psidd2) {
   long i, j;
   double psi2, psi2lin, psidd2lin, tmp;

   #pragma omp parallel for private(i, j, psi2, psi2lin, psidd2lin, tmp)
   for (i = 0; i < Nx; i ++) {
      for (j = 0; j < Ny; j ++) {
         psi2 = cabs(psi[i][j]);
         psi2 *= psi2;
         psi2lin = psi2 * g;
         psidd2lin = psidd2[i][j] * gd;
         tmp = dt * (pot[i][j] + psi2lin + psidd2lin);
         psi[i][j] *= cexp(- I * tmp);
      }
   }
}
```

vector sizes. An implementation of this concept is shown in Listing 3.15, which demonstrates how `calcnorm` function can be parallelized in 2D. The threads fill out their portion of the `tmpy` temporary array, and perform the integration on their portion of the array. The listing also shows how a single thread in a parallel region can be used to compute the final result. As the variable holding the final result is not declared to be private, its value is visible to all threads, and can be used in the parallel region that follows. Other than `calcnorm` function, parallelization is achieved in this way in `calcmuen` and `calcrms`, so we would not go into their specifics here.

Listing 3.15: Concept used in parallelization of functions requiring temporary storage, demonstrated on imaginary-time `calcnorm` function.

```
#pragma omp parallel
#pragma omp master
nthreads = omp_get_num_threads();
...
tmpx = alloc_double_matrix(nthreads, Nx); // Allocate separate array for each thread
tmpy = alloc_double_matrix(nthreads, Ny);
...
void calcnorm(double *norm, double **psi, double **tmpx, double **tmpy) {
   int threadid;
   long i, j;
   double tmp;

   #pragma omp parallel private(threadid, i, j)
   {
      threadid = omp_get_thread_num(); // Get a thread identification number

      #pragma omp for
      for (i = 0; i < Nx; i ++) {
         for (j = 0; j < Ny; j ++) {
            tmpy[threadid][j] = psi[i][j] * psi[i][j]; // Store data in separate slots
         }
         (*tmpx)[i] = simpint(dy, tmpy[threadid], Ny);
      }
      #pragma omp barrier // Wait for all threads to complete

      #pragma omp single
      {
         *norm = sqrt(simpint(dx, *tmpx, Nx));
         tmp = 1. / *norm; // 'tmp' is a shared variable, visible to all other threads
      }

      #pragma omp for
      for (i = 0; i < Nx; i ++) {
         for (j = 0; j < Ny; j ++) {
            psi[i][j] *= tmp;
         }
      }
   }
}
```

These functions rely on numerical integration, which essentially computes a sum of a sequence, an operation also known as *reduction*. Reduction can be handled with OpenMP via a special directive, as we can see in Listing 3.16, which shows how the `simpint` function was parallelized. Note that we need parallelized version of numerical integration only when invoked from outside the threaded loops, as the computation loops themselves are parallelized. Further attempts to perform nested parallelization would not yield any improvement in this case.

Listing 3.16: OpenMP parallelization of numerical integration, using the reduction clause.

```
double simpint_th(double h, double *f, long N) {
   int c;
   long i;
   double sum = f[0];

   #pragma omp parallel for private(i, c) reduction(+:sum)
   for (i = 1; i < N - 1; i ++) {
      c = 2 + 2 * (i % 2);
      sum += c * f[i];
   }
   sum += f[N - 1];

   return sum * h / 3.;
}
```

The final portion of the programs that has to be parallelized is the DFT. Instructing FFTW to use OpenMP is a simple task, and requires minimal changes to the code. Before any plan is created or any FFTW function is used, we initialize the threaded FFTW by calling two functions shown in Listing 3.17. The plan creation and execution remain unchanged. Finally, the compiled programs should be linked with the threaded FFTW, by linking to `-lfftw3_omp` in addition to the `-lfftw3`.

Listing 3.17: Initialization of threaded FFTW.

```
#pragma omp parallel
#pragma omp master
nthreads = omp_get_num_threads();
...
fftw_init_threads();
fftw_plan_with_nthreads(nthreads);
// Create plans as usual, e.g., 3D FFT plans
plan_forward = fftw_plan_dft_r2c_3d(Nx, Ny, Nz, **psidd2, psidd2fft, FFT_MEASURE);
plan_backward = fftw_plan_dft_c2r_3d(Nx, Ny, Nz, psidd2fft, **psidd2, FFT_MEASURE);
```

This step completes the OpenMP parallelization of the programs from previous section. The remaining portions of the code, mainly output functions, are not suitable for parallelization, and were thus not improved.

Using the concepts described, we have parallelized all programs in 1D, 2D and 3D, for both imaginary- and real-time propagation, resulting in 10 parallel programs. The programs form a package named *DBEC-GP-OMP*, a subset of a package *DBEC-GP-OMP-CUDA-MPI*, published in Ref. [18] and available for download from Refs. [46, 37]. Names of the programs are the same as in serial implementation with the suffix "`-th`" added. Detailed performance review of parallelized programs presented in this section will follow in Chapter 8.

Chapter 4

Solving NLSE using GPU accelerators

With the shared memory algorithm described and parallelized in the previous chapter, we can consider using an accelerator processor to achieve even better performance. Use of special-purpose hardware architectures to accelerate computationally demanding applications is becoming standard practice in the scientific community. In this chapter we will present the algorithm and implementation which uses one such accelerator, namely the graphics processing unit (GPU). As their name implies, GPUs have been originally designed for graphics-oriented tasks, however nowadays their enormous processing power can be used for general-purpose computation, giving rise to a practice that is now known as *general-purpose computing on graphics processing units* (GPGPU). They achieve this by relying on the large number of slower low-power processing cores, designed to process many parallel streams of data simultaneously, but are ill-suited for complex processing of few streams of data. This is in contrast to modern CPUs, which have few very fast latency-oriented cores, designed to handle single or few streams of data with complex processing (such as regular desktop applications). As a result, GPUs are better suited to handle tasks that significantly benefit from parallelization, which is why we considered using them for our algorithms.

Using GPUs for general-purpose computing by programming only with graphics primitives would be very time consuming and error-prone, severely limiting the usefulness of GPUs in high-performance computing. Fortunately, GPU vendors have recognized this and developed the interfaces for general-purpose programming on GPUs. There are two competing APIs in this space, CUDA [47] and OpenCL [48]. CUDA (formerly an acronym for Compute Unified Device Architecture) is a mature, proprietary, C, C++ and Fortran API and computing platform created by Nvidia, available since 2007. All Nvidia GPUs from that time onward support CUDA, with each generation of GPUs adding more features. GPUs from other vendors are not supported. It is available for free, but is not open-source.

On the other hand, OpenCL, initially developed by Apple, but since transferred to Khronos group, is the standardized approach to programming on heterogeneous platforms. As such, it is not limited only to GPUs from a single vendor (like CUDA is with Nvidia), rather it supports a plethora of different platforms, from CPUs and GPUs to digital signal processors (DSPs), field-programmable gate arrays (FPGAs) and more. Nvidia GPUs also support OpenCL, however it is implemented on top of CUDA, and often has worse performance than using pure CUDA [49]. OpenCL specification is based on the subset of C language, and is similar in functionality to CUDA. Being a standard though, OpenCL often lags a bit with adding new functionality when compared to CUDA, which is controlled by a single company. Note that this is usually not a problem, even for advanced usage.

With the functionality being similar in both CUDA and OpenCL, the choice between the two is often based on the available hardware or developer's personal preference. Since the PARADOX supercomputing facility, where all our programs were developed and tested, has Nvidia GPUs, we have decided to use CUDA for implementation of our algorithm.

In the following sections of this chapter we will first describe the CUDA programming and execution model, taking note of the features we used in our programs, and then proceed with the explanation of the changes to the algorithm required by the implementation using CUDA, and finish with the important details of the implementation itself.

4.1 CUDA programming and execution model

Hardware architecture of Nvidia (and other) GPUs differs in some significant ways to the x86 CPU architecture. Understanding these differences is the key to exploiting the full potential of GPUs. CPU architectures have historically been focused on increasing performance for serial applications, however this trend has slowed down due to energy consumption and heat dissipation problems that arose with increasing clock frequency. Since then the CPU architectures switched to using multiple processing units, giving rise to the multi-core systems which are in use today. On the other hand, GPUs have been focused with increasing performance of floating-point calculations in response to the demand from video game industry. So instead of investing in advancing control logic and branch prediction that would rival the current CPU designs, GPU vendors have focused on increasing throughput through the use of massive number of threads. This approach has been dubbed *many-core*, signifying the larger number of threads than the multi-core systems have. The software written for this type of computing platform is expected to work with the large number of parallel threads, which the hardware takes advantage of to hide the latency of the memory accesses or arithmetic operations.

Hardware implementation of aforementioned concepts comes in a form of *Streaming Multiprocessor* (SM). The Nvidia GPU architecture is built around an array of these SMs, each executing in parallel with the others. Inside each SM there is a number of *CUDA cores* that execute a sequential thread, various caches, special function units (SFU) that are used for transcendental operations (e.g., sin, cos) and a small amount of shared memory available to all CUDA cores within the SM. A simplified architecture of a CUDA device is shown in Figure 4.1. Modern Nvidia GPUs have a large number of SMs and cores, which, when combined, allow for hundreds or even thousands of simultaneous threads to execute. For example, the previous-generation Tesla M2090 has 512 cores, while the newer Tesla K80 has 4992 CUDA cores. Cores execute in *Single Instruction Multiple Thread* (SIMT) model, similar to the Single Instruction Multiple Data (SIMD) model [50]. This means that all cores within the same group execute the same instruction at the same time, in lock-step fashion. These groups are called *warps*, and in current hardware implementations, each warp has 32 threads. One peculiar behavior of SIMT is in case of conditionals, which SIMT handles by disabling the threads that would not execute the same code, and then rerun the same code with the previously disabled threads re-enabled. This has negative performance impact, therefore conditional computation that would cause thread divergence should be avoided in GPU code, if possible.

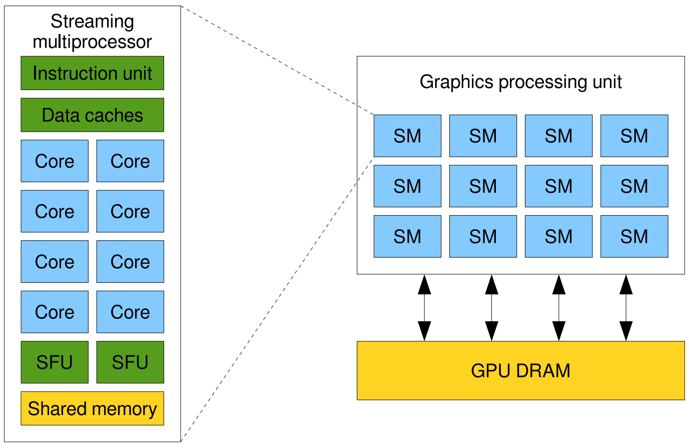


Figure 4.1: Simplified architecture of modern Nvidia GPU.

Within each SM, there is also a small amount of cache memory, named *shared memory*, which is available to all threads within the SM. Because it resides on-chip, shared memory is much faster than main GPU memory, but it is limited in size, with most GPUs having access to 48–96 KB. This type of memory is useful for many scenarios, such as user-managed data caches or high-performance cooperative parallel algorithms (like parallel reductions, which we use in our algorithm for numerical integration).

The main memory of the GPU (named *global memory* in CUDA terminology), its RAM, is separate from the main memory of the host computer on which the GPU is installed. GPU is connected to a host through PCI-Express, and all data transfers between the host and GPU device have to go through this I/O bus. Data is usually transferred between the GPU and host memory using direct memory access (DMA), either in synchronous or asynchronous fashion. Alternatively, GPU may access host's memory through so called *mapped memory*, where data transfers happen on the fly, however, accessing memory in this way is usually far slower and should thus be used sparingly. Current-generation Nvidia GPUs offer between 2 GB and 24 GB of memory, which is enough for some use-cases, but still noticeably less than amount of RAM memory current high-end computing nodes have. This means that for memory-demanding programs, such as the ones we describe in this thesis, efficient use of GPU memory is paramount.

CUDA execution model states that instructions are issued and executed per warp. This is also true for global memory operations. The CUDA device combines, or *coalesces*, these global memory load/store operations issued by threads of a warp into as few transactions as possible, to maximize memory throughput. Additionally, for data to be read or written by memory transactions it must be naturally aligned, i.e., its first address must be a multiple of their size (32, 64, or 128 bytes). This means that the most favorable memory access pattern is achieved when all threads in a warp access consecutive global memory locations, and in the case of multidimensional data, its innermost dimension must be padded to the nearest natural boundary. The difference between coalesced and non-coalesced (non-sequential or unaligned) access pattern is illustrated in Figure 4.2. Failure to meet the coalesced access pattern will result in more transactions per instruction, reducing the memory throughput and increasing execution time. Memory coalescence is also very important when dealing with multidimensional data. For accesses to this type of data to be fully coalesced, the number of elements in its innermost dimension must be a multiple of the warp size. In particular, this means that data whose innermost dimension is not a multiple of this size will be accessed much more efficiently if it is actually allocated with a number of elements in its innermost dimension rounded up to the closest multiple of this size.

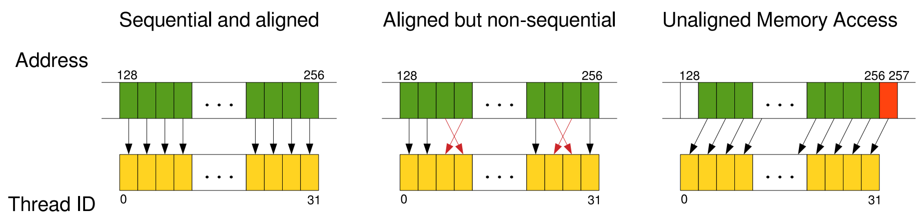


Figure 4.2: Different memory access patterns on GPU device.

Programming for CUDA devices in done via a C-like language, which is essentially an extension of C language to support GPU computation. CUDA programming model reflects the heterogeneous nature of a system which has a GPU, by making a distinction between the *host* code, which executes on the CPU, and the *device* code, which executes on the GPU. Host code may contain any valid C code, whereas device code is declared by special language extensions. Any given CUDA source file may contain a mixture of host and device code. Because CUDA source file with device code is not valid C, it needs to be compiled with a special compiler, the Nvidia C compiler (`nvcc`). Internally, `nvcc` works by splitting the source files into host and device code, compiling them separately, and then linking them into a final executable. It can also be used to compile and link only the device code, which is useful when integrating CUDA code into existing programs.

A key component of CUDA programming model is the *kernel*, a specially annotated function that executes on the CUDA device. When the kernel is invoked, or *launched*, it is executed by a large number of threads on the device. Kernels do not create or manipulate threads directly, instead the number of threads is defined during kernel launch and device takes control of creating and scheduling them for execution. To differentiate between the threads and to assign different work for them, special variables exist which can only be used inside kernels. Kernel is launched using a special syntax, in the form of `kernel_name«< ... »>(...)`, where the launch parameters are specified. Listing 4.1 demonstrates the basic definition and execution of a CUDA kernel.

Listing 4.1: Example of a simple CUDA program, with transferring data to/from GPU device, executing a kernel, and allocating/deallocating resources.

```
#include <stdio.h>
// Kernels are functions defined with '__global__' and returning 'void'
__global__ void square(int Nx, double *psi, double *psi2) {
   int i = blockIdx.x * blockDim.x + threadIdx.x; // Compute the thread index in a grid
   if (i < Nx) psi2[i] = psi[i] * psi[i]; // Ensure that we don't go out of bounds
}

void main() {
   int Nx = 1<<20; // 1048576
   double *h_data, *h_data2, *d_data, *d_data2;
   // Allocate data on host and device
   data = (double*) malloc(Nx * sizeof(double));
   data2 = (double*) malloc(Nx * sizeof(double));
```

```
    cudaMalloc(&d_data, Nx * sizeof(double));
    cudaMalloc(&d_data2, Nx * sizeof(double));
    // Initialize 'h_data'
    for (int i = 0; i < Nx; i++) { ... }
    // Copy data from host to device
    cudaMemcpy(d_data, h_data, Nx * sizeof(double), cudaMemcpyHostToDevice);
    // Invoke kernel with 256 threads per block, 4096 blocks in total
    square<<<(Nx + 255) / 256, 256>>>(Nx, d_data, d_data2);
    // Copy the result from device back to host
    cudaMemcpy(h_data2, d_data2, Nx * sizeof(double), cudaMemcpyDeviceToHost);
    // Use 'h_data2' for anything
    for (int i = 0; i < Nx; i++) { ... }
    // Free resources
    cudaFree(d_data); cudaFree(d_data2); free(h_data); free(h_data2);
}
```

Collection of threads that are generated by a kernel launch are referred to as a *grid*. When a host code launches a kernel, the CUDA runtime system generates a grid of threads that are organized in a two-level hierarchy. On the first level, each grid is organized into an array of *thread blocks*, which are in turn organized as an array of threads on the second level. All thread blocks of a grid are of the same size, and (on current CUDA devices) each thread block can contain up to 2048 threads. The number of threads in each thread block and the number of thread blocks are specified in the host code with kernel launch parameters. This hierarchy allows CUDA programming model to be scalable regardless of the underlying GPU architecture. CUDA runtime is in charge of scheduling thread blocks for execution on available SMs. Because there is no guarantee of which thread blocks are currently executing nor is their order of execution known in advance, there is no way to synchronize between thread blocks. Instead, it is expected that algorithms will be designed in a way so that they do not require synchronization between thread blocks. The array of thread blocks in a grid and the array of threads in a block are not necessarily 1D, and may be organized as 2D or 3D arrays. In fact, these arrays are always 3D, with unused dimensions initialized to 1. Figure 4.3 illustrates the thread hierarchy on an example with 2D array of thread blocks.

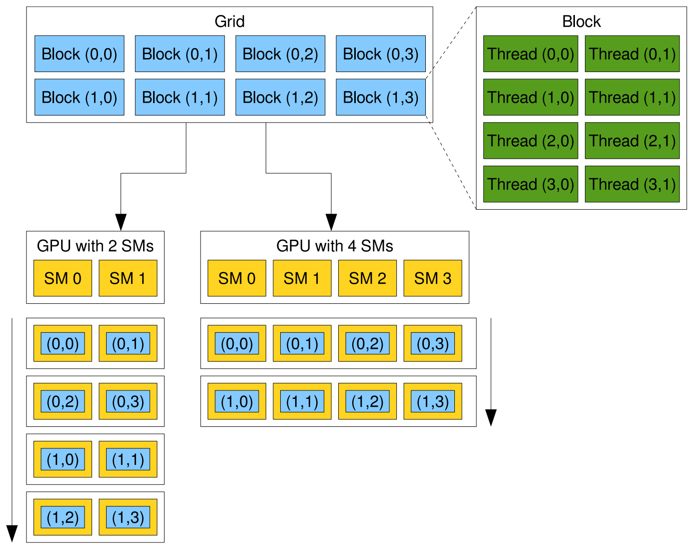


Figure 4.3: Execution of 2D array of thread blocks on a different number of SMs.

Kernels operate on device memory and in most cases have no access to host memory. Just like the memory in a traditional C program, memory on the device must first be allocated, and then freed after use. Working with device memory is very similar to the way we work with host memory. There are four main memory operations, `cudaMalloc`, `cudaMemcpy`, `cudaMemset` and `cudaFree`, which are the counterparts of the standard C functions `malloc`, `memcpy`, `memset` and `free`, respectively. In most cases, CUDA programs follow the same processing flow, where the data required for computation is first copied from the host's memory to the device, then one or more kernels are invoked that perform some computation using that data, and finally the resulting data is copied back to the host. Since the memory transfers use the PCI-Express bus to copy the data between memories of host and device, they are in general much slower than the computation on either the host or the device. For this reason it is very important to minimize memory transfers, as well as to use asynchronous operations to overlap memory transfer with computation. We relied on these techniques in our heterogeneous implementation, with details presented in Chapter 5.

An important aspect of CUDA programming is error handling. All CUDA runtime functions return an error code and it is good practice to check for errors after any CUDA runtime function is invoked. Since the errors they produce usually do not terminate the whole program, failure to check for errors can lead to many unexpected situations and complicate debugging. In examples which we present throughout this thesis we will omit the error checking for brevity, however the proper error handling is in fact implemented in all of the GPU-related portions of the programs presented.

In this section we have described only the most basic concepts of GPU programming with CUDA. Many details have been omitted as we focus on the core concepts that we relied on in our implementations. More detailed description of CUDA can be found in [51, 52, 53].

4.2 CUDA implementation of shared memory algorithm

The significant differences in architecture of the GPU and its accompanying execution model on one side, and CPU architecture with its conventional execution model on the other side, did not result in significant changes to the main algorithm we employ. From the algorithm's point of view, both CPU and GPU are shared memory systems with multiple processing units, even though they are vastly different, and therefore can be used in a similar manner. Main loop of the algorithm remains unchanged, as well as each substep. Auxiliary algorithms, like the algorithm for numerical integration, which are more dependent on the execution model, need to be adapted to the new environment, as we will describe below.

While the algorithm and thus the flow of the programs remain conceptually the same, actual implementation on CUDA platform required significant changes to C/OpenMP programs. We have focused our efforts on using GPU for computationally demanding functions performing time evolution of the wave function (`calcpsidd2`, `calcnu`, `calclux`, `calcluy` and `calcluz`), normalization (`calcnorm`) and calculation of physical quantities (`calcmuen`, `calcrms`). As 1D variants of the programs are sufficiently fast even in their serial implementation, and the inherent difficulty in parallelizing recursive relations which are part of `calclux`, `calcluy` and `calcluz` functions, we have implemented the algorithm on GPU for 2D and 3D variants of the programs. In the resulting implementation, CPU host was used for initialization of variables, control of program flow, and I/O. Computation functions were reimplemented as CUDA kernels, while the initialization and output functions have remained host-only and serial. Since the initialization of variables uses host RAM memory, programs require the same amount of host and device memory. Before any computation begins, variables are copied to GPU device, where they remain during computation, and only wave function variable (`psi`) is returned back to CPU memory when it is required for writing output. Implementation was done in several steps, each offloading more computation to the GPU.

The first step in migration to the CUDA platform was to use CUDA built-in types for complex numbers instead of the ones provided by the C99 standard. This was necessary only in real-time propagation programs because complex numbers, as implemented in C99 standard of the C language, are not directly supported in CUDA device code. However, converting them to CUDA built-in complex types is possible as they share the same binary representation. Complex numbers in CUDA do not support all the operations that C99 standard provides. Namely, the function to compute the complex exponential of a number, which we rely on in function `calcnu`, is missing. This function can be replaced by an approximation using `sincos` function available in Linux (though it is not part of a standard C library).

Next step was to replace the FFTW library used by C/OpenMP programs with the cuFFT library, which is available in CUDA. cuFFT provides two interfaces: the native one and the FFTW one [54]. FFTW interface is intended to be used as a drop-in replacement for FFTW, allowing programs written primarily with FFTW in mind to use CUDA GPUs with minimal modifications to the source code. While this interface could be used as a temporary solution during the development of CUDA programs, our goal of using CUDA for all computation relating to the propagation of the wave function could only be reached with the native cuFFT interface. This interface is modeled after FFTW but differs from it in the way plans are created and executed. In cuFFT, precision and type of data of the transform is determined by the functions initiating execution of plans, which results in a slightly different sequence of function calls. We used cuFFT in much the same way as FFTW, for doing R2C and C2R FFT transforms.

With the computation of FFT offloaded to GPU, we proceed and port the remaining wave function propagation functions to GPU. Most of the computation functions have nested loops, either two or three levels deep. This could naturally be mapped onto a 2D or 3D grid of threads, with each thread processing a single element of the corresponding arrays. This is the standard approach when writing CUDA kernels, called *monolithic kernel*, and is often found in CUDA textbooks. While this way of writing kernels is sufficient for our needs, we have opted to use *grid-stride* loops when writing kernels, as proposed in Ref. [55]. Basic grid-stride loop is illustrated in Listing 4.2. The stride of the loop is the total number of threads in the grid, which means that all memory accesses are coalesced for maximum performance, in the same way as in the monolithic approach. Addition of a loop in a kernel does not add to its instruction complexity, which would impact performance. This is because we need instructions of the same complexity in the form of conditional statements in the monolithic kernel to ensure that we do not access data out of bounds. We should also note that we have found grid-stride loops to be more flexible and easier to debug. Kernels with grid-stride loops can be launched with just a single thread and a single block, which would essentially make the kernel serial in nature and thus far easier to debug.

The remaining functions we ported, `calcnorm`, `calcmuen` and `calcrms`, rely on 1D spatial integration implemented with composite Simpson's rule. This function requires computation of a sum of elements of an array, which is trivial to implement in serial or OpenMP code, but can be much more difficult using CUDA due to lack of global synchronization between thread blocks.

Listing 4.2: Example of 1D grid-stride loop.

```
int i, tid_x, grid_stride_x;

tid_x = blockIdx.x * blockDim.x + threadIdx.x;
grid_stride_x = blockDim.x * gridDim.x;

for (i = tid_x; i < d_Nx; i += grid_stride_x) {
   ...
}
```

Nvidia GPUs have support for atomic operations, which could be used to significantly simplify the implementation, however, only the latest-generation Nvidia GPUs have the hardware support for atomic operations on double precision data. Alternative implementations of atomic operations on double precision data rely on compare-and-swap (CAS) instructions that may incur a performance penalty [56]. For this reason we have based our implementation of composite Simpson's rule on the reduction algorithm from Ref. [57]. We extended the reduction algorithm to support array sizes which are not powers of two and to compute three sums simultaneously. The algorithm operates in two steps, implemented as two kernel invocations. In the first invocation, each thread block creates a partial sum using a tree-based approach with shared memory and stores the partial result in main memory. In the second kernel invocation, this time with only one thread block, the partial sums from previous invocation are combined to produce the final result. An illustration of the algorithm is given in Figure 4.4, and its implementation is shown in Listing 4.3.

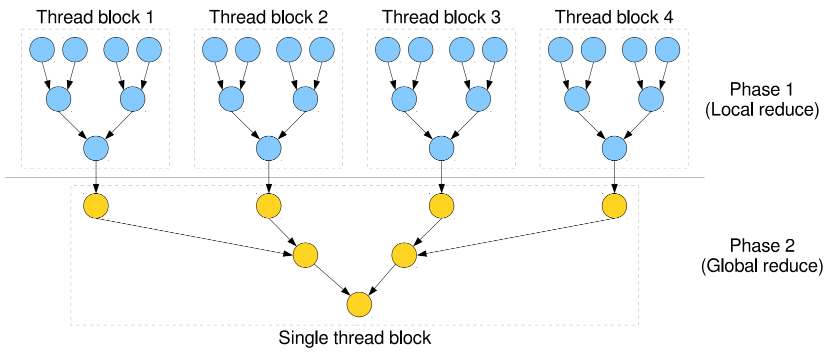


Figure 4.4: Two-phase reduction algorithm on CUDA device. In this example, four thread blocks are used in the first phase, while a single block computes the final result in the second phase.

Listing 4.3: Implementation of composite Simpson's rule with CUDA.

```
// Initiate temporary storage on device
void simpint_init(long N) {
   sum = alloc_double_vector_device(1);
   sumi = alloc_double_vector_device(1);
   sumj = alloc_double_vector_device(1);
   sumk = alloc_double_vector_device(1);
   tsumi = alloc_double_vector_device(ceil(1. * N / CUDA_BLOCK_SIZE / 2));
   tsumj = alloc_double_vector_device(ceil(1. * N / CUDA_BLOCK_SIZE / 2));
   tsumk = alloc_double_vector_device(ceil(1. * N / CUDA_BLOCK_SIZE / 2));
}
```

```
double simpint_gpu(double h, double *f, long N) {
   double f_sum;

   dim3 block; dim3 grid; int shmem;
   block.x = CUDA_BLOCK_SIZE; // For example, 128
   grid.x = ceil(1. * N / block.x / 2);
   shmem = 3 * block.x * sizeof(double);
   // Compute partial sums and reduce to a single block
   simpint1d_kernel1<<<grid, block, shmem>>>(tsumi, tsumj, tsumk, f+1, f, f+2, N-2, 2);
   // Compute final sums with just one block
   simpint1d_kernel1<<<1, block, shmem>>>(sumi, sumj, sumk, tsumi, tsumj, tsumk, grid.x,1);
   // Compute the final result of integration
   simpint1d_kernel2<<<1,1>>>(h, sum, sumi, sumj, sumk, f, N);
   cudaMemcpy(&f_sum, sum, 1 * sizeof(double), cudaMemcpyDeviceToHost);

   return f_sum;
}

__global__ void simpint1d_kernel1(double *sumi, double *sumj, double *sumk, double *ini,
    double *inj, double *ink, long N, int step) {
   extern __shared__ double psumi[]; // Shared memory for partial sums
   double *psumj = &psumi[blockDim.x]; double *psumk = &psumi[2 * blockDim.x];
   double tsumi = 0.; double tsumj = 0.; double tsumk = 0.;

   int tid = threadIdx.x;
   long idx = (blockIdx.x * blockDim.x + tid) * step;

   if (idx < N) {
      tsumi += ini[idx]; tsumj += inj[idx]; tsumk += ink[idx];
   }

   psumi[tid] = tsumi; psumj[tid] = tsumj; psumk[tid] = tsumk;

   __syncthreads();

   // Start the shared memory loop on the next power of 2 less than the block size.
   // If block size != power of 2, accumulate the intermediate sums in the remainder range.
   int pow2 = blockDim.x;

   if (pow2 & (pow2-1)) {
      while (pow2 & (pow2-1)) {
         pow2 &= pow2-1;
      }
      if (tid >= pow2) {
         psumi[tid-pow2] += psumi[tid];
         psumj[tid-pow2] += psumj[tid];
         psumk[tid-pow2] += psumk[tid];
      }
      __syncthreads();
   }
   for (int th = pow2 >> 1; activeThreads; activeThreads >>= 1) {
      if (tid < activeThreads) {
         psumi[tid] += psumi[tid+activeThreads];
         psumj[tid] += psumj[tid+activeThreads];
         psumk[tid] += psumk[tid+activeThreads];
      }
      __syncthreads();
   }
   if (tid == 0) {
     sumi[blockIdx.x] = psumi[0]; sumj[blockIdx.x] = psumj[0]; sumk[blockIdx.x] = psumk[0];
   }
}

__global__ void simpint1d_kernel2(double h, double *sum, double *sumi, double *sumj,
    double *sumk, double *f, long N) {
   *sum = *sumj + 4. * *sumi + *sumk;
   if(N % 2 == 0) *sum += (5. * f[N - 1] + 8. * f[N - 2] - f[N - 3]) / 4.;
   *sum = *sum * h / 3.;
}
```

The aforementioned algorithm works well in situations where we need just one call of the integration function. However, in our programs, this routine is often called inside nested loops. Invoking two kernels in each iteration creates an overhead that makes this approach slower than the OpenMP-threaded programs. In case of nested loops, we can use the similar approach used in `calclux`, `calcluy` and `calcluz` functions, by performing the whole integration in each GPU thread. Just like in the case of recursive relations, the integration via composite Simpson's rule is much slower on GPU threads than on CPU threads, however the large number of parallel GPU threads successfully hides their individual latency. Since both approaches have strengths and weaknesses, we used a combination of both in our implementation of `calcnorm`, `calcmuen` and `calcrms` functions. These functions share a common flow, shown in Listing 4.4.

Listing 4.4: Performing reduction along multiple directions, demonstrated on the 3D real-time `calcrms` function.

```
void calcrms(...) {
   // Prepare data in a tensor
   calcrms_kernel1x<<<dimGrid3d, dimBlock3d>>>(psi, tmpyzx, d_x2);
   // Reduce from 3D to 2D
   simpint3d_kernel<<<dimGrid2d, dimBlock2d>>>(dx, tmpyzx, tmpyz, Ny, Nz, Nx);
   // Reduce from 2D to 1D
   simpint2d_kernel<<<dimGrid1d, dimBlock1d>>>(dz, tmpyz, tmpy, Ny, Nz);
   // Reduce from 1D to scalar
   rms[1] = sqrt(simpint_gpu(dy, tmpy, Ny));
   ...
}

__global__ void simpint2d_kernel(double h, cudaPitchedPtr matrix, double *vector, long N1,
    long N2) {
   ...
   for (i = blockIdx.x * blockDim.x + threadIdx.x; i < N1; i += blockDim.x * gridDim.x) {
      matrixrow = get_double_matrix_row(matrix, i);
      vector[i] = simpint(h, matrixrow, N2); // Each thread calls sequential function
   }
}

__global__ void simpint3d_kernel(double h, cudaPitchedPtr tensor, cudaPitchedPtr matrix,
    long N1, long N2, long N3) {
   ...
   for (i = blockIdx.y * blockDim.y + threadIdx.y; i < N1; i += blockDim.y * gridDim.y) {
      matrixrow = get_double_matrix_row(matrix, i);
      for (j = blockIdx.x * blockDim.x + threadIdx.x; j < N2; j += blockDim.x * gridDim.x){
         tensorrow = get_double_tensor_row(tensor, i, j);
         matrixrow[j] = simpint(h, tensorrow, N3);
      }
   }
}
```

In Section 3.1 we mentioned that we perform flat allocation of CPU memory even when dealing with multidimensional data. This property was especially useful when transitioning to GPU. Contiguous (flat) memory allowed us to transfer whole multidimensional variables with one call to `cudaMemcpy`. While we also used flat allocations on GPU, we did not rely on double or triple pointers to index our variables. Using them is possible on GPU, however correct usage is much more complicated, and may reduce performance [58]. Since we are focused on 2D and 3D variants of our programs, we also had to ensure that we are optimally accessing memory of multidimensional variables. To this end, we have used *pitched* memory allocations, to make sure that the allocation is appropriately padded to allow coalesced access. Pitched memory can be allocated with `cudaMallocPitch` or `cudaMalloc3D` which perform the same allocation routine, just with a slightly different interface. Transferring pitched data to/from GPU is also done with special functions, `cudaMemcpy2D` and `cudaMemcpy3D`, which follow the same differences. We settled on using `cudaMalloc3D` and `cudaMemcpy3D` in our programs, which we found to be more convenient due to the fact that `cudaMalloc3D` neatly groups all allocation-related variables in one C structure. The pitch of allocation must be used to access elements of an allocated array, which makes the index calculation slightly different, as shown in Listing 4.5. Note that in these functions the innermost dimension (`Nz` in our case) is specified first, and the slowest changing (`Nx`) last, resulting in an unusual order when compared to the memory allocation functions we presented in the previous chapter. Nevertheless, this is just a cosmetic issue which does not impact performance in any way.

Listing 4.5: Accessing pithed memory on a device.

```c
// Allocated memory resides in a special structure
struct cudaPitchedPtr tensor;
cudaMalloc3D(&tensor, make_cudaExtent(Nz * sizeof(double), Ny, Nx);

// Define code for accessing specified row
// Equivalent to double *row = tensor[slice][row] on host
__device__ double *get_double_tensor_row(cudaPitchedPtr tensor, long slice, long row) {
  return (double *)((((char *)tensor.ptr) + slice * tensor.pitch * tensor.ysize) + row *
      tensor.pitch);
}

// From device code, access the specified row, and k-th element inside
// Equivalent to double val = tensor[i][j][k] in host code
double *tensorrow = get_double_tensor_row(tensor, i, j);
double val = tensorrow[k];
```

Changes required to support pitched memory were in most cases straightforward, however special care has to be taken when working with cuFFT, as its basic interface does not assume that memory is padded. Similarly to the advanced interface of FFTW, the advanced interface of cuFFT supports working with a subset of a larger (in this case padded) multidimensional array. Using pitched memory does not cause a significant increase in memory usage on GPU in most cases. On the other hand, parallelization of outer loops requires additional per-thread memory, in the same way as in OpenMP parallelization. Since GPU usually executes more threads simultaneously than what is available on the CPU, this increase can be significant. Given that GPUs in most cases have less RAM than their host, high memory usage means that the GPU can be used for much smaller problem sizes.

To curb memory usage, we reused memory as much as possible, and reduced the memory footprint of some functions. We allocate up to four matrices/tensors (for 2D and 3D, respectively), which are used throughout the programs. These four matrices/tensors hold wave function values (variable `psi`), temporary data, trap potential (`pot`) and Fourier transform of the dipolar potential (`potdd`).

Allocating data for the two potentials that we use has been made optional, and can be exploited to further reduce memory usage. We exposed this functionality through `POTMEM` input parameter. The three possible values of `POTMEM` are `0`, `1` and `2`. Setting the value of `POTMEM` to `2` instructs the programs to allocate two separate matrices/tensors which will be used to store trap potential and dipolar potential in GPU memory. This provides the best performance at the expense of increased memory usage, and thus smaller maximum mesh size. Setting `POTMEM` to `1` will allocate only one matrix/tensor, which will store one of the two potentials, as needed. The required potential will be copied asynchronously in the background while other computations on GPU take place. Initially, the matrix/tensor will store trap potential. During execution of the `calcpsidd2` function, the dipolar potential will be copied over the trap potential, and after it is no longer needed, the trap potential will replace it for the rest of the computation. This cycle will repeat itself in each iteration of the wave function propagation. The last option, setting `POTMEM` to `0`, will not allocate any memory for the potentials on GPU, and will instead expose the host memory of the potentials as mapped memory. Leaving the `POTMEM` parameter undefined will instruct the programs to try and select the optimal value based on the size of the mesh. Figure 4.5 illustrates the three different settings of the `POTMEM` parameter. We suggest using `POTMEM` value of `2` if possible, or leaving it undefined, and using values of `1` or `0` if programs do not correctly predict the optimal value, or the problem cannot fit into GPU memory.

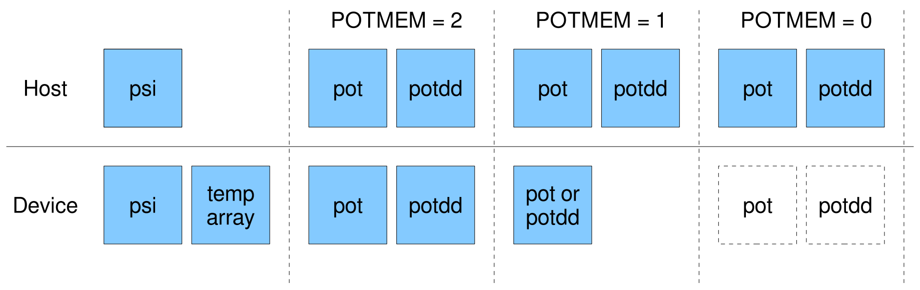


Figure 4.5: Illustration of placement of relevant variables in CPU and GPU memory.

We allocate matrix/tensor for temporary data as a flat array of double or double complex values, in the same way as for the other matrices/tensors. This matrix/tensor is used in real-time propagation functions `calclux`, `calcluy` and `calcluz` to store the Crank-Nicolson coefficients. Alternatively, complex matrix/tensor can be further divided into two matrices/tensors of double values. Since the arrays we get in this way are also used in computation of R2C FFT, we had to ensure the matrix/tensor had the proper size, that is `2 * Nx * (Ny / 2)` and `2 * Nx * Ny * (Nz / 2)` in 2D and 3D variants of programs, respectively. To use this memory allocation scheme effectively, we had to reorganize some of the computations inside `calcmuen` and `calcpsidd2`. In fact, the memory optimizations of those two functions mentioned in Chapter 3 were inspired by the optimizations we had to introduce in GPU variants of programs. Changes required in these functions are conceptually very similar to the ones we presented in Chapter 3, therefore we will not go into all of their details here. One notable difference is the additional restriction we have to place on FFT computation in order to keep its memory usage in check. We do this by ensuring that cuFFT reuses temporary storage we already allocated to store its intermediate results. By default, cuFFT will allocate memory for intermediate results separate from the already allocated input/output arrays. This can be changed, however we must ensure that we have allocated enough memory to meet cuFFT demands. The amount of memory required for FFT temporary data varies with the transform size due to different algorithms used. Some transform sizes require much more memory than others, in some cases up to eight times more [54]. This amount can be larger than what we have already allocated, in which case the programs will select a nearest larger transform size that can fit in the allocated memory and perform the computation as if the user has entered the corresponding number in the input file for the mesh size. The output of programs will report if it had to make these adjustments to the mesh size.

When all the memory optimizations are combined, we get an approximately 50% reduction in the memory usage compared to the initial GPU implementation. This not only allowed us to use this implementation for much larger mesh sizes, but also to use similar optimizations to improve all variants of programs descried in this thesis.

CUDA programs have been grouped and published as *DBEC-GP-CUDA* package in Ref. [17], which can be downloaded from Refs. [59, 37]. The package consists of 2D (corresponding to x-y and x-z planes) and 3D programs, in both imaginary- and real-time propagation, resulting in 6 programs in total. We followed the similar naming convention as in the case of OpenMP-parallelized programs, by adding a suffix “`-cuda`” to the base program name. Performance evaluation of these programs is given in Chapter 8.

Chapter 5

Hybrid algorithm for heterogeneous computing systems

In this chapter we describe the hybrid algorithm targeting heterogeneous systems. While this algorithm can be implemented on every heterogeneous system where processing resources have separate memory (i.e., the memory is not shared), here we focus on platforms consisting of CPUs and GPUs on the same computer, as it is the most common heterogeneous platform. Thus, the algorithm described here is a combination of algorithms from Chapters 3 and 4, further extended to allow for simultaneous computation. The resulting hybrid algorithm and its implementation is made possible by the fact that pure GPU solution does not use CPU for any computation, rather the CPU is used only to control the execution of GPU kernels and perform I/O operations. Modern computers with powerful GPUs usually also have a powerful multi-core CPUs, so using only GPU presents a missed opportunity to achieve better performance through the use of all available resources.

Our goal was to extend the algorithm so that we can reuse as many portions of the CPU and GPU programs as we can. We did this by introducing GPU support over the base OpenMP-parallelized CPU programs from Chapter 3. Numerically most demanding functions were extended to asynchronously offload computation to GPU, and then proceed with the computation on CPU using multiple threads.

The most important part of creating a hybrid algorithm is the distribution of data between CPU and GPU in a way that maximizes the performance of any function involved in time propagation of the wave function. We first give a detailed description of how data is distributed, followed by a description of hybrid computation functions and hybrid FFT in Section 5.1. Details of basic implementation that uses all available CPU cores and GPUs are given in Section 5.2. The implementation of the data distribution scheme revealed that the time it takes to transfer the data to GPU and back dominates the execution time. To optimize the implementation and minimize the impact of data transfers, we have used CUDA streams to hide the latency of data transfers, as demonstrated in Section 5.3 along with other memory usage optimizations we applied.

5.1 Description of the hybrid algorithm

In order to use CPU and GPU at the same time, we have to divide the work efficiently between them. If we put aside the obvious hardware architecture difference between CPU and GPU which we described in Section 4.1, we can consider the computer with GPU as being similar to the distributed memory system consisting of two computing nodes with different characteristics. Both CPU and GPU have their own memory, and are connected through a fast interconnect, in this case the PCI-Express. This means that we need an approach similar to the one distributed memory algorithms use to achieve the desired parallelization. The most important difference here is that all data is available in main (CPU) memory, and only portions of it need to be transferred to GPU memory, as opposed to the true distributed memory systems, where no single part of the system has all the data. Therefore, the general approach to perform this type of hybrid computation is to:

(i) designate a portion of data to be processed by GPU,

(ii) copy that data to GPU while simultaneously using CPU for computation over the remaining data,

(iii) transfer the data back from GPU, overwriting old data, and

(iv) synchronize CPU and GPU.

This flow is illustrated in Figure 5.1.

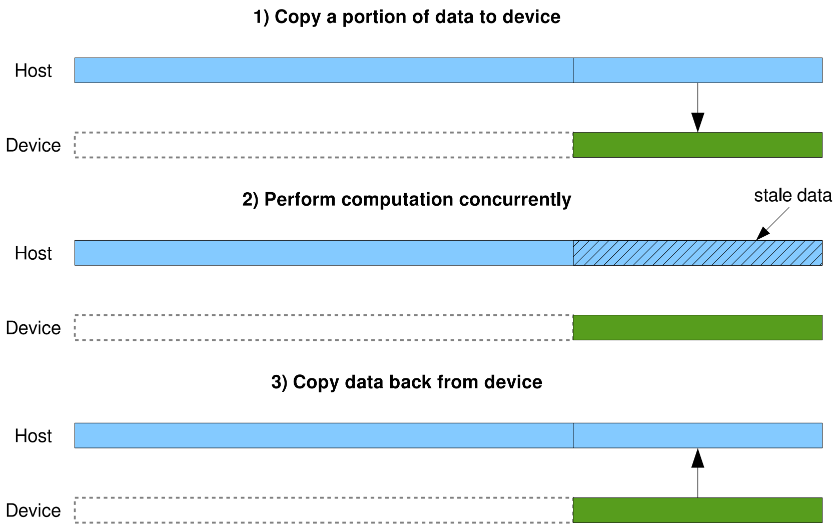

1) Copy a portion of data to device
Host
Device
2) Perform computation concurrently
stale data
Host
Device
3) Copy data back from device
Host
Device

Figure 5.1: Flow of data between host and device.

Not every function is feasible for simultaneous computation on CPU host and GPU device. Functions calculating physical properties (chemical potential, energy, norm and RMS size), or functions not executed in each iteration are not suitable for computation on GPU for various reasons. Specifically, functions which require multiple invocations of numerical integration do not achieve noticeable speedup on GPU, and when coupled with data transfer, actually perform worse than on CPU alone. We have therefore focused our efforts on speeding up only the functions relating to time propagation of the wave function, which are invoked in each step of the main loop, i.e.

`calcnu`, `calclux`, `calcluy`, `calcluz` and `calcpsidd2`. The function computing the norm of the wave function was not offloaded to GPU for real-time propagation, as it is not invoked in each iteration. On the other hand, in imaginary-time propagation, in which normalization is performed as the last step of each iteration, we perform a concurrent computation on both CPU and GPU only for the normalization step (i.e., dividing the wave function by the norm), not the computation of the norm itself.

Data can be offloaded in various ways [60, 61] and, in general, any data distribution scheme which targets distributed memory systems may be used. However, we settled on a simple approach using 1D decomposition, also known as *slab decomposition*.

When using slab decomposition, data is distributed along one dimension, usually the slowest-changing one, and the remaining dimension (in 2D programs) or dimensions (in 3D programs) of data remain local. This means that we can perform computation on the local data efficiently, and no data exchanges are necessary. Depending on a data access pattern, in our programs we rely on decompositions either along the x direction (discretized with the `Nx` spatial points), or along the y direction (discretized with the `Ny` spatial points). In 3D programs, there is no need to decompose the data along the z direction. In case we need data from the distributed dimension to become local in order to perform some computation (e.g., to update the wave function values in `calclux` function when data are decomposed along the x direction), we have to reassemble data in host memory and decompose again along the appropriate dimension. Figure 5.2 illustrates this concept.

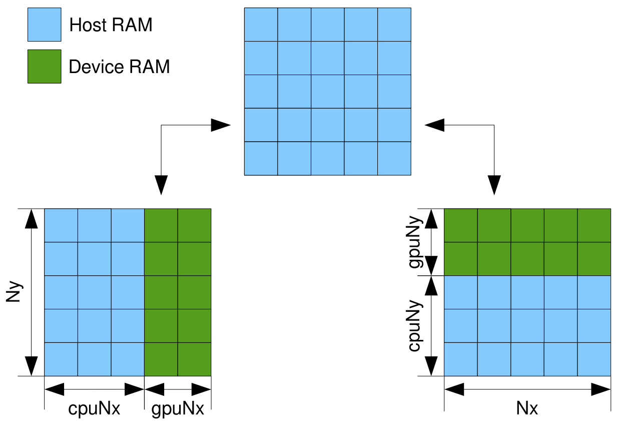


Figure 5.2: Two offload patterns used. If offloaded along x direction, the y direction remains local, however if we need to access whole x direction data, we need to decompose along y direction.

With the decomposition scheme in place, we can consider how to divide the computation of the wave function propagation in time between CPU and GPU. Once the data have been distributed between CPU and GPU, on the CPU side only the exit condition of the outer loop needs to be adjusted, so that the CPU processes a smaller number of elements (e.g., by replacing `Nx` with `cpuNx`). Similar changes are required on the GPU side as well, however we also need to ensure that GPU has valid data to work with, and that data are reassembled in host memory after the computation on GPU is done. Introduction of offloading to GPU did not require significant changes to the functions calculating individual parts of the wave function propagation, and all changes are similar.

Initially, we distribute the data along the slowest-changing dimension. This allows CPU and GPU to simultaneously perform computation of wave function propagation w.r.t. H_1 and H_3 parts of the Hamiltonian (and H_4 when working in 3D). The propagation of the wave function w.r.t. H_2, corresponding to the x direction, can be done locally only if we decompose the data along the y direction before invoking the function `calclux` on CPU and GPU. Note that we do not have to transfer data from GPU back to host memory at the end of each function, due to the fact that time propagation w.r.t. H_2, H_3 and H_4 parts of the Hamiltonian can be done in arbitrary order in each step. Without this, the time propagation workflow of 3D programs would have to involve the following steps, according to Figure 3.2:

1. calculate the dipolar term,
2. transfer the data decomposed along x direction to GPU,
3. propagate the wave function w.r.t. H_1
4. reassemble the data in host memory,
5. transfer the data decomposed along y direction to GPU,
6. propagate the wave function w.r.t. H_2
7. reassemble the data in host memory,
8. transfer the data decomposed along x direction to GPU,
9. propagate the wave function w.r.t. H_3
10. propagate the wave function w.r.t. H_4
11. reassemble the data in host memory.

Rearranging the order of time-propagation substeps allows us to remove one transfer of data to GPU and its subsequent reassembly in host memory:

1. calculate the dipolar term,
2. transfer the data decomposed along x direction to GPU,
3. propagate the wave function w.r.t. H_1
4. propagate the wave function w.r.t. H_3
5. propagate the wave function w.r.t. H_4
6. reassemble the data in host memory.
7. transfer the data decomposed along y direction to GPU,
8. propagate the wave function w.r.t. H_2
9. reassemble the data in host memory,

Therefore, we use the above, optimized sequence in our 3D programs, and similarly in 2D.

To complete our hybrid algorithm, we also need to distribute computation of the dipolar term between CPU and GPU, where complexity arises in performing DFT on distributed data. Currently available FFT libraries target either CPU or GPU for their computation, but unfortunately not both at the same time. There are numerous attempts to develop specialized FFT libraries which would enable this [62, 63, 64], however a full-featured library with support for R2C transforms with advanced data layout is still not available. The approach we used is to rely on existing libraries for actual transforms, but perform data distributions manually. The libraries we use, FFTW on CPU and cuFFT on GPU, support advanced data layouts as well as working on a subset of the whole data, allowing for an efficient implementation.

To perform the Fourier transform simultaneously on CPU host and GPU device, we split the single multidimensional transform into a series of 1D transforms along each dimension of the input data, which can be computed on CPU and GPU independently. This approach is known as the *row-column* algorithm [65], and is often used in FFT libraries. The essence of this algorithm can best be summarized in an example of a 2D FFT. Given a matrix `Nx` $\times$ `Ny`, we compute the DFT in the following way:

1. Transfer portion of the input array, decomposed along the x direction, to device memory. We transfer last `gpuNx` $\times$ `Ny` consecutive array elements to GPU. This can be done as a single memory copy operation due to the flat allocation that was used. The choice whether to offload data to GPU memory from the beginning or the end of the input array is arbitrary.

2. Perform DFT along the y direction on both CPU and GPU concurrently. CPU will perform `cpuNx` such transformations, while the GPU will perform `gpuNx` 1D DFTs. Each of these transforms will take a subset of the input array, which are `Ny / 2 + 1` elements apart. Note that CPU will not do anything with the last `gpuNx` $\times$ `Ny` elements.

3. Copy the array with the transform back from GPU to CPU, writing over the stale data with the relevant portion transformed on GPU. After this step, we have the complete array transformed along the y direction residing in host memory.

4. Transfer portion of the input array, decomposed along the y direction, to GPU memory. Similarly to step 1, we transfer last `Nx` $\times$ `gpuNy` elements to GPU.

5. Perform DFT along the x direction on both CPU and GPU. This time, CPU will perform `cpuNy / 2 + 1` such transformations, while GPU will perform remaining `gpuNy / 2` transformations. The halving of the number of transformations is due to use of R2C transformations. In each transform, elements are `Ny / 2 + 1` places apart, while the first element of each transform is adjacent to the previous one.

6. Copy the array with the transform back from GPU to CPU. With this step completed, we have the full FFT of the input array residing in host memory. This step may be omitted if the computation that follows does not require the FFT of input data to be fully assembled in host memory. This is the case in our algorithm, as we use the resulting transformed array only for subsequent computation on GPU. Since the memory transfer is expensive, omitting this step leads to significant performance improvement.

The inverse Fourier transform can be done in an analogous way. In 3D, the general principle remains the same, however, we do not separate the DFT into three 1D transforms, but to one 2D and one 1D transform. We do this due to better performance of 2D transforms, which in both libraries used is found to be better than two 1D transforms. Figure 5.3 illustrates the DFT algorithm described above.

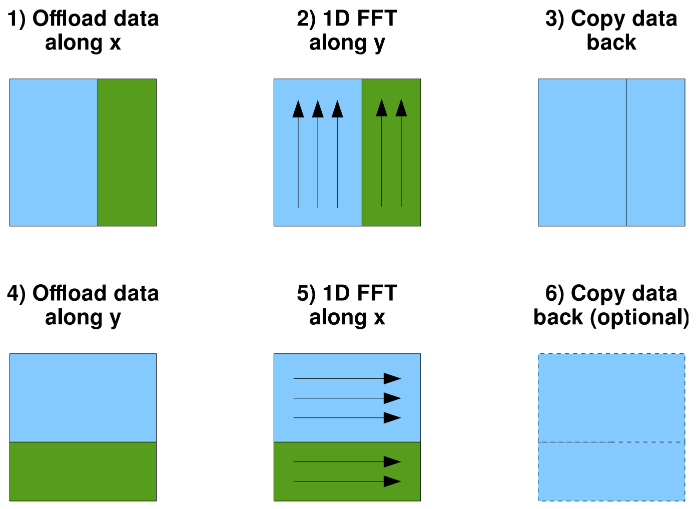


Figure 5.3: Hybrid algorithm for concurrent FFT on host and device.

5.2 Implementation of the hybrid algorithm

The hybrid algorithm was implemented by extending the C/OpenMP implementation with the data distribution scheme from the previous section, while the computation functions on the GPU device were taken from the CUDA implementation described in Section 4.2. This allowed us to reuse much of the existing code, with straightforward modifications.

Decomposition of data was implemented efficiently using the CUDA built-in functionality for copying memory to and from the GPU. In case of decomposition along the x direction, we divide the data into two (possibly uneven) parts with sizes `cpuNx` and `gpuNx`, where `cpuNx + gpuNx = Nx`. Now the CPU works with the first `cpuNx` $\times$ `Ny` elements in 2D programs and `cpuNx` $\times$ `Ny` $\times$ `Nz` elements in 3D programs, while the GPU works with remaining `gpuNx` $\times$ `Ny` elements in 2D programs and `cpuNx` $\times$ `Ny` $\times$ `Nz` elements in 3D programs. Listing 5.1 shows how the memory is copied from host to device in this scenario. The opposite, copying from device to host, is done in a similar way, where only the source and destination pointers are exchanged in the copy parameters.

Listing 5.1: Initialization and copying of data from host to device, distributed along the x direction.

```
// Allocate host data
double *h_matrix = alloc_double_matrix(Nx, Ny);

// Allocate only the required amount of data on device
struct cudaPitchedPtr d_matrix_x = alloc_double_matrix_device(gpuNx, Ny);

// Prepare memcpy operation
struct cudaMemcpy3DParms cpy_x_h2d;
cpy_x_h2d.srcPtr = make_cudaPitchedPtr(h_matrix[cpuNx], Ny * sizeof(double), Ny, gpuNx);
```

```
cpy_x_h2d.dstPtr = d_matrix_x;
cpy_x_h2d.extent = make_cudaExtent(Ny * sizeof(double), gpuNx, 1);
cpy_x_h2d.kind = cudaMemcpyHostToDevice;

// Copy data from host to device
cudaMemcpy3D(&cpy_x_h2d);
```

Similar concept is applied when distributing data along the y direction. This time CPU works with `Nx` $\times$ `cpuNy` in 2D (`Nx` $\times$ `cpuNy` $\times$ `Nz` in 3D), while the GPU portion is `Nx` $\times$ `gpuNy` in 2D, `Nx` $\times$ `gpuNy` $\times$ `Nz` in 3D (Listing 5.2).

Listing 5.2: Initialization and copying of data from host to device, distributed along the y direction.

```
// Allocate only the required amount of data on device
struct cudaPitchedPtr d_matrix_y = alloc_double_matrix_device(Nx, gpuNy);

// Prepare memcpy operation
struct cudaMemcpy3DParms cpy_y_h2d;
cpy_y_h2d.srcPtr = make_cudaPitchedPtr(h_matrix[0], Ny * sizeof(double), Ny, Nx);
cpy_y_h2d.srcPos = make_cudaPos((cpuNy) * sizeof(double), 0, 0);
cpy_y_h2d.dstPtr = d_matrix_y;
cpy_y_h2d.extent = make_cudaExtent(gpuNy * sizeof(double), Nx, 1);
cpy_y_h2d.kind = cudaMemcpyHostToDevice;

// Copy data from host to device
cudaMemcpy3D(&cpy_y_h2d);
```

We can also notice how the flat allocation of memory on CPU is once again useful. Without it, the transfer of data between CPU and GPU would be much more complicated, involving copying individual rows of the innermost dimension. Flat allocation allows for a single copy operation to be invoked to copy the whole GPU portion of the data. It is also much easier to further divide the copy operation when using CUDA streams, as we will demonstrate in Section 5.3.

With the data transfer mechanism in place, we can focus on enabling concurrent computation. CUDA kernel launches are asynchronous with regard to the CPU, allowing us to combine them with OpenMP-based computation on CPU. There are numerous ways to implement GPU offloading. We have investigated three possible approaches, which we now describe.

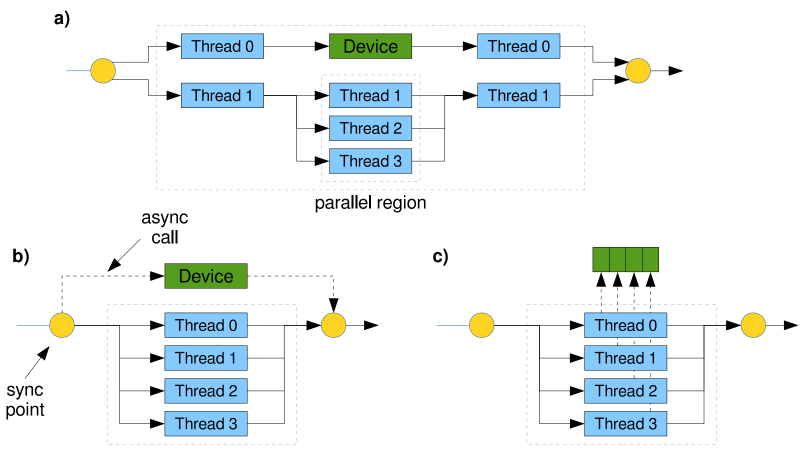


Figure 5.4: Three strategies considered for GPU offloading.

The approach we decided to use, illustrated on Figure 5.4(a), involves the use of OpenMP nested parallelism. We first create a parallel region that consists of two threads, one of which will invoke the GPU functions, while the other will do the same for CPU functions. The GPU-related thread invokes the required functions and waits for their completion. Meanwhile, the CPU-related thread spawns new (nested) threads, which perform the CPU computation with the changed loop exit condition (Listing 5.3). The optimal number of newly spawned threads is equal to `nthreads - 1`, where `nthreads` is the total number CPU cores in the system.

Listing 5.3: Use of nested OpenMP parallelism to control both host and device computation.

```
void hybrid_function(...) {
...
   #pragma omp parallel private(threadid) num_threads(2)
   {
      threadid = omp_get_thread_num();

      if (threadid == 0) {
         gpu_function(); // Async calls on device
         sync_with_gpu(); // Wait for completion
      }

      if (threadid == 1 || omp_get_num_threads() != 2) {
         cpu_function(...); // Invokes nested parallelism
      }
   }
...
}

void cpu_function(...) {
...
   nthreads = omp_get_num_threads();

   #pragma omp parallel for private(...) num_threads(nthreads - 1)
   for (...) { ... }
...
}
```

The second approach, illustrated in Figure 5.4(b), does not use nested parallelism. Instead of dedicating one thread to GPU-related tasks, here we asynchronously invoke GPU functions from the master thread. Before performing computation on CPU using OpenMP threads, we instruct the GPU to perform the work from the master CPU thread. Next we invoke the CPU computation via OpenMP threads (one for each CPU core), and finally, after CPU completes its computation, we synchronize with the GPU to avoid race conditions (Listing 5.4). This approach works because GPU supports asynchronous calls. Kernel invocations are naturally asynchronous with regard to the CPU, while the memory transfer calls can be asynchronous, a feature which will be explained in detail in Section 5.3. While simpler to implement, this solution performs worse than aforementioned implementation with a dedicated GPU thread. This is because the asynchronous GPU calls still require some CPU work, and the frequent context switches the operating system has to make to service the GPU calls negatively affects performance of OpenMP threads.

Final approach we considered involves using each OpenMP thread for both CPU and GPU functions. This way, each OpenMP thread was responsible for transferring optimal amount of data to GPU and back, and invoking kernels on GPU, as illustrated in Figure 5.4(c). This approach did

Listing 5.4: Use of asynchronous calls to device functions while using OpenMP for computation on host.

```
void hybrid_function(...) {
...
   gpu_function();

   #pragma omp parallel private(threadid, ...)
   {
      threadid = omp_get_thread_num();

      #pragma omp for
      for (...) { ... }
   }

   sync_with_gpu();
...
}
```

not bring observable performance improvement and led to a much more complex implementation, requiring more bookkeeping, especially with the introduction of multi-GPU support. Additionally, the flexibility of choosing an ideal amount of data to offload to GPU for each function was somewhat lost because the amount of data offloaded to GPU was now controlled by the number of active OpenMP threads. The introduction of per-thread default streams in CUDA 7.0 (not available at the time of development) could have been used to simplify the implementation, however it would have still been more complex than our other approach, and would not solve the flexibility issue. For these reasons we have discarded this approach.

At this point, we have an implementation that performs most of the computation simultaneously on CPU and GPU, but we also need to implement hybrid FFT routines. The hybrid row-column algorithm from the previous section for DFT can be implemented using existing libraries very efficiently, without the requirement to make the input data of every transform consecutive in memory. The only complexity of this approach is that we have to use the *advanced* interface of FFTW and cuFFT that allows setting appropriate offsets, strides and number of transforms manually. We can see how this works in Listing 5.5, which demonstrates how a 3D FFT can be divided between host and device. This approach does not require more memory than regular multidimensional CPU or GPU FFT. We have tested several combinations of in-place and out-of-place transforms, and have concluded that on the device, the transform along the x direction (when data is distributed along the y direction) performs much better when done out-of-place, while for all other transforms the difference between in-place and out-of-place operation is usually not significant. In our case, switching from one type of transform to another is easy, the only change needed is to allocate memory for the output array, and pass its pointer to the planner.

Listing 5.5: Creating and executing plans for concurrent FFT on host and device.

```
// Allocate proper amount of data on host
double ***tensor = alloc_double_tensor(Nx, Ny, Nz);
fftw_complex *fft_array = alloc_complex_vector(Nx * Ny * (Nz/2+1));

// Using FFTW's 'advanced' interface on host
int fft_rank = 2; int nfr[] = {Ny, Nz}; int howmany = cpuNx;
int idist = Ny * (Nz/2+1) * 2; int odist = Ny * (Nz/2+1);
int istride = 1; int ostride = 1;
int *inembed = NULL, *onembed = NULL;
// Define an out-of-place plan for cpuNx transforms of size Ny * (Nz/2+1)
plan_fw_row = fftw_plan_many_dft_r2c(fft_rank, nfr, howmany, **tensor, inembed, istride,
    idist, fft_array, onembed, ostride, odist, FFTW_MEASURE);

fft_rank = 1; int nfc[] = {Nx}; howmany = cpuNy * (Nz/2 + 1);
idist = 1; odist = 1;
istride = Ny * (Nz/2+1); ostride = Ny * (Nz/2 + 1);
// Define an in-place plan for cpuNy * (Nz/2 + 1) transforms of size Nx
plan_fw_col = fftw_plan_many_dft(fft_rank, nfc, howmany, fft_array, inembed, istride,
    idist, fft_array, onembed, ostride, odist, FFTW_FORWARD, FFTW_MEASURE);
...
// Plans can be executed with the following command
fftw_execute_dft_r2c(plan_fw_row, **tensor, fft_array);
fftw_execute_dft(plan_fw_col, fft_array, fft_array);

// Allocate proper amount of data on device
struct cudaPitchedPtr d_tensor_x = alloc_double_tensor_device(gpuNx, Ny, (Nz/2+1)*2);
struct cudaPitchedPtr d_tensor_y = alloc_double_tensor_device(Nx, gpuNy, (Nz/2+1)*2);
struct cudaPitchedPtr d_tensor_y_tran = alloc_double_tensor_device(Nx, gpuNy, (Nz/2+1)*2);

// Using equivalent cuFFT calls on device
fft_rank = 2; int nfr[] = {Ny, Nz}; howmany = gpuNx;
idist = Ny * (d_tensor_x.pitch / sizeof(cufftDoubleReal));
odist = Ny * (d_tensor_x.pitch / sizeof(cufftDoubleComplex));
istride = 1; ostride = 1;
int inembed_fwr[] = {Ny, d_tensor_x.pitch / sizeof(cufftDoubleReal)};
int onembed_fwr[] = {Ny, d_tensor_x.pitch / sizeof(cufftDoubleComplex)};
// Define an in-place plan for gpuNx transforms of size Ny * (Nz/2+1), padded by 'pitch'
cufftMakePlanMany(plan_fw_row, fft_rank, nfr, inembed_fwr, istride, idist, onembed_fwr,
    ostride, odist, CUFFT_D2Z, howmany, &ws_fwr);

fft_rank = 1; int nfc[] = {Nx}; howmany = gpuNy * (Nz/2+1);
idist = 1; odist = 1;
istride = gpuNy * (d_tensor_y.pitch / sizeof(cufftDoubleComplex));
ostride = gpuNy * (d_tensor_y_tran.pitch / sizeof(cufftDoubleComplex));
int inembed_fc[] = {Nx}; int onembed_fc[] = {Nx};
// Define an out-of-place plan for gpuNy * (Nz/2+1) transforms of size Nx,padded by 'pitch'
cufftMakePlanMany(plan_fw_col, fft_rank, nfc, inembed_fc, istride, idist, onembed_fc,
    ostride, odist, CUFFT_Z2Z, howmany, &ws_fwc);
...
// Plans can be executed with the following command
cufftExecD2Z(plan_fw_row, (cufftDoubleReal *) d_tensor_x.ptr, (cufftDoubleComplex *)
    d_tensor_x.ptr)
cufftExecZ2Z(plan_fw_col, d_tensor_y.ptr, d_tensor_y_tran.ptr, CUFFT_FORWARD);
```

Implementation described so far utilizes all available CPU cores, but only a single GPU. Since a computer may have multiple GPUs installed, we have implemented support for using all available GPUs. Currently, CUDA supports up to four GPUs in a single computer, meaning we can offload up to four times the amount of data to GPUs. We have tested the programs on a computer with up to three GPUs, maximum allowed by the hardware at PARADOX supercomputing facility.

CUDA provides a simple API for choosing which GPU is in use, via the `cudaSetDevice` function. Every command issued afterward targets the selected device. The developer only needs to keep track of pointers used for each device. This simple mechanism allows us to keep our data transfer routines intact, and add support for multiple GPUs with little effort. The general concept we used is to loop over available GPUs, and issue the desired operations on each device (Listing 5.6).

Listing 5.6: Changing the currently active GPU device.

```
cudaGetDeviceCount(&nDevices);

for (i = 0; i < nDevices; i ++) {
   cudaSetDevice(i);
   ... // Issue commands as usual
}
```

This approach has only one drawback, and that is that it will issue the same amount of work to all GPUs, which may not be optimal if GPUs are different. We believe that the trade-off between the simplicity of implementation and feature-completeness was worthwhile, because in most HPC installations, all GPUs within a single computing node are of the same model. Alternatively, we could have also made our programs aware of the differences between multiple GPUs installed. CUDA has an API for discovering features of GPUs which would be useful in selecting the optimal parameters for the amount of work for each CPU, however it would still be a significant effort to implement this in our programs, and for the users to optimally use this functionality.

During implementation of the algorithm described in this section we noticed that the majority of the execution time on GPU is spent waiting for the memory transfers to complete. To perform a single multidimensional FFT, we need four memory transfers, which, if implemented in a sequential manner, becomes the main bottleneck. In the next section we describe a technique to mitigate this.

5.3 Optimization of data transfers

Transferring memory is usually a significantly slower process than computation, to such an extent that if memory copies between host and device are common, that defeats the whole purpose of using GPU. In our implementation, several copies are needed during one iteration of the wave function propagation, limiting the use of GPU to smaller mesh sizes.

There are several remedies to this problem. An obvious one is to reduce the amount of data transferred, by only transferring subsets of data that are required on the GPU side. The programs presented here minimize the amount of data involved by copying only the parts of arrays required by the computation that immediately follows. Although this technique brings observable improvement, it alone is not enough to consider the problem solved. In fact, reducing the amount of data that is copied to GPU and back also means that we use less GPU memory, allowing for use of larger meshes on GPU, which in turn take more time to transfer. Therefore, we rely on additional optimizations in order to improve memory transfer.

To be able to do this, we have to understand and exploit the nature of data transfers between CPU and GPU. Data allocations on the CPU side are *pageable* by default, and the operating system can swap-out memory pages at any point. The GPU cannot access data directly from pageable CPU memory, so when a data transfer from pageable host memory to GPU memory is invoked, the CUDA driver must first allocate a temporary page-locked, or *pinned*, host array, copy the host data to the pinned array, and then transfer the data from the pinned array to device memory.

As can be seen in Figure 5.5, pinned memory is used as a staging area for transfers. We can avoid the cost of the transfer between pageable and pinned host arrays either by directly allocating CPU arrays in pinned memory, or by declaring that already allocated array should be page-locked, or pinned. Pinned memory has its downsides: if the operating system cannot swap-out pages, it

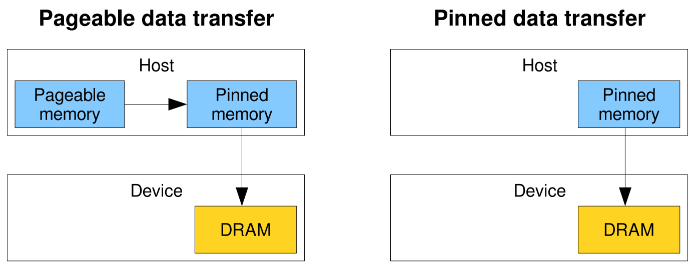


Figure 5.5: Memory copy between host and device using pageable (left) and pinned (right) memory.

may run out of resources, causing it to behave in an unexpected way. Consequently, pinned memory should not be overused. Since our programs transfer only subsets of whole arrays, pinning just those parts allows us to benefit from improved transfer speeds, as well as leave the rest of the memory allocated as pageable, in case the operating system wishes to swap it out.

Even with the use of pinned memory, the data transfer times are still large in comparison to the computation time, and leave the GPU idle during transfer. Most CUDA GPUs support simultaneous data transfers and computation via CUDA *streams*, which are sequences of operations that execute on the device in order by which they are issued by the host code. While operations within a stream are guaranteed to execute in the prescribed order, operations in different streams can be interleaved and, when possible, they can even run concurrently. Proper use of streams allows us to overlap data transfers and computation, thus mitigating the slow transfer speeds. Streams are actually always used internally within CUDA runtime, and unless a stream is specified explicitly, all operations are queued to the default stream where they appear to be synchronous due to their order of execution.

Streams are relatively simple to use when there are multiple operations on disjoint arrays queued for execution. However, when there is just one transfer to GPU, followed by computation on that data and transfer of data back from GPU, streams alone do not offer a way to improve performance. Therefore, to obtain speedup we divide the data into smaller chunks, which can then be transferred and processed independently (Figure 5.6). This idea allows us to reduce the idle time of GPU, from the time it takes to transfer the whole array to GPU and back, to the time it takes to transfer just one chunk to GPU and back, which can be significant.

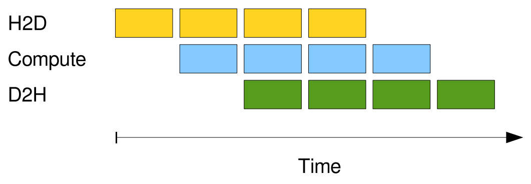


Figure 5.6: Dividing data into smaller chunks and transferring them using streams reduces the GPU idle time. The idle time is reduced to the time it takes to complete one host to device (H2D) copy operation and one device to host (D2H) copy operation.

The overlap illustrated in Figure 5.6 is possible only on certain CUDA GPUs. In general, the overlap depends on the order the operations are issued to a stream, and the number of *copy engines* that the GPU has. Operations in each engine's queue are executed in the order they are issued. Therefore, to queue all operations, we have two choices, shown in Listing 5.7.

Listing 5.7: Two approaches to issuing commands to CUDA streams.

```
h_data = ... // Allocate data on host
d_data = ... // Allocate data on device
str = ... // Initialize 'nStreams' CUDA streams

// Asynchronous copy v1
for (i = 0; i < nStreams; i++) {
  int offset = i * streamSize;
  cudaMemcpyAsync(&d_data[offset], &h_data[offset], size, cudaMemcpyHostToDevice, str[i]);
  kernel<<<gridSize, blockSize, 0, str[i]>>>(d_data, offset);
  cudaMemcpyAsync(&h_data[offset], &d_data[offset], size, cudaMemcpyDeviceToHost, str[i]);
}

// Asynchronous copy v2
for (i = 0; i < nStreams; i++) {
  int offset = i * streamSize;
  cudaMemcpyAsync(&d_data[offset], &h_data[offset], size, cudaMemcpyHostToDevice, str[i]);
}
for (i = 0; i < nStreams; i++) {
  int offset = i * streamSize;
  kernel<<<gridSize, blockSize, 0, str[i]>>>(d_data, offset);
}
for (i = 0; i < nStreams; i++) {
  int offset = i * streamSize;
  cudaMemcpyAsync(&h_data[offset], &d_data[offset], size, cudaMemcpyDeviceToHost, str[i]);
}
```

Depending on the number of copy engines, overlaps may be different, or not exist at all. If a GPU has two copy engines, one for transferring to and one for transferring back from the GPU, the overlap obtained by the first approach is ideal, while the second one is not. If there is one copy engine, the situation is different, with the second approach performing better. This is shown on Figure 5.7.

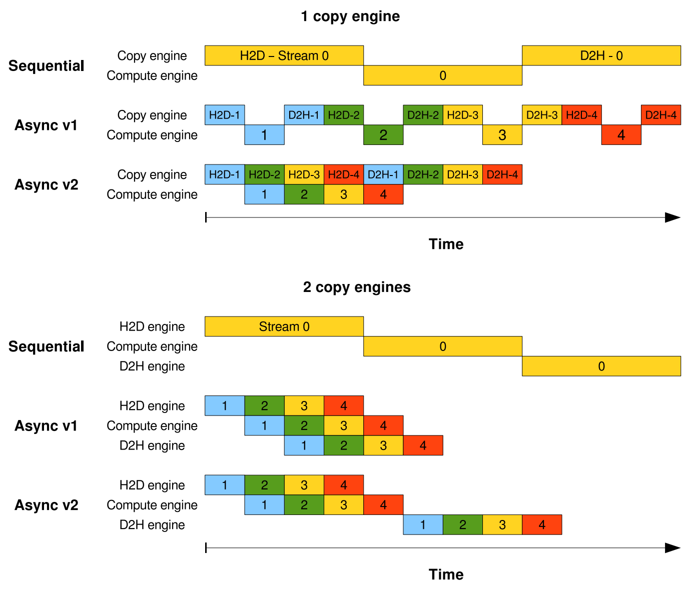


Figure 5.7: Difference between one and two copy engines for overlapping data transfer and computation.

Nvidia Tesla line of GPUs usually has two copy engines, while the consumer-oriented GeForce line has just one [66]. On newer Nvidia GPUs with Compute Capability 3.5 and higher this is not a problem, so both approaches have equal performance. However, we decided to aim for maximum compatibility and implemented both approaches, and our programs themselves select the correct approach at runtime. We created separate functions for different cases, and a function pointer to select between the two. Number of copy engines (among other things) for CUDA device in use can be obtained from CUDA runtime. With this piece of information programs choose the correct functions during initialization.

In Figures 5.6 and 5.7 we assume that the time required for transfer and computation is approximately the same. This, however, is not necessarily easy to achieve, as it depends on the size and number of chunks that the GPU will process. Users can control the size and number of chunks in our programs, and a detailed guide to choosing the best parameters is given in Chapter 8.

Chapter 6

Distributed memory algorithm

Algorithms presented in previous chapters target a single computer, where they use all available CPU and/or GPU resources. Next logical step is to extend them to multiple machines, i.e., clusters of computing nodes with distributed memory model, where no single process contains the whole dataset. Using multiple computing nodes can help us achieve two goals, namely faster computation on existing mesh sizes due to more computing resources being available, and availability of larger mesh sizes that cannot fit on a single computing node. In a distributed memory system, we often need to exchange data between processes running in parallel on different computing nodes. Explicit message passing is typically deployed to facilitate data exchanges and achieve parallelism.

The most popular standardized protocol for message passing paradigm is the Message Passing Interface (MPI) [67], which we use in our implementations targeting distributed memory clusters. For this development, focus was placed only on the 3D variants of our programs. We made this decision because modern computers (especially powerful cluster nodes) have enough memory for 1D or 2D calculations our programs require. An MPI algorithm with three different implementations came out as a result of this effort. The three implementations are:

1. pure CPU version built on top of shared memory algorithm from Chapter 3,
2. pure GPU version built on top of CUDA implementation of the shared memory algorithm described in Chapter 4 and
3. hybrid version based on the algorithm described in Chapter 5.

Henceforth, we refer to these implementations as OpenMP/MPI, CUDA/MPI and Hybrid/MPI, respectively. Two of the three implementations, the OpenMP/MPI and CUDA/MPI implementations, have been published in Ref. [18] and is available for download [46, 37]. The Hybrid/MPI implementation is also publicly available [37].

In the following sections, we demonstrate how the algorithms from previous chapters can be extended with a data distribution scheme to enable their execution on distributed memory systems, and how MPI can be successfully used in the implementation of these new algorithms. All three implementations we developed share a common data distribution scheme, and thus share the same general structure, so we describe them together in this chapter, emphasizing the differences in implementations where applicable. We start with the description of data decomposition among the nodes in the cluster (Section 6.1), and how the necessary transpose of data can be implemented.

In Section 6.2 we move on to the description of the changes in computation imposed by the distributed environment. Finally, in Section 6.3, we describe the changes needed to input and output components in the new environment.

6.1 Data distribution scheme

When we design algorithms for distributed memory systems there are two general approaches we can take. We can either focus our efforts on developing distributed algorithms, or we can dynamically redistribute the data by doing a transpose among processes in order to apply our existing algorithms on each computing node.

In the first approach, we would have to develop many novel algorithms from scratch, such as a parallel tridiagonal solver used in the functions `calclux`, `calcluy` and `calcluz`, or a parallel Fourier transform working on distributed data, and in the case of a heterogeneous architecture, further divided between CPU and GPU. This would require a significant effort, and while in the end it may provide significant performance improvement, the complexity of the implementation would be detrimental to future modifications.

The other approach, to redistribute the data, requires only development of transpose routines necessary to transform the data so that any desired dimension is local to a process. Thus, the second approach is preferred here due to its simplicity. The existing algorithms, optimized for a single machine, remain mostly unchanged and preserve all of the original logic. Conceptually, the most important changes are that we now initialize only the relevant portion of data on each process, and rely on data transposes to bring the nonlocal data to each process.

Working with 3D data allows us to have a 1D, 2D or 3D decomposition, as shown in Figure 6.1. In 1D or 2D decompositions we can arbitrarily choose the dimensions which would be split among the processes. The 3D decomposition would imply that no single dimension is local to any process, a scenario that would require us to develop distributed algorithms from scratch, which we do not pursue.

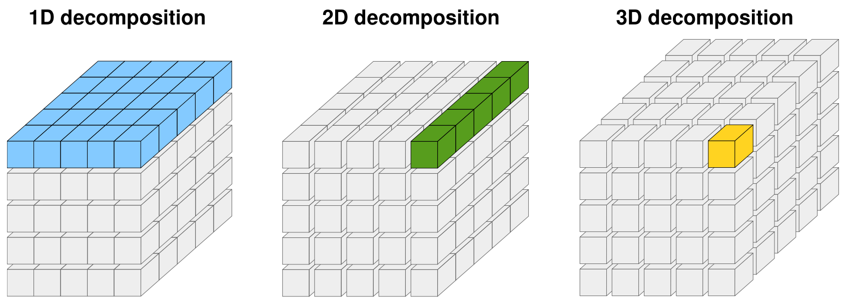


Figure 6.1: Decomposition of 3D data along one (left), two (center) and three (right) dimensions.

The 1D decomposition (slab decomposition, Figure 6.1, left), which we have already used as the data distribution method in our hybrid algorithm, is the simplest way to distribute the data among separate processes here as well. In this decomposition approach, each MPI process is assigned one or more 2D slices of data s shown in the figure. The downside of this type of decomposition is its scalability. In order to perform a transpose over slab-decomposed data, all processes communicate to must exchange data, which in cases of large number of processes (over 1000), can lead to scalability issues [68]. On the other hand, in 2D decomposition, also known as block or pencil decomposition, MPI processes are organized in a 2D array and each contains one or more 1D blocks of data as shown in Figure 6.1 (center). The block decomposition requires only between processes in a single logical row or column, which scales better.

Transposing data in both of these decomposition types can be implemented using all-to-all MPI functions, e.g., `MPI_Alltoall` or `MPI_Alltoallv`. However, 2D decomposition is more difficult to implement since the communication pattern is quite complex [69], e.g., the cost of communications is very sensitive to the orientations of blocks and their associated memory patterns. The packing and unpacking of memory buffers for the MPI library must be handled with great care for efficiency. All this is further complicated by the fact that we need two such decompositions, depending on the dimension which we need to be local to each process. In the implementation of 1D decomposition, there is an alternative way to transpose the data, via the FFTW library, whose implementation of transpose routines relies on internal mechanisms used in the computation of multidimensional FFT. This approach is simpler and has an additional benefit in that FFTW routines can operate in-place, whereas `MPI_Alltoall`-based approach can only operate out-of-place. Even for out-of-place transposes, FFTW's routine should be equal or faster, since one of the possible algorithms that FFTW uses for an out-of-place transpose is simply to call `MPI_Alltoall`.

Since PARADOX cluster used during the development has slightly more than a hundred nodes (with a total of 1696 cores), we have decided to base our algorithm and its implementations on the 1D decomposition. In order to avoid its scalability issues for large number of processes, we use MPI only for communication between nodes in a cluster, whereas all computation within a single node remains unchanged, with OpenMP, CUDA, or a combination of the two. Similarly to approach used in our hybrid algorithm, here we also decompose along the x and y direction, according to data access patterns, an example of which we show in the next section. This means that each process holds `localNx` × `Ny` × `Nz` amount of data (or `Nx` × `localNy` × `Nz`), where `localNx = Nx / nprocs` and `nprocs` is the number of MPI processes (`localNy = Ny / nprocs`). Transpose along the innermost, fastest changing dimension is not required by any computation.

With the data distributed among the MPI processes, we need an efficient transpose routine to redistribute the data along a different dimension. As mentioned before, one easy way to achieve this is to use the transpose routines from the FFTW library, which we use in the OpenMP/MPI and Hybrid/MPI implementations. Transpose interface in FFTW works in a similar way as the rest of the library, i.e., it involves creating end executing a plan (Listing 6.1). We use a special type of transpose routine from FFTW, where data is *locally* transposed. The full transpose of data would require performing an additional, local transpose operation, which we omit since we can perform all computation with locally transposed data. This means that the data after such partial transpose operation are stored structured as `Nx` × `localNy` × `Nz` in row-major order, instead of the expected `localNy` × `Nx` × `Nz` structure in row-major order. Similar procedure can be used to transpose the data back to its original layout. This way, we do not have to change the data access pattern, we only have to adjust the loop limits. To instruct FFTW to perform transpose in this way, we have to pass an additional flag when creating a transpose plan.

While it is very easy to use, FFTW transpose interface currently works only on data stored in host memory of each computing node, not the GPU memory. As a consequence, we had to implement our own transpose routines for CUDA/MPI version. We used the MPI data types and

Listing 6.1: The transpose operation implemented using FFTW, using locally transposed data layout.

```
// Allocate required memory
double ***tensor = alloc_double_tensor(localNx, Ny, Nz);
double ***tensor_t = alloc_double_tensor(Nx, localNy, Nz);

// Create a plan with a special flag specifying that output should be locally transposed
fftw_plan plan_tran_x = fftw_mpi_plan_many_transpose(Nx, Ny * Nz, 1, localNx, localNy *
    Nz, **tensor, **tensor_t, MPI_COMM_WORLD, FFT_MEASURE | FFTW_MPI_TRANSPOSED_OUT);
fftw_plan plan_tran_y = fftw_mpi_plan_many_transpose(Ny * Nz, Nx, 1, localNy * Nz,
    localNx, **tensor_t, **tensor, MPI_COMM_WORLD, FFT_MEASURE | FFTW_MPI_TRANSPOSED_IN);

//Execute like a regular FFTW plan
fftw_execute(plan_tran_x);
fftw_execute(plan_tran_y);
```

all-to-all communication (by using `MPI_Alltoall` function or a series of `MPI_Isend` and `MPI_Irecv` functions) to perform this task. We implemented the same locally transposed input/output data layout. Figure 6.2 shows the communication pattern required to perform such transpose operation. Since the result of the transpose is the same with FFTW and our own implementation, we made available the choice between the two transpose routines as a compile-time variable in Hybrid/MPI implementation, allowing users to select routine to be used.

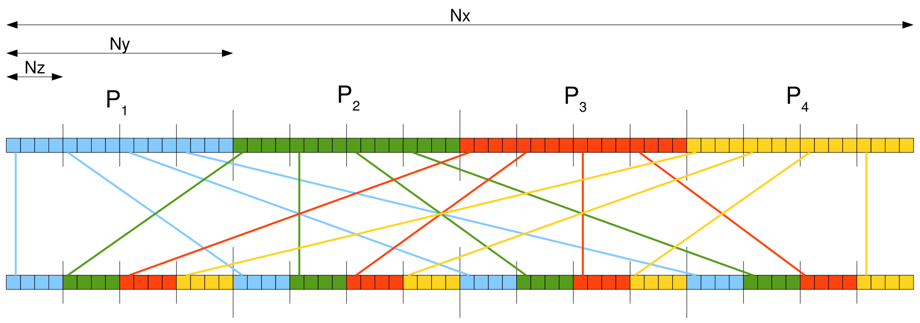


Figure 6.2: Communication pattern of transpose operation between four processes. Note that the data after the operation are locally transposed, i.e., do not represent full transpose in general case.

Two custom MPI data types were required, representing a portion of the data which is to be exchanged in x and y dimension. Special care must be taken to account for the third dimension, which is padded in CUDA implementation and should thus be expressed in bytes and not in the actual number of elements. To achieve this, we use MPI vector type, as described in Listing 6.2.

Listing 6.2: The transpose operation implemented using MPI, using locally transposed data layout.

```
struct tran_params { // We use a structure to keep the data relevant to transpose
   void *orig_buf, *tran_buf; // Data buffers
   int *orig_cnts, *tran_cnts; // Counts
   int *orig_displ, *tran_displ; // Displacements
   MPI_Datatype *orig_types, *tran_types; // Custom MPI types
   int nprocs; // Number of processes
   MPI_Request *send_req, *recv_req; // Used to track async sends and recv's
};
```

```
struct tran_params init_transpose_double(int nprocs, long localNx, long localNy, long Ny,
    long Nz, void *send_buf, size_t send_pitch, void *recv_buf, size_t recv_pitch) {
   long i;
   MPI_Datatype tran_dtype;
   MPI_Datatype lxyz_dtype, xlyz_dtype;
   struct tran_params tran;

   // Create a custom data type representing a Ny * Nz matrix, pitched, in bytes
   MPI_Type_contiguous(localNy * (send_pitch / sizeof(double)), MPI_DOUBLE, &tran_dtype);
   MPI_Type_commit(&tran_dtype);

   // MPI vector representing the data to be sent to each process
   MPI_Type_create_hvector(localNx, 1, Ny * send_pitch, tran_dtype, &lxyz_dtype);
   MPI_Type_create_hvector(localNx, 1, localNy * recv_pitch, tran_dtype, &xlyz_dtype);
   MPI_Type_commit(&lxyz_dtype);
   MPI_Type_commit(&xlyz_dtype);

   ... // Allocate tran_params variables (*_cnts, *_displ, *_types, *_req)

   // Set the parameters for each process
   for (i = 0; i < nprocs; i ++) {
      orig_cnts[i] = 1;
      tran_cnts[i] = 1;

      orig_displ[i] = i * localNy * send_pitch;
      tran_displ[i] = i * localNx * localNy * recv_pitch;

      orig_types[i] = lxyz_dtype;
      tran_types[i] = xlyz_dtype;
   }

   return tran;
}

void transpose(struct tran_params tran) {
   long i;

   // Exchange data using asynchronous send and receive
   for (i = 0; i < tran.nprocs; i++) {
      MPI_Irecv(((char *)tran.tran_buf) + tran.tran_displ[i], 1, tran.tran_types[i], i, 0,
          MPI_COMM_WORLD, &(tran.recv_req[i]));
      MPI_Isend(((char *)tran.orig_buf) + tran.orig_displ[i], 1, tran.orig_types[i], i, 0,
          MPI_COMM_WORLD, &(tran.send_req[i]));
   }
   for (i = 0; i < tran.nprocs; i++) {
      MPI_Wait(&(tran.recv_req[i]), MPI_STATUSES_IGNORE);
      MPI_Wait(&(tran.send_req[i]), MPI_STATUSES_IGNORE);
   }

   // Alternatively, we can use MPI_Alltoallw
   //MPI_Alltoallw(tran.orig_buf, tran.orig_cnts, tran.orig_displ, tran.orig_types,
       tran.tran_buf, tran.tran_cnts, tran.tran_displ, tran.tran_types, MPI_COMM_WORLD);
}

// To initialize the transpose operation
d_tensor = alloc_double_tensor_device(localNx, Ny, Nz);
d_tensor_t = alloc_double_tensor_device(Nx, localNy, Nz);
tran_tensor = init_transpose_double(nprocs, localNx, localNy, Ny, Nz, d_tensor.ptr,
    d_tensor.pitch, d_tensor_t.ptr, d_tensor_t.pitch);
transpose(tran_tran_tensor);
```

An interesting property of the CUDA/MPI implementation is that the transpose routine is invoked on data residing in GPU memory, as opposed to the other two distributed memory implementations. This is possible due to the functionality available in some of the modern, CUDA-aware MPI implementations, where MPI runtime can directly access data in GPU memory without the need for copying to the host memory. Otherwise, we would need to copy the data to host memory before any MPI routine is called, a task known to be a bottleneck, which would make CUDA/MPI implementation far less usable. A list of CUDA-aware MPI implementations is available in Ref. [70]. During testing and development of the three distributed memory implementations, we have used Open MPI library [71], which can be compiled with support for CUDA.

6.2 Distributing the computation over multiple processes

With our data distribution scheme explained in the previous section, we can outline the changes required to the computation functions. We first have to ensure that each process works with a different portion of the data, or rather, initialize equal, non-overlapping portions of data on each process. This requires that the number of discretization points of each dimension that will be distributed must be divisible by the number of processes. By introducing additional processing logic it is possible to support the distribution of non-equal portions of data, however, we decided against such implementation since it leads to unbalanced computing load. Note that this is not a significant restriction because users can always adapt and select slightly larger or smaller, but evenly divisible mesh.

Data initialization begins by calculating the amount of data each process will get. Based on that we calculate each process' logical offset into the data, so we can initialize only the relevant portion. Initially, we distribute the data along the x dimension and perform a transpose operation whenever the data need to be local in that dimension. Listing 6.3 shows how each process can identify and initialize only the portion of data that logically belongs to it.

Listing 6.3: Example of initialization of the relevant portion of data on each process.

```
// Get the rank of the current process and total number of processes
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
// Compute offsets
long localNx = Nx / nprocs; long offsetNx = rank * localNx;
// Allocate memory for data
psi = alloc_double_tensor(localNx, Ny, Nz);
// Example of initialization of relevant portion of wave function
for (i = 0; i < Nx; i ++) {
   x[i] = (i - Nx / 2) * dx; x2[i] = x[i] * x[i];
}
...
for (i = 0; i < localNx; i ++) {
   for (j = 0; j < Ny; j ++) {
      for (k = 0; k < Nz; k ++) {
         tmp = exp(- 0.5 * (vgamma * x2[offsetNx + i] + vnu * y2[j] + vlambda * z2[k]));
         psi[i][j][k] = tmp / cpsi;
      }
   }
}
```

Now that each process has initialized the relevant portion of the data, the only change to the computation functions is the adjustment of the exit clause (limit) of the outermost loop. Computation in the inner loops that require other two dimensions to be local remains unchanged. This means that functions `calcnu`, `calcluy` and `calcluz` do not require any changes other than modifications of the loop limits, as their relevant portion of data is already local. Function `calclux`, which requires x dimension to be local, is changed so that we first transpose the data, afterward perform the computation as before, and then transpose the data back to original layout (Listing 6.4).

Listing 6.4: Pseudocode of the changed `caclux` function.

```
// Using FFTW transpose for OpenMP/MPI implementation
void calclux(...) {
   ...
   fftw_execute(plan_transpose_x); // Transpose 'psi' and place the result in 'psi_t'
   ... // Compute calclux using 'psi_t', with the outer loop limit set at 'localNy'
   fftw_execute(plan_transpose_y); // Transpose 'psi_t' back to original 'psi'
}

// Using custom MPI transpose for Hybrid/MPI and CUDA/MPI implementations
void calclux(...) {
   ...
   transpose(tran_psi);
   ...
   transpose_back(tran_psi);
}
```

This is also a common pattern for other functions that require x dimension to be local, namely, parts of computation of chemical potential and energy, as well as RMS size. These functions also require communication between processes to gather results of computation at *root* process (usually the process with the lowest identification number, i.e., 0), which post-processes them (if needed) and writes the summary output. Listing 6.5 demonstrates this pattern.

Listing 6.5: Pseudocode of the data gathering pattern employed by `cacnorm`, `cacmuen` and `cacrms` functions.

```
void calcnorm(...) {
   ...// Place the temporary data in 'tmpx' as before
   void *sendbuf = (rank == 0) ? MPI_IN_PLACE : *tmpx;
   MPI_Gather(sendbuf, localNx, MPI_DOUBLE, *tmpx, localNx, MPI_DOUBLE, 0, MPI_COMM_WORLD);

   if (rank == 0) {
      *norm = sqrt(simpint(dx, *tmpx, Nx));
   }

   MPI_Bcast(norm, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
   ...
}
```

The final piece is the computation of Fourier transforms in a distributed memory environment. Fourier transform for OpenMP/MPI implementation is a straightforward task, because FFTW library supports MPI, and uses the same slab decomposition. FFTW's MPI transform algorithms work by first computing transforms of the data local to each process, then by globally transposing the data to redistribute them among the processes and performing a transform of the new data local to each process, and optionally transposing back. This sequence corresponds to the *row-column* algorithm, which we have already seen in our hybrid algorithm in Chapter 5. An example of the row-column algorithm in a distributed memory system for 2D Fourier transform is illustrated in Figure 6.3. If the final data transpose is omitted, the transform results are produced in transposed order, distributed over y dimension. This has a significant performance advantage, at the cost of having to change the data layout of the Fourier transform of the dipolar potential (variable `potdd`). This was a simple matter of allocating memory in a different layout to fit the transposed order and reordering the loops in the initialization of the dipolar term.

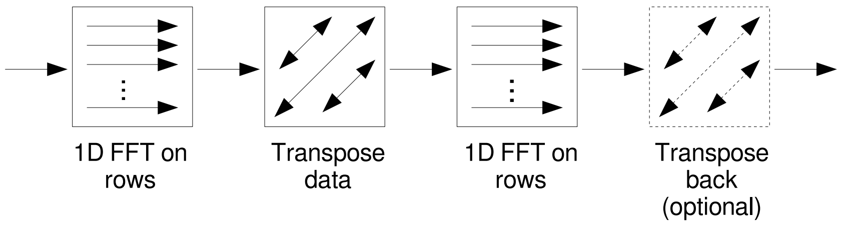


Figure 6.3: Illustration of four stages of the row-column FFT algorithm.

In the case of CUDA/MPI and Hybrid/MPI implementations though, it is not possible to use FFTW's MPI transforms. The cuFFT library does not have support for MPI, and our Hybrid/MPI implementation relies on the heterogeneous Fourier transform we described in Section 5.2. However, we can still use transpose routines in conjunction with the row-column algorithm. To mimic the behavior of FFTW's MPI transforms with transposed order, we rely on the same transpose routines we used earlier, either ones available in FFTW for the Hybrid/MPI implementation, or the ones we developed for the CUDA/MPI version. Listing 6.6 illustrates how we perform FFT using FFTW's MPI transforms, using cuFFT and our own transpose routines CUDA/MPI, as well as using cuFFT and FFTW with FFTW transposes in Hybrid/MPI implementation.

Listing 6.6: Three different implementations of distributed FFT: using FFTW, cuFFT and both. We show only the *forward* FFT, the *inverse* transform that follows is done analogously.

```
// FFT plan creation and usage with FFTW MPI interface, used in OpenMP/MPI implementation
fftw_plan plan_fw = fftw_mpi_plan_many_dft_r2c(3, n, 1, localNx, localNy, psidd2[0][0],
    psidd2fft, MPI_COMM_WORLD, FFT_MEASURE | FFTW_MPI_TRANSPOSED_OUT);
...
fftw_execute(plan_fw); // Executed as any other plan

// FFT with cuFFT , used in CUDA/MPI implementation
int dim = 2;
long long nfr[] = {Ny, Nz};
long long howmany = localNx;
long long idist = Ny * (psidd2.pitch / sizeof(cufftDoubleReal))
long long odist = Ny * (psidd2.pitch / sizeof(cufftDoubleComplex));
long long istride = 1, ostride = 1;
long long inembedfr[] = {Ny, psidd2.pitch / sizeof(cufftDoubleReal)};
long long onembedfr[] = {Ny, psidd2.pitch / sizeof(cufftDoubleComplex)};

cufftMakePlanMany64(plan_fwr, dim, nfr, inembedfr, istride, idist, onembedfr, ostride,
    odist, CUFFT_D2Z, howmany, &ws_fwr);

dim = 1;
long long nfc[] = {Nx};
```

```
howmany = localNy * (Nz/2 + 1);
idist = 1; odist = 1;
istride = localNy * psidd2_t.pitch / sizeof(cufftDoubleComplex);
ostride = localNy * psidd2_t.pitch / sizeof(cufftDoubleComplex);
long long inembedfc[] = {Nx}, onembedfc[] = {Nx};
cufftMakePlanMany64(plan_fwc, dim, nfc, inembedfc, istride, idist, onembedfc, ostride,
    odist, CUFFT_Z2Z, howmany, &ws_fwr);
// Create a transpose plan
tran_dd2 = init_transpose_complex(nprocs, localNx, localNy, Ny, Nz/2 + 1, psidd2.ptr,
    psidd2.pitch, psidd2_t.ptr, psidd2_t.pitch);
...
cufftExecD2Z(plan_fwr, (cufftDoubleReal *) psidd2.ptr, (cufftDoubleComplex *) psidd2.ptr);
transpose(tran_dd2);
cufftExecZ2Z(plan_fwc, (cufftDoubleComplex *) psidd2_t.ptr, (cufftDoubleComplex *)
    psidd2_t.ptr, CUFFT_FORWARD);

// FFT using FFTW on host and cuFFT on device, as used in Hybrid/MPI implementation
... // Initialize FFT plans on host and device as in Listing 5.5
// Make transpose plan
plan_tr_psidd2_x = fftw_mpi_plan_many_transpose(Nx, Ny * (Nz/2 + 1), 2, localNx, localNy *
    (Nz/2 + 1), (double *) psidd2fft, (double *) psidd2fft, MPI_COMM_WORLD, FFTW_ESTIMATE
    | FFTW_MPI_TRANSPOSED_OUT);

#pragma omp parallel private(threadid) num_threads(2)
{
   threadid = omp_get_thread_num();
   if (threadid == 0) {
      ... // Copy data to GPU, perform FFT along Ny and Nz, and sync data back
   }
   if (threadid == 1 || omp_get_num_threads() != 2) {
      ... // Perform FFT on host, using FFTW without MPI
   }
}

fftw_execute(plan_tr_psidd2_x); // Transpose data using FFTW

#pragma omp parallel private(threadid) num_threads(2)
{
   threadid = omp_get_thread_num();
   if (threadid == 0) {
      ... // Copy data to GPU, perform FFT along Nx, without syncing back
   }
   if (threadid == 1 || omp_get_num_threads() != 2) {
      ... // Perform FFT on host, using FFTW without MPI
   }
}
```

The patterns we described in this section are used throughout the three implementations. Since their use is relatively straightforward, for brevity we will not go through all the individual changes.

6.3 Improvements of input and output operations

With the introduction of multiple computing nodes, each working on a separate set of data, the I/O components of our programs needed to be adjusted accordingly. Note that the *input* mentioned in this section is the output of imaginary-time propagation programs, used as input for real-time propagation. Input files containing parameters (passed via `-i` argument during program startup) are very small, and identical for every process, thus there is no need for handling them in any special way.

A naive approach to reading input data in a cluster environment is to let one process read all data, and then distribute them to all other processes. This is inefficient, since it relies on I/O throughput of a single node to perform the read, and therefore scales badly. Additionally, the input may be very large, larger than the available memory of a single node, requiring additional processing logic to read and split the data. As far as output is concerned, a simple approach would be to let each process write its own portion of the data in a separate file, ensuring only that processes do not overwrite each other's output. While the simplicity of this approach to I/O is appealing, it may result in potentially many files being created, which complicates analysis of the results, forcing users to rely on some external process to collect, combine and otherwise process the output. Alternatively, we could send all data to one process, which would then write all output data aggregated. This approach again has scalability issues similar to the naive approach to reading input data, as it would limit the mesh size to the size of the single node's memory. Efforts to remedy this, for example by sequentially obtaining and writing data from other processes would further reduce the performance.

Another limiting factor is the sheer amount of data written. Previous implementations used a textual output, which is simple to analyze and process. With larger mesh sizes, however this output grows to become very large. Textual representation of floating-point values stored in memory that we used before takes up more space than the equivalent binary representation, and we can reduce the size of textual representation only by decreasing precision.

In light of these limitations, we have decided to use MPI for I/O as well, and to switch from textual to binary representation of data. While it is possible to write textual data with MPI, by calculating the offset based on the amount of bytes the text is going to occupy, this solution is not preferred due to the reasons mentioned above. Textual output could still be obtained from binary representation using `hexdump` command, available in most UNIX-like operating systems. We have included an example script that will convert binary output to a textual one as part of the documentation of our MPI programs.

MPI has a simple mechanism for writing to and reading from files based on the offset into data at which each process will perform its work (Figure 6.4). Behind the scenes, MPI runtime will choose an optimal way to perform I/O.

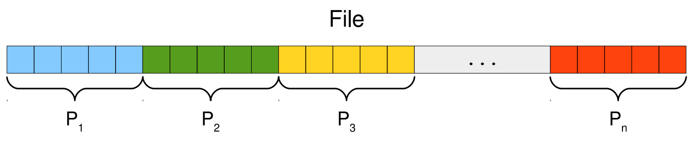


Figure 6.4: I/O using MPI, where each process accesses a portion of the file at a different offset.

Reading or writing data with MPI in binary form is straightforward, as illustrated by several output functions shown in Listing 6.7. We did not use the approach of single-node implementations, where small amount of data is written to a file in each iteration of the loop. Instead, we now use temporary variables to prepare data for output, and write them all at once. In case of very large outputs, like 3D density profiles, MPI buffers may be too small to write all data at once, so in that case we can simply write a 2D slice in a loop. Writing output this way has obvious performance benefits, and the amount of additional memory required for the temporary variables is negligible in a distributed memory system.

Listing 6.7: Examples of output functions implemented using MPI I/O.

```c
// Write integrated XY density
void outdenxy(double ***psi, double ***outxy, double *tmpz, MPI_File file) {
   long i, j, k;
   // Compute the appropriate offset for each process
   MPI_Offset fileoffset= rank * 3 * sizeof(double) * (localNx / outstpx) * (Ny / outstpy);

   for (i = 0; i < localNx; i += outstpx) {
      for (j = 0; j < Ny; j += outstpy) {
         for (k = 0; k < Nz; k ++) {
            tmpz[k] = psi[i][j][k] * psi[i][j][k];
         }
         // Prepare output data
         outxy[i / outstpx][j / outstpy][0] = x[offsetNx + i];
         outxy[i / outstpx][j / outstpy][1] = y[j];
         outxy[i / outstpx][j / outstpy][2] = simpint(dz, tmpz, Nz);
      }
   }
   // Write all prepared data at once, leaving MPI to choose the best strategy for I/O
   MPI_File_write_at_all(file, fileoffset, **outxy, (localNx / outstpx) * (Ny / outstpy) *
       3, MPI_DOUBLE, MPI_STATUS_IGNORE);
}

// Write integrated Y density
void outdeny(double ***psi_t, double **outy, double *tmpx, double *tmpz, MPI_File file) {
   long i, j, k;
   MPI_Offsetfileoffset = rank * 2 * sizeof(double) * (localNy / outstpy);
   fftw_execute(plan_transpose_x); // With a transpose we can make required data local

   for (j = 0; j < localNy; j += outstpy) {
      for (i = 0; i < Nx; i ++) {
         for (k = 0; k < Nz; k ++) {
            tmpz[k] = psi_t[i][j][k] * psi_t[i][j][k];
         }
         tmpx[i] = simpint(dz, tmpz, Nz);
      }
      outy[j / outstpy][0] = y[offsetNy + j];
      outy[j / outstpy][1] = simpint(dx, tmpx, Nx);
   }
   MPI_File_write_at_all(file, fileoffset, *outy, (localNy / outstpy) * 2, MPI_DOUBLE,
       MPI_STATUS_IGNORE);
}

// Write final density
void outdenxyz(double ***psi, double ***outxyz, MPI_File file) {
   long i, j, k;
   MPI_Offsetf ileoffset = rank * sizeof(double) * localNx * Ny * Nz;

   for (i = 0; i < localNx; i += outstpx) {
      for (j = 0; j < Ny; j += outstpy) {
         for (k = 0; k < Nz; k += outstpz) {
            outxyz[0][j][k] = psi[i][j][k] * psi[i][j][k];
         }
      }
      // MPI I/O returns error if the array is too large, so we write Ny * Nz at a time.
      MPI_File_write_at_all(file, fileoffset, **outxyz, (Ny / outstpy) * (Nz / outstpz),
          MPI_DOUBLE, MPI_STATUS_IGNORE);
      fileoffset += (Ny / outstpy) * (Nz / outstpz) * sizeof(double);
   }
}
```

```
// Reading the output of imaginary-time programs in real-time programs
void readdenxyz(double complex ***psi, double *tmpz, MPI_File inputfile) {
   long i, j, k;
   MPI_Offset fileoffset = rank * sizeof(double) * localNx * Ny * Nz;

   for (i = 0; i < localNx; i ++) {
      for (j = 0; j < Ny; j ++) {
         // Read whole row at a time, faster then each element individually
         MPI_File_read_at(file, fileoffset, tmpz, Nz, MPI_DOUBLE, MPI_STATUS_IGNORE);
         for (k = 0; k < Nz; k ++) {
            psi[i][j][k] = sqrt(tmpz[k]);
         }
         fileoffset += Nz * sizeof(double);
      }
   }
   MPI_File_close(&file);
}
```

Note that in some output functions we require data along x direction to be local to each process. We use the same techniques described in the previous two sections to achieve this, therefore we will not go into those details again.

Chapter 7

Interacting with simulation

Programs presented in previous chapters are not interactive. That is, programs are parametrized via an input file and produce a summary output, and optionally write the density profiles of the wave function to disk at predefined intervals. This output can then be analyzed further or visualized with some visualization tool. While this workflow is sufficient for many use-cases, sometimes it is useful to analyze and visualize data as it is being generated, as well as to interact with the simulation (for example, to start, pause, stop, or restart the simulation) and modify parameters it is running. This type of visualization is called *in-situ visualization*, meaning that it is applied on datasets in-place where they are computed (host memory of a computer), rather than writing files to disk, and then reading them back into a visualization tool. The ability to steer the visualized simulation, e.g., by stopping the simulation if we are not satisfied with the intermediate results, or modifying the parameters of a running simulation, can help reduce resource consumption and reduce the time it takes to obtain the desired results. In situ visualization can also be a valuable research tool, allowing users to investigate phenomena that appear during the simulation, even if not anticipated previously.

Creating a visualization component and embedding it in our programs would be an arduous task. The preferred approach therefore is to link a visualization component of an existing tool that supports in-situ visualization to our programs. There are plenty of such tools available, but two have garnered a lot of attention in the scientific community: ParaView [72] and VisIt [73]. Both are powerful and mature products with similar functionality, distributed under an open-source license. ParaView relies on a library called `Catalyst` to provide in-situ visualization support, which is a reduced version of the ParaView's server component, allowing a program to share data with ParaView for visualization. Similar capabilities are available within VisIt with the `libsim` library. In addition to data sharing between the simulation and VisIt, `libsim` also enables the opposite flow of information, sending data from the client to the simulation, enabling simulation steering. We chose to use `libsim` and VisIt for our visualizations and steering since it offers a simple C API, which was easier to include in existing programs.

In the following sections, we describe how we used `libsim` component of VisIt to provide visualization of data from our programs and enable simulation steering.

7.1 Visualization of data

VisIt is a free, open-source, platform-independent, distributed, parallel, visualization tool for visualizing data defined on 2D and 3D, structured and unstructured meshes, as well as plots of 1D data. VisIt uses a client-server model (viewer-compute engine), where the server is parallelized and can be distributed on a cluster. This architecture allows for remote visualization, allowing us to perform large-scale visualization on a powerful computer cluster while running viewer component on an ordinary desktop computer connected to a cluster. VisIt's compute engine handles coordination between distributed processes and, given the information on how the data are distributed, can reassemble and process them for visualization (e.g., generate a plot and send it to the viewer component). Figure 7.1 depicts VisIt's compute engine accessing data in parallel.

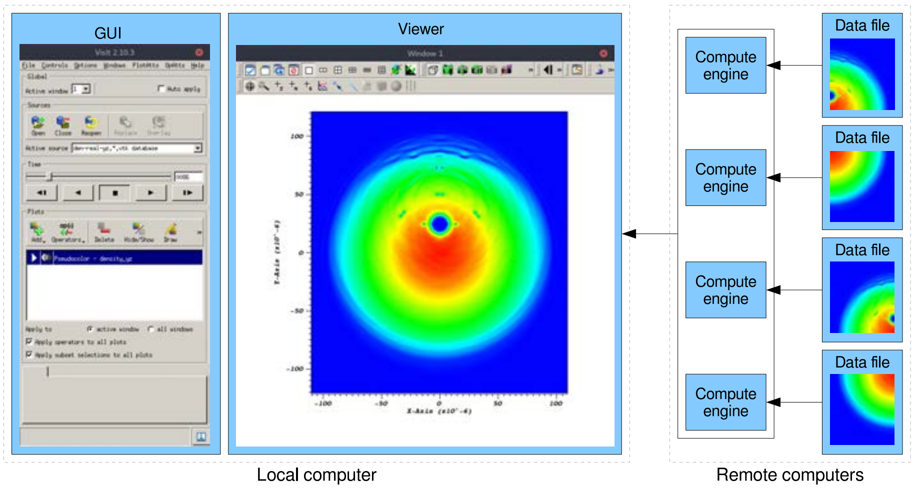


Figure 7.1: Illustration of VisIt's components. VisIt's compute engine reads data files in parallel and sends data to the viewer component, which user controls via GUI.

A scenario where processes of a program write separate data files is illustrated in Figure 7.1. Data created in such way are analyzed and visualized as a post-processing step. In order to make the simulation interactive, and to remove the need for writing data to disk, we have to allow VisIt's compute engine access to our data directly in memory. To do this, we rely on Visit's `libsim` library. This library can be inserted into a program to make the simulation behave like a VisIt compute engine. The `libsim` library and the accompanying data-access code gives VisIt's data-processing routines access to the simulation's calculated data without the need for the simulation to write files to disk (Figure 7.2). Simulation instrumented in this way can begin its processing while periodically listening for connections from an instance of VisIt using `libsim`. When an incoming connection is detected, `libsim` loads its dynamic runtime library that contains the VisIt compute engine's data-processing functions. Once the runtime is loaded, the simulation connects to VisIt's viewer, which can make requests for plots as if the simulation was an ordinary VisIt compute engine.

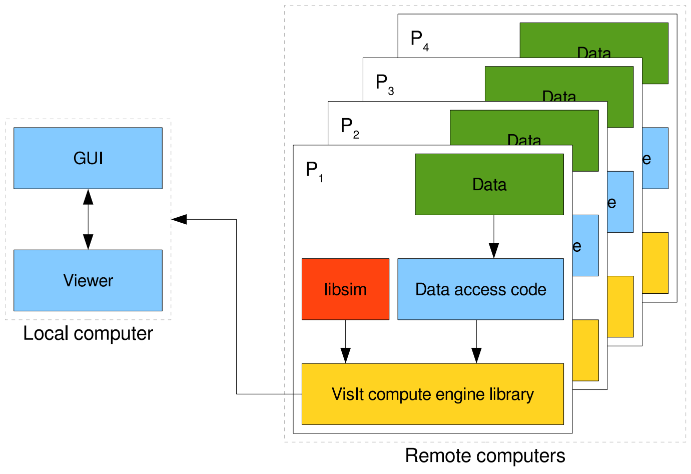


Figure 7.2: Schematic depiction of VisIt getting data from an instrumented parallel simulation.

User's requests for data from VisIt's viewer are passed to the simulation, which has to provide data through a set of callback functions. These callback functions are the main part of the integration with VisIt, as they provide information on what the simulation is doing, its current state, what data are available, how they can be represented, and how they are distributed among the processes. In addition to the callback functions, we also have to change the flow of our program to integrate `libsim`. General flow of a simulation with `libsim` integrated is illustrated in Figure 7.3.

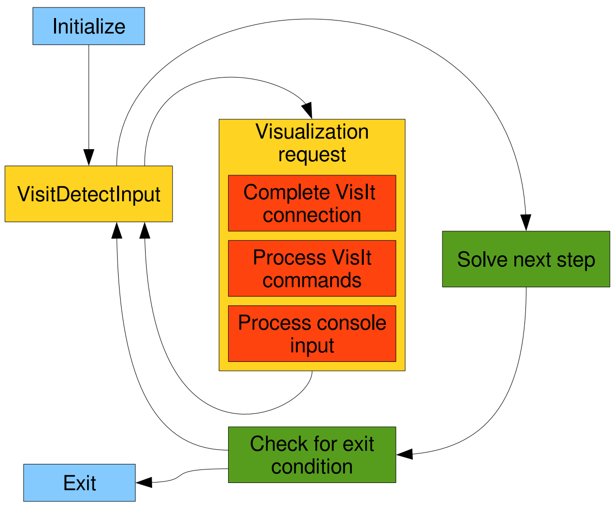


Figure 7.3: Main simulation loop with `libsim` integrated.

Restructuring programs to integrate `libsim` can be done in many ways, however we selected a pattern that VisIt uses in its examples and which is also recommended by VisIt's developers. Focus was placed on 3D variants of the programs, specifically, the MPI-parallelized versions described in Chapter 6, however other versions of programs would also be suitable candidates for integration with VisIt. We note that the necessary changes were not difficult to implement. While the changes are extensive, the only conceptually major change is the restructuring of the main loop to support connections from VisIt. Much of the restructured code can be shared between real- and imaginary-time programs, so we used a single implementation for most of the functions communicating with `libsim`. The differences were handled with conditional compilation, allowing us to enable the different functionality via a compiler switch. Listing 7.1 shows the flow of simulation after restructuring.

Listing 7.1: The flow of simulation in 3D programs after integration with `libsim`.

```
void main(...) {
   ... // Initialize variables and simulation data
   SimData *sim = ...; // Structure containing simulation state and variables
   SimMainLoop(sim);
   ... // Free resources and clean up
}

void SimMainLoop(SimData *sim) {
   int blocking, vstate, err = 0;

   // Register VisIt commands (start/stop, pause...), variables, on-demand output...
   SimRegisterCommands(sim);
   do {
      // Determines if the simulation should wait for user input or run anyway
      blocking = (sim->running == SIM_STOPPED) ? 1 : 0;

      if(sim->par_rank == 0) {
         vstate = VisItDetectInput(blocking, fileno(stdin)); // Detect user input, if any
      }
      MPI_Bcast(&vstate, 1, MPI_INT, 0, MPI_COMM_WORLD);

      switch(vstate) {
      case 0:
         SimNextStep(sim); // If there was no user input, proceed with the next iteration
         break;
      case 1:
         // Initial case, user attempts to connect to VisIt
         if(VisItAttemptToCompleteConnection() == VISIT_OKAY) {
            // Set command callbacks
            VisItSetCommandCallback(SimCommandCallback, (void*)sim);
            VisItSetSlaveProcessCallback2(SlaveProcessCallback, (void*)sim);
            // Set simulation metadata and variable callback
            VisItSetGetMetaData(SimGetMetaData, (void*)sim);
            VisItSetGetMesh(SimGetMesh, (void*)sim);
            VisItSetGetVariable(SimGetVariable, (void*)sim);
            VisItSetGetCurve(SimGetCurve, (void*) sim);
            VisItSetGetDomainList(SimGetDomainList, (void*)sim);
         } else {
            ... // Handle connection error
         }
         break;
      case 2:
         // User has sent an UI command, decode and process it
         if(!ProcessVisItCommand(sim)) {
            VisItDisconnect(); // Disconnect in case of failure
            sim->done = 1;
         }
         break;
      case 3:
         // User has sent an console command, decode and process it
         SimProcessConsoleCommand(sim);
         break;
      default:
         ... // Other states should not happen, handle errors if they do
         break;
```

```
    }
  } while (!sim->done && err == 0); // Loop until simulation is stopped or reaches the end
}
```

As part of the restructuring, we introduced a structure called `SimData` that represents the simulation's global state, has pointers to all important variables, and is used to pass information inside callback functions. We define several callback functions that return information about data in our programs and register them with VisIt by calling the appropriate function from `VisItSetGet*` set.

First callback function to be invoked is `VisItSetGetMetaData`, which provides metadata about the simulation. This is the function invoked when VisIt connects to the simulation, and the obtained metadata is used to determine which meshes and variables are available. The associated callback function must return a handle to a `SimulationMetaData` object that contains lists of metadata objects. Listing 7.2 demonstrates creation of some of the metadata used. For 3D programs, we define one 3D mesh, and three 2D meshes (every combination of two axes). These meshes represent the base on top of which the corresponding 2D or 3D variables will be visualized. The 3D mesh is used as a base for the variable containing the absolute square of the wave function ($|\psi|^2$), while the 2D meshes are used for the 2D integrated density profiles the programs generate. VisIt supports numerous types of meshes for many different use-cases. Due to the nature of the data in our programs, a *rectilinear* mesh is the best choice for our data, as it directly corresponds to the spatial grid we use. Rectilinear meshes are specified by lists of coordinate values for each axis, which map to spatial grid variables we already defined in our programs. For variables representing 1D arrays, there is no concept of 1D mesh, rather they are defined as *curves* in VisIt, and are visualized as ordinary plots. Using meshes and curves we have augmented the output system of our programs, which can now also send data to VisIt for visualization instead of just writing the computed output to a file.

Listing 7.2: The portion of the metadata exported from the 3D program to VisIt.

```
visit_handle SimGetMetaData(void *cbdata) {
  visit_handle md = VISIT_INVALID_HANDLE;
  SimData *sim = (SimData *)cbdata;
  // Allocate main SimulationMetaData object
  if(VisIt_SimulationMetaData_alloc(&md) == VISIT_OKAY) {
    // Define empty handles to meshes, variables and curves
    visit_handle xyzmd = VISIT_INVALID_HANDLE;
    visit_handle xymd = VISIT_INVALID_HANDLE;
    visit_handle xzmd = VISIT_INVALID_HANDLE;
#ifndef REAL
    visit_handle psimd = VISIT_INVALID_HANDLE;
#endif
    visit_handle psi2md = VISIT_INVALID_HANDLE;
    ...
    VisIt_SimulationMetaData_setMode(md, (sim->running == SIM_STOPPED) ?
        VISIT_SIMMODE_STOPPED : VISIT_SIMMODE_RUNNING);
    VisIt_SimulationMetaData_setCycleTime(md, sim->step, sim->time);
    // 3D mesh (XYZ) metadata
    if(VisIt_MeshMetaData_alloc(&xyzmd) == VISIT_OKAY) {
      VisIt_MeshMetaData_setName(xyzmd, "xyz_mesh");
      VisIt_MeshMetaData_setMeshType(xyzmd, VISIT_MESHTYPE_RECTILINEAR);
      VisIt_MeshMetaData_setTopologicalDimension(xyzmd, 3);
      VisIt_MeshMetaData_setSpatialDimension(xyzmd, 3);
```

```
        VisIt_MeshMetaData_setNumDomains(xyzmd, sim->par_size);

        VisIt_MeshMetaData_setXLabel(xyzmd, "z");
        VisIt_MeshMetaData_setYLabel(xyzmd, "y");
        VisIt_MeshMetaData_setZLabel(xyzmd, "x");

        VisIt_SimulationMetaData_addMesh(md, xyzmd);
     }
     // 2D mesh (XY) metadata
     if(VisIt_MeshMetaData_alloc(&xymd) == VISIT_OKAY) {
        VisIt_MeshMetaData_setName(xymd, "xy_mesh");
        VisIt_MeshMetaData_setMeshType(xymd, VISIT_MESHTYPE_RECTILINEAR);
        VisIt_MeshMetaData_setTopologicalDimension(xymd, 2);
        VisIt_MeshMetaData_setSpatialDimension(xymd, 2);
        VisIt_MeshMetaData_setNumDomains(xymd, sim->par_size);

        VisIt_MeshMetaData_setXLabel(xymd, "y");
        VisIt_MeshMetaData_setYLabel(xymd, "x");

        VisIt_SimulationMetaData_addMesh(md, xymd);
     }
     ... // Remaining 2D meshes
#ifndef REAL
     // psi variable
     if(VisIt_VariableMetaData_alloc(&psimd) == VISIT_OKAY) {
        VisIt_VariableMetaData_setName(psimd, "psi");
        VisIt_VariableMetaData_setMeshName(psimd, "xyz_mesh");
        VisIt_VariableMetaData_setType(psimd, VISIT_VARTYPE_SCALAR);
        VisIt_VariableMetaData_setCentering(psimd, VISIT_VARCENTERING_NODE);

        VisIt_SimulationMetaData_addVariable(md, psimd);
     }
#endif
     // psi2 variable
     if(VisIt_VariableMetaData_alloc(&psi2md) == VISIT_OKAY) {
        VisIt_VariableMetaData_setName(psi2md, "psi^2");
        VisIt_VariableMetaData_setMeshName(psi2md, "xyz_mesh");
        VisIt_VariableMetaData_setType(psi2md, VISIT_VARTYPE_SCALAR);
        VisIt_VariableMetaData_setCentering(psi2md, VISIT_VARCENTERING_NODE);

        VisIt_SimulationMetaData_addVariable(md, psi2md);
     }
     // Density (XY) variable
     if(VisIt_VariableMetaData_alloc(&denxymd) == VISIT_OKAY) {
        VisIt_VariableMetaData_setName(denxymd, "density_xy");
        VisIt_VariableMetaData_setMeshName(denxymd, "xy_mesh");
        VisIt_VariableMetaData_setType(denxymd, VISIT_VARTYPE_SCALAR);
        VisIt_VariableMetaData_setCentering(denxymd, VISIT_VARCENTERING_NODE);

        VisIt_SimulationMetaData_addVariable(md, denxymd);
     }
     ... // Other 2D variables
     /* Density (X) curve */
     if(VisIt_CurveMetaData_alloc(&denxmd) == VISIT_OKAY) {
        VisIt_CurveMetaData_setName(denxmd, "density_x");
        VisIt_SimulationMetaData_addCurve(md, denxmd);
     }
     ... // Remaining curves
     // Custom commands
     visit_handle cmd = VISIT_INVALID_HANDLE;
     if(VisIt_CommandMetaData_alloc(&cmd) == VISIT_OKAY) {
      VisIt_CommandMetaData_setName(cmd, "START/PAUSE");
      VisIt_SimulationMetaData_addGenericCommand(md, cmd);
    }
    ... // Remaining commands
  }

  return md;
}
```

Once VisIt's GUI is populated by the metadata obtained, the user can request the visualization of variables from the simulation, which will call the associated callback. For example, if one wants to visualize a variable labeled `psi^2`, one selects the appropriate command from the VisIt's GUI, and in the background, VisIt will use `libsim` to call the `SimGetMesh` (callback function registered in Listing 7.1) to get the info about the 3D mesh, then `SimGetVariable` to get the actual data, and other callback functions as needed. Listing 7.3 shows the portion of these callback functions. As part of the callback mechanism, functions receive the string with the name of the desired object (mesh, variable, curve, . . .), which can be used to provide different data. In this way we can define a single function to return all possible meshes or variables.

Listing 7.3: Portions of the mesh, variable and curve callbacks.

```
visit_handle SimGetMesh(int domain, const char *name, void *cbdata) {
  visit_handle h = VISIT_INVALID_HANDLE;
  visit_handle hxc, hyc, hzc;
  SimData *sim = (SimData *)cbdata;

  if(strcmp(name, "xyz_mesh") == 0) {
    if(VisIt_RectilinearMesh_alloc(&h) != VISIT_ERROR) {
      VisIt_VariableData_alloc(&hxc);
      VisIt_VariableData_alloc(&hyc);
      VisIt_VariableData_alloc(&hzc);

      VisIt_VariableData_setDataD(hxc, VISIT_OWNER_SIM, 1, sim->var.localNx /
          sim->out.outstpx + sim->ghost, sim->var.xstp + (sim->var.offsetNx /
          sim->out.outstpx));
      VisIt_VariableData_setDataD(hyc, VISIT_OWNER_SIM, 1, sim->var.Ny /
          sim->out.outstpy, sim->var.ystp);
      VisIt_VariableData_setDataD(hzc, VISIT_OWNER_SIM, 1, sim->var.Nz /
          sim->out.outstpz, sim->var.zstp);
      VisIt_RectilinearMesh_setCoordsXYZ(h, hzc, hyc, hxc); // We use inverted order

      return h;
    }
  }
  ... // Remaining meshes
  return h;
}

visit_handle SimGetVariable(int domain, const char *name, void *cbdata) {
  visit_handle h = VISIT_INVALID_HANDLE;
  long ntuples;
  SimData *sim = (SimData *)cbdata;

  if(strcmp(name, "psi^2") == 0) {
    VisIt_VariableData_alloc(&h);
    ntuples = (sim->var.localNx / sim->out.outstpx + sim->ghost) * (sim->var.Ny /
        sim->out.outstpy) * (sim->var.Nz / sim->out.outstpz);
    VisIt_VariableData_setDataD(h, VISIT_OWNER_SIM, 1, ntuples, **(sim->var.psi2));
```

```
      return h;
   }
   ... // Remining variables
   return h;
}

visit_handle SimGetCurve(const char *name, void *cbdata) {
   visit_handle h = VISIT_INVALID_HANDLE;
   SimData *sim = (SimData *)cbdata;

   // Note that only rank 0 calls this function
   if(strcmp(name, "density_x") == 0) {
      if(VisIt_CurveData_alloc(&h) != VISIT_ERROR) {
         visit_handle hxc, hyc;
         VisIt_VariableData_alloc(&hxc);
         VisIt_VariableData_alloc(&hyc);

         VisIt_VariableData_setDataD(hxc, VISIT_OWNER_SIM, 1, sim->var.Nx /
              sim->out.outstpx, sim->var.xstp);
         VisIt_VariableData_setDataD(hyc, VISIT_OWNER_SIM, 1, sim->var.Nx /
              sim->out.outstpx, sim->var.denx);
         VisIt_CurveData_setCoordsXY(h, hxc, hyc);

         return h;
      }
   }
   ... // Remaining curves
   return h;
}
```

In our simulations, we distribute data among multiple processes, and this information must also be passed via `libsim`. VisIt supports working with distributed data through the concept of *domains*. A domain is a unit of work assigned to a given process that corresponds to a portion of the mesh. By decomposing the meshes into multiple domains, VisIt is able to handle large-scale simulations. Therefore, in a distributed simulation, we have to provide VisIt with information on how domains abut, via another callback function. Additionally, each domain needs information on its adjacent domains, to ensure continuity between domains. This information is provided via *ghost nodes*. There are a few ways to provide information on ghost nodes, in our case the simplest way was to extend the data on each process to include the first spatial points of adjacent processes. The overlap of data obtained in this way is designated as a ghost node between adjacent domains. This has proved to be simple to implement, as it does not require changes to any computation. Instead, the processes exchange data just prior to sending them to VisIt, with a call to `MPI_Sendrecv`. Since the amount of data is small, the impact on memory usage and simulation performance is negligible. Figure 7.4 illustrates this concept.

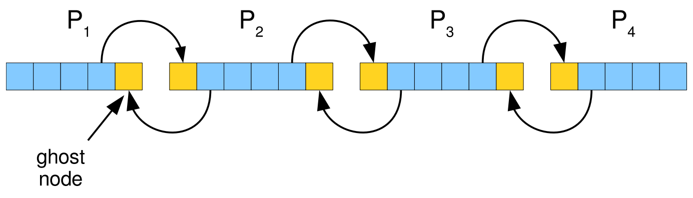


Figure 7.4: Exchange of data between neighboring processes to construct ghost nodes.

This and other callback functions are implemented in a similar manner to the ones presented above, i.e., they all return a handle to an object that contains the necessary information. By providing the aforementioned metadata, VisIt can now connect to our programs, and visualize variables as they are updated. Figure 7.5 illustrates some of the visualizations made possible with this integration.

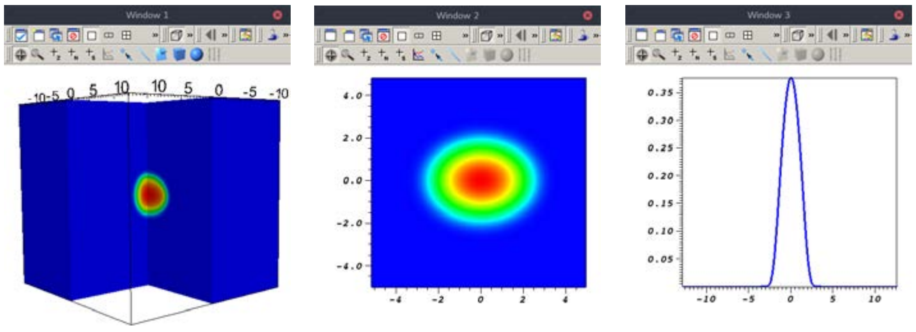


Figure 7.5: Examples of visualizations in VisIt: 3D pseudocolor plot (left), 2D pseudocolor plot (center) and curve plot (right).

We note that, while interacting with the simulation is very useful, sometimes the classic, approach where analysis of data happens after the simulation, is preferred. To enable this usage scenario, we added support of exporting data to VisIt as files independent of the running simulation. For visualization purposes, VisIt supports numerous formats via plugins, however two formats stand out as being the most commonly used: Silo and VTK. Silo [74] is data format and library for reading and writing a wide variety of scientific data, developed together with VisIt. It is mostly used for data intended to be visualized with VisIt, however it can be used with other tools as well. VTK [75] stands for Visualization Toolkit and is a software system for 3D computer graphics, image processing and visualization. It is developed by the same company that is also the primary developer of ParaView. VTK supports two different file formats, the legacy textual format that is easy to write either by hand or a library, and the newer XML-based format which has more advanced features and is often written with its own library. As all three file formats (Silo, textual VTK and XML-based VTK) meet our needs, the choice between them was based on personal preference and ease of use. We decided to use textual VTK, which we found to be more widespread in scientific communities, has good compatibility with the major visualization tools, and is simple to use. VisIt provides sample programs for writing textual VTK files, which we used as the base for our implementation. In this way the same set of meshes and variables can be exported as VTK files. Since the binary output produced by our programs can still be useful for data analysis without visualization, we decided to keep it together with VTK output.

7.2 Simulation steering

With changes shown in the previous section VisIt viewer component can connect to our programs and visualize the exported variables. However, the simulation is still independent of VisIt, and runs even if there is no connection from VisIt. In this section we describe how our simulation is extended to accept commands from VisIt.

VisIt supports issuing custom commands that can be used to send data to the simulation. The data sent can be interpreted as instructions to alter the simulation state. There are three ways to send data to the simulation: via simulation window in viewer GUI, via a custom user interface, or via console interface. In order to cover more use-cases, we implemented support for all three ways of commanding.

Simulation window, available after VisIt connects to a simulation, allows custom commands to be specified, which are accessible as push buttons the user can interact with. Pushing a button invokes a callback function in the simulation, which can perform some action based on the button selected. While useful for some basic commanding, this approach is very limited because there is no way to send any parameters to the simulation. That means that the callback function can only perform work based on the name of the command. For this reason, we use this functionality only for the most basic commands, i.e., start/pause the simulation, reset the simulation to the initial state, perform one step of time propagation, force the update of VisIt's plots, and stop the simulation and free the resources. To create these commands, they first must be registered with VisIt as part of the metadata callback (Listing 7.1). This essentially provides labels that will be written on the buttons. After that, along with other callbacks, we provide a callback to a function that will be invoked when a user clicks on a button in the simulation window. Listing 7.4 demonstrates this.

Listing 7.4: Registering basic VisIt commands.

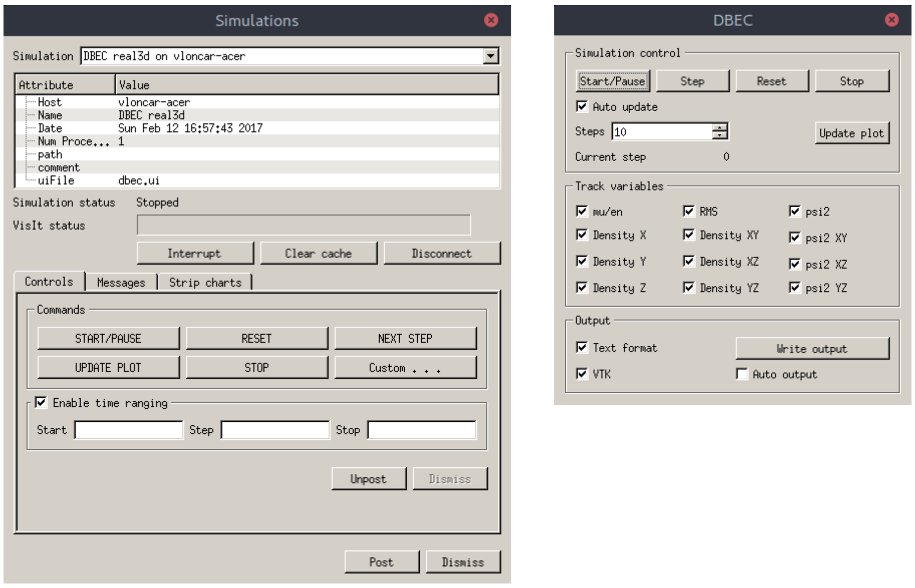
```
// Callback for VisIt’s Simulation window
void SimCommandCallback(const char* cmd, const char* args, void* cbdata) {
   if (strcmp(cmd, "START/PAUSE") == 0) {
      SimUIStartPauseClick(cbdata);
   } else if (strcmp(cmd, "RESET") == 0) {
      SimUIResetClick(cbdata);
   } else if (strcmp(cmd, "NEXT STEP") == 0) {
      SimUIStepClick(cbdata);
   } else if (strcmp(cmd, "UPDATE PLOTS") == 0) {
      SimUIUpdatePlotClick(cbdata);
   } else if (strcmp(cmd, "STOP") == 0) {
      SimUIStopClick(cbdata);
   }
}
void SimUIStartPauseClick(void* cbdata) {
   SimData* sim = (SimData*) cbdata;
   sim->running = !sim->running;
   // Temporary disable the Simulation window so that the user can not issue multiple
       commands and cause concurrency problems
   SimSetupUI(sim, !sim->running);
   VisItTimeStepChanged(); // Notify VisIt so that the GUI is refreshed with the new data
}
void SimUIStepClick(void* cbdata) {
   SimData* sim = (SimData*) cbdata;
   SimSetupUI(sim, 0);
   SimNextStep(sim); // Perform next step in the time propagation and update GUI
   SimSetupUI(sim, 1);
}
... // Remaining three functions omitted for brevity
```

Greater flexibility can be obtained by creating a custom GUI. They are made using Qt toolkit [76], in which VisIt's GUI itself is made. Custom GUIs are designed using Qt Designer tool, which is a part of Qt software package. While simulation window only allows push buttons to be programmed, custom UIs support three different widgets (at the time of writing): push button, check box, and spin box widget. Push button widgets work the same way as in simulation window, allowing some predefined action to be taken. On the other hand, check box widgets can be used to change the value of some boolean parameter in the simulation, while spin box widgets can be used to change a numeric value of some variable. Noticeably missing is the support for text input widget, which would be useful to send some textual (or decimal) value to the simulation. Support for additional widgets may come in a future version of VisIt, however we find the current widgets to be adequate for most cases. By using these three widget types we implemented support for: toggling computation of output variables (which may significantly speed up each step of the simulation), toggling automatic updates of VisIt's plots, writing current state of variables on-demand in multiple formats, and performing the selected number of steps in the simulation (Listing 7.5). The basic commands of the simulation window were also implemented in the custom GUI. In addition to sending data to the simulation, custom GUIs can also receive data from it. In this way, the custom GUI may present some information to the user. For example, we use this functionality to display the current iteration number on the custom UI.

Listing 7.5: Registering VisIt commands for the custom GUI. The callbacks for individual actions are similar to the functions in Listing 7.4

```
void SimRegisterCommands(SimData* sim) {
   // Simulation control
   VisItUI_clicked("btnStartPause", SimUIStartPauseClick, sim); // Button
   VisItUI_clicked("btnStep", SimUIStepClick, sim);
   VisItUI_clicked("btnReset", SimUIResetClick, sim);
   VisItUI_clicked("btnStop", SimUIStopClick, sim);
   VisItUI_clicked("btnUpdatePlot", SimUIUpdatePlotClick, sim);
   VisItUI_stateChanged("boxAutoUpdate", SimUIAutoUpdateChange, sim); // Check box
   VisItUI_valueChanged("spnStep", SimUIStepValue, sim); // Spinner
   // Variables to track
   VisItUI_stateChanged("boxMuen", SimUIMuenChange, sim);
   VisItUI_stateChanged("boxRms", SimUIRmsChange, sim);
   VisItUI_stateChanged("boxPsi2", SimUIPsi2Change, sim);
   VisItUI_stateChanged("boxDenX", SimUIDenXChange, sim);
   VisItUI_stateChanged("boxDenY", SimUIDenYChange, sim);
   VisItUI_stateChanged("boxDenZ", SimUIDenZChange, sim);
   ...
   // Output
   VisItUI_stateChanged("boxTextOut", SimUITextOutChange, sim);
   VisItUI_stateChanged("boxVtkOut", SimUIVtkOutChange, sim);
   VisItUI_stateChanged("boxAutoOut", SimUIAutoOutChange, sim);
   VisItUI_clicked("btnOutput", SimUIOutputClick, sim);
}
```

The custom GUI can be invoked from the simulation window, and interaction with it causes the registered functions to be called. Again, a simple mechanism based on a name of the widget user is interacting with is used to differentiate between the widgets in the GUI. The simulation window and the custom GUI are shown in Figure 7.6.



Simulation: DBEC real3d on vloncar-acer

Attribute	Value
Host	vloncar-acer
Name	DBEC real3d
Date	Sun Feb 12 16:57:43 2017
Num Proce...	1
path	
comment	
uiFile	dbec.ui

Simulation status: Stopped
VisIt status:

☑ Enable time ranging
Start: Step: Stop:

☑ Auto update
Steps: 10
Current step: 0

☑ mu/en ☑ RMS ☑ psi2
☑ Density X ☑ Density XY ☑ psi2 XY
☑ Density Y ☑ Density XZ ☑ psi2 XZ
☑ Density Z ☑ Density YZ ☑ psi2 YZ

☑ Text format
☑ VTK ☐ Auto output

Figure 7.6: Simulation window with five programmable buttons (left) and more advanced custom GUI (right).

GUIs created in Qt Designer are essentially XML files, and the user does not have to compile them or link them in any special way with VisIt. The user only has to place the XML file of the GUI in a directory where VisIt can find it. They are completely independent of the simulation itself, and their (un)availability does not interfere with the simulation.

The third option for issuing commands to the simulation is via the console interface. This interface allows the user to type commands in the console window of the running simulation. This way of commanding predates other two approaches and is perhaps the most flexible. This flexibility comes from the fact that anything the user types in the console gets sent to the simulation as a string that can be further processed in any way. Therefore, with sufficiently capable parser, the console interface can alleviate the current limitations of GUI commanding. Additionally, this is the only available interface when VisIt is started without its GUI component. VisIt can be started without its GUI and scripted to perform many advanced actions, however this scenario is out of scope for this thesis. Since we did not focus on advanced scripting with VisIt, we implemented the console interface to have the same functionality as the graphical one.

We added support for the console interface as a special case in the main VisIt loop inside the simulation, as illustrated in Listing 7.1. `VisItDetectInput` returns a value of 3 to denote that the command issued was a console command, and we process it with the `SimProcessConsoleCommand` function. This function may then perform any work based on user's input. We do not present its implementation because it covers the same functionality already presented.

Chapter 8

Performance evaluation and modeling

In previous chapters we presented various parallel implementations of programs solving the dipolar GP equation without providing details of their measured performance. This chapter is dedicated to filling that gap. Before performance evaluation, all programs were tested for correctness to make sure they produce the same results as original serial programs, and extended to record execution time of the main loop.

We tested various scaling scenarios, strong and weak, from a single computing node to the cluster. All testing was done at the PARADOX supercomputing facility located at the Scientific Computing Laboratory, Center for the Study of Complex Systems of the Institute of Physics Belgrade.

Performance of hybrid implementations is highly dependent on the amount of work offloaded to GPU(s), so the key to maximizing performance of these versions is proper selection of parameters which control this feature. We used evolutionary computation techniques to find the optimal solutions, as described in Section 8.1. In the remaining two sections we give an overview of the tests and the methodology they relied on (Section 8.2), and present the performance results and their modeling in Sections 8.3 and 8.4. Section 8.5 discusses how to select the optimal algorithm for a given hardware platform.

8.1 Optimization of input parameters for hybrid implementations

Hybrid implementations, running on a single computer or a cluster with multiple computing nodes, can potentially offer the best performance of all algorithms presented in this thesis, by utilizing all allocated computing resources. For this to happen, work must be divided between CPU and GPU in such a way as to maximize their throughput and minimize their idle time. A question that naturally arises is how to achieve this for any possible mesh size for a given CPU/GPU combination. Unfortunately, there is no single best way to divide the work, since processing powers of CPUs and GPUs vary significantly. For example, in a computer where a powerful new GPU is paired with an older type of CPU, more work should be offloaded to the GPU, and vice versa for computers with a powerful CPU and low-performing GPU. Even if the installed CPU and GPU offer similar performance in terms of floating point operations per second (as is the case for the PARADOX cluster), the differing architectures mean that certain portions of the algorithm are better suited to CPU or GPU. If we were to divide the work naively, e.g., equally, this would lead to unbalanced computation, as illustrated in Figure 8.1(a), which in turn increases the computation time.

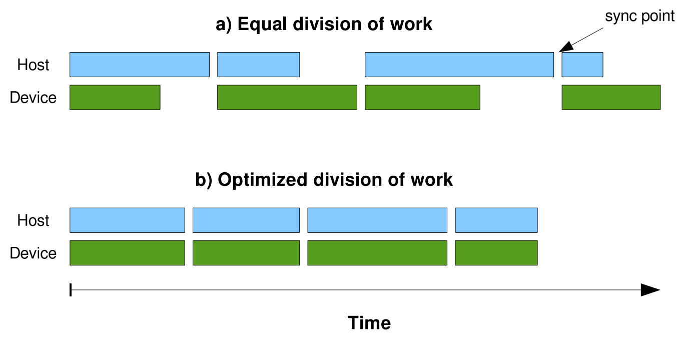


Figure 8.1: Execution timeline in a hybrid system: (a) with equal data distribution, leading to unbalanced computation time; (b) with optimized data distribution and ideal computation load.

To get over this problem, our hybrid implementations have an option to manually specify each parameter that controls the work being done on a GPU. Ideally, this flexibility would allow us to overlap computation on CPU and GPU as much as possible, thus minimizing the execution and idle time of each resource. The execution timeline for the ideal case looks like the illustration in Figure 8.1(b).

The aforementioned parameters include the total amount of data transferred to GPU, the number of chunks, kernel grid size parameters and a special parameter controlling whether some functions will even be offloaded to GPU or not, which is useful in situations where the CPU is much more powerful than the GPU. In 3D programs, this amounts to 33 integer-valued parameters for which we would like to find the optimal values for the desired mesh size on a given computing system (with given hardware characteristics). Furthermore, there are constraints that the parameters must satisfy. For example, the total amount of data transferred to GPU must be divisible by the number of chunks, and the size of a block that kernel launches must fit within the limits imposed by the underlying GPU.

To evaluate any combination of parameters, the programs have to be executed, and their execution time measured. The search space of 33 parameters is clearly too large to be exhaustively traversed in a reasonable amount of time. Fortunately, we can reduce the number of parameters we have to search through at any single time by grouping them based on the synchronization point. After the CPU and GPU synchronize, data have to be divided again, so the only relevant parameters in the region between two synchronization points are the ones controlling the division of work and the ones controlling the kernels which are executed in that region. During a single iteration, CPU and GPU have to synchronize four times:

1. after the FFT on local dimensions,
2. after the inverse FFT on the non-local dimension,
3. before `calclux` function, and

4. after `calclux` (which represents the end of a single iteration).

This gives us four distinct parameter sets to optimize: PS_1 with 6 parameters, PS_2 also with 6 parameters, PS_3 with 15 parameters and PS_4 with 5 parameters. We can consider each parameter set independently.

We investigated several approaches to optimization of parameter sets: naive brute-force search, iterative optimization via gradient descent, and metaheuristic via genetic algorithm. Our goal was not to find the best possible optimization algorithm for our problem, but rather than to implement a reasonably good one. By this we mean the algorithm which is not difficult to implement, which will give us a set of parameters that are close to the optimal ones, and will not take a long time to finish. Our focus was on 3D variants of both single node and MPI versions of hybrid programs. The concepts presented here apply equally to 2D variants and there is no important difference between the single node and the MPI version, so we will not make this distinction in the remainder of this section. We will now describe how the three approaches were implemented.

A brute-force search (BFS), or exhaustive search, considers every possible combination of parameter values in order to find the optimal one. Even with parameters divided into smaller sets, the exhaustive search takes too long, and is only feasible if we narrow down the ranges of all parameters involved. Unfortunately, this is only possible when we have a deep understanding of the performance of the CPU and GPU models used, when we can provide a reasonable guess for the optimal solution. An alternative is to consider only certain values along the range of a single parameter, rather than every value in the range. This approach is feasible and we were able to find reasonably good parameter values using it. The downside is that this way we may miss the optimal values of parameters, as they often lie between the selected test points. For instance, on one PARADOX computing node and a mesh size $256 \times 256 \times 256$, with the work divided between CPU and GPU along the outermost, x dimension, if we test only for the values of offloaded data to GPU that are multiples of 32 (i.e., $32 \times 256 \times 256$, $64 \times 256 \times 256$, etc.), we will miss the optimal value which lies around $140 \times 256 \times 256$ (assuming all other parameters are fixed). Even though BFS is often not feasible, it is useful as the baseline for the evaluation of other methods we implemented.

With the BFS implemented as a baseline, we switched our focus to the implementation of an iterative optimization algorithm. A well-known and widely used algorithm is *gradient descent* (GD) [77]. It is an iterative procedure in which every iteration aims to get closer to the minimum of the given function $F(\mathbf{x})$, where $\mathbf{x}$ represents a vector of function parameters. If the function $F(\mathbf{x})$ is defined and differentiable around some point $\mathbf{a}$, then $F(\mathbf{x})$ decreases fastest in the direction of the negative gradient of F at $\mathbf{a}$. GD exploits this fact and defines $\mathbf{b}$ as

$$\mathbf{b} = \mathbf{a} - \gamma \nabla F(\mathbf{a})\,, \tag{8.1}$$

so that $F(\mathbf{b}) \leq F(\mathbf{a})$, for γ small enough. This observation is used as a building block for the algorithm. We start from an initial guess $\mathbf{x_0}$ for a location of the minimum of F and construct a sequence $\mathbf{x}_n (n \geq 1)$ such that:

$$\mathbf{x}_n = \mathbf{x}_{n-1} - \gamma \nabla F(\mathbf{x}_{n-1})\,. \tag{8.2}$$

This sequence will eventually converge to the local minimum, if one exists. GD is most applicable to functions for which the gradient can be computed analytically, however it can also be used when the derivative can only be obtained numerically. To apply it here, the execution time of our programs will be the function F that we minimize, and its arguments are the $n = 33$ parameters that control the GPU. The execution time is measured as the average time of a single iteration of the main time-propagation loop, sampled over a given number of iterations. As explained earlier, we can divide the problem into four separate minimization problems, with four sets of parameters of dimensionality m_i $(i = 1, 2, 3, 4)$, in the same way as in BFS. To numerically compute the k-th partial derivative of one of the minimization functions at a set of candidate parameters $a_1, a_2, \dots, a_m$, we use first order approximation:

$$\frac{\partial F}{\partial x_k}(a_1, \dots, a_m) \approx \frac{F(a_1, \dots, a_k + h_k, \dots, a_m) - F(a_1, \dots, a_k, \dots, a_m)}{h_k}, \qquad (8.3)$$

where h_k is the increment of a given parameter. Since the parameters are integer-valued, we have to carefully choose the value of h_k, to be as small as possible, while still satisfying all the constraints that the corresponding parameter may have w.r.t. other parameters. We note that in order to evaluate the full gradient of F at $(a_1, \dots, a_m)$ we need to execute our programs $m + 1$ times.

We stress that execution times of our programs are not always the same due to the hardware, software and OS scheduling issues, making the minimization functions noisy. This noise affects the calculation of the gradient and could point the algorithm in the wrong direction. We can detect this by checking if the value of minimization function has increased in subsequent GD iteration, and discard this move if necessary. While useful, resorting to this tactic means that in case the GD gets stuck in a local minimum, it will not be able to get out and thus will never reach the global minimum. A naive way to check if this is the case would be to start over from a different initial point, and see if the GD algorithm converges to the same minimum. There are more sophisticated approaches to addressing this issue, which we discuss later.

In general, it is not possible to completely eliminate noise, due to inherent problems of accurately measuring time on a computer. However, we can try to minimize it by increasing the precision of execution time measurement of our programs. This can be achieved by averaging execution times over larger number of iterations of the main time-propagation loop. Unfortunately, this also means that the total execution time of the GD algorithm will significantly increase.

Another issue in our implementation of GD arises when we have to select the next values of our parameters based on the output values of previous GD iteration. These output values are real-valued, meaning that we need to convert them to integers somehow. Simply rounding them up or down to the nearest integer will not be enough, especially for small value of γ. For example, if we set the initial value of some parameter to 10, and after one GD iteration the proposed value of that parameter is 10.2, rounding it down will reset the value back to 10, effectively discarding the whole GD iteration. To avoid getting stuck in this way, our implementation selects the next possible value in the direction of change. In the example above it would mean selecting 11 as the new value. This change makes it impossible for the algorithm to converge, but it will usually stay close the minimum where all proposed solutions are of similar quality.

With the GD algorithm implemented as described above, we are able to get a set of optimized parameters from a random set of initial values. However, such optimized parameters are often suboptimal, and do not have the best performance. This is due to the fact that in most cases the GD will converge to the nearest local minimum, and there is no guarantee that that local minimum is also the global one. Using our previous example from BFS, a mesh of $256 \times 256 \times 256$, with GD we obtain that local minima exist for all power of two values, i.e., parameters suggest offloading $2^n \times 256 \times 256$ to the GPU. If we randomly select a small initial value, GD will get stuck in the nearest local minimum, which in this example will be far from the global one, thus producing a very bad solution. We can attempt to avoid getting stuck in a local minimum by adapting the parameter γ, also known as *learning rate* in machine learning literature. Keeping the learning rate high for a first few iterations would allow the algorithm to find the general location of the minimum, after which we could gradually lower the learning rate until convergence is achieved, in a process called *annealing* [78]. In practice this did not completely solve the problem, as we found that the optimal learning rate to start with varies with the initial set of parameters and the mesh size, and thus has to be manually selected. This complicates attempts to automate the process of finding the optimal parameters for a range of mesh sizes, which we needed as part of the tests of both hybrid implementations. A method to remove the manual tuning of the learning rate exists [79], but we find it to be too complex to implement for our programs, because it would require significant changes to the way the parameters are selected.

From the discussion above, we conclude that the GD is not the best-suited method for the optimization in our case, mostly because our minimization function is noisy and the numerical computation of the gradient is costly. Derivative-free optimization methods would be better suited to the problem, e.g., stochastic approximation algorithms like simultaneous perturbation (SPSA) [80, 81], or metaheuristics like genetic algorithms. We decided to use a genetic algorithm approach, as it is simple to understand and implement, as well as easy to adjust to get the desired behavior.

Genetic algorithm (GA) is an optimization method based on natural selection that mimics the process found in biological evolution [82]. GA works by creating a population of individual solutions, which it then evaluates and modifies, creating a new population, and iterating this process. Unlike the classical algorithms like GD, which iterate a single candidate solution towards the optimal one, the GA iterates a population of solutions in which the best individuals approach the optimal solution. The initial population is usually created at random, giving the GA different points in the search space to start from. The individual solutions are evaluated using a user-supplied fitness function, giving each individual a score based on how well they perform the given task. Individuals with the highest score are then selected to "reproduce" and create new offspring, after which they may be mutated randomly. The offspring form a new population, and the process can be repeated again. The GA continues until a suitable solution is found, or after a certain number of generations has passed.

To implement a GA, we first have to decide how to represent an individual. Individuals are created based on their blueprint, called *chromosome* in GA terminology. The most often used representation is bit-string [83], where all properties of an individual (its *genes*) are serialized to an array of bits, and then concatenated. More advanced representations exist, e.g., for encoding real values, permutations and general data structures [84, 85], however since our individuals are sets of integer parameters, we did not develop any special representation and instead we used ordinary arrays of integers as a chromosome. Therefore, each gene in a chromosome is a single GPU parameter of our programs. Individuals need to be evaluated using a fitness function. In our case, this means executing the programs with the parameters extracted from the individual's chromosome, and reporting the execution time as fitness score, with lower execution time being better.

Next step is the implementation of the three GA operators: selection, reproduction (*crossover* in GA terminology) and mutation. Each operator can be implemented in different ways. Most common type of the selection operator is *roulette wheel* selection. This is a type of fitness-proportionate selection, with the idea to give each individual a slice of the circular roulette wheel based on their fitness. The wheel is then spun, and when the roulette ball stops, the individual in whose region the ball stopped is selected. In this way, the fittest individuals have the greatest chance of being selected for reproduction, while the ones with very low score quickly die out. We also tested an alternative type of selection, the *tournament* selection. In this type of selection, we randomly select several individuals, and host a tournament for them. The individual with the best fitness score wins, and is selected for reproduction. By changing the size of the tournament we can control the selection pressure, e.g., by using small tournament size in the first few iterations we can prevent premature convergence and increase it in later generations when we have explored the search space enough. In our tests, the tournament selection gave slightly better results for smaller populations, by keeping the population diverse in early generations. For larger population size, both selection rules performed equally. Alongside the main selection algorithm, we also used *elitist* selection, where first few individuals are copied to the next generation without changing their chromosome. This prevented the loss of the best individuals in the next generation, but has to be used carefully as it may lead to premature convergence to a suboptimal solution.

To produce the next generation, selected individuals should combine their chromosomes and produce offspring. This is the task of the crossover operator. Crossover exchanges parts of the chromosomes, mimicking the biological recombination from nature. In its most basic form, crossover works by randomly selecting a point and exchanging segments before and after the point to create two new offspring from two parents. This type of crossover is called *single-point crossover*, and is illustrated in Figure 8.2(a). Other popular crossover techniques are *two-point crossover* and *uniform crossover*, illustrated in Figures 8.2(b) and 8.2(c). Two-point crossover is similar to the single-point crossover, just with two points instead of one. On the other hand, in the uniform crossover each gene of the offspring is selected randomly, either from the first parent or from the second one, with some fixed probability, typically 0.5. Using the uniform crossover leads to a wider exploration of the search space [86], but this may not always result in better performance of the operator [87]. We tested single- and two-point crossover and found that there is little difference in terms of performance between them. Since some of our parameters have constraints, it is important to select only the crossover points which lead to an allowed recombination of genes. Unfortunately, this made the implementation of uniform crossover impossible, and we did not pursue it further.

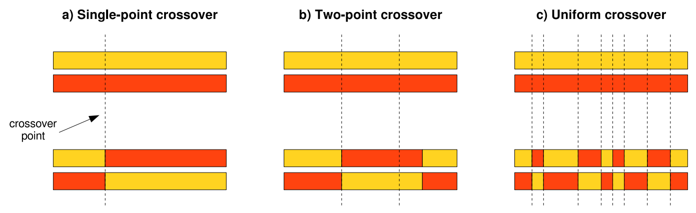


Figure 8.2: Three different crossover techniques in GA: a) single-point crossover, b) two-point crossover and c) uniform crossover.

The final step in producing the next generation is to apply the mutation operator on the new population. The mutation operator randomly changes genes to new values, which is equivalent of performing a random walk through the search space. Changing every gene would not be desired, so mutation operates with a very small probability, meaning most of the genes will be left unchanged.

When chromosomes are implemented as bits, the mutation would be equivalent to flipping a random bit. However, we could not implement mutation this way, as random changes to the chromosome would often result in non-functioning individuals, e.g., parameter constraints would not be satisfied. Special care must be taken to ensure that the mutation produces a healthy individual, similarly to the crossover operator. The importance of mutation operator and its relation to the crossover is often debated [88, 89], with its role being defined as "to maintain diversity within the population and inhibit premature convergence" [90], as the crossover operator does not introduce new information to the population. In our tests, the mutation plays a crucial role if the population size is small, e.g., less than 20 individuals, in accordance with Refs. [91, 92]. If the population size is large, e.g., over 200 individuals, the positive effects of mutation on the population fitness are not that evident.

GA is typically iterated for a fixed number of generations, as is the case in our implementation. Alternatively, we could have implemented some exit clause in the main loop of the GA which would stop the evolution after the best individuals have not been improved for some number of generations. After each run of the GA, there is usually several highly fit individuals in the population. Since randomness is an integral part of any GA, different runs of the algorithm produce slightly different results. Therefore, our GA does not converge to a single solution, but produces candidate solutions that have very similar parameter values and overall fitness. Among the candidate solutions, there may be small differences in the amount of data offloaded to GPU or the kernel parameters, with the execution times negligibly different. The number of generations required to produce a good solution also varies due to the randomness in the initial population and the population size. In our tests, it has a value between 10 and 50.

The implementation described above provides only the basis for a successful application of GA to the problem at hand. As can be seen, there are several important parameters of GA to tune in order to get optimal results. These include the population size, the number of generations, mutation rate, and the selection parameters (e.g., elite selection rate, the tournament size). We did not perform thorough testing of the performance of the GA that would allow us to obtain the best values for these parameters, as we were more focused on the quality of the candidate solutions GA creates. However, we observed that our GA finds good solutions faster if the initial population size is between 100 and 200, with the number of generations between 10 and 20, depending on the set of parameters we wish to optimize.

To get a perspective of how the three optimization methods perform, we tested them by comparing their final solutions, as well as by recording the number of program executions needed to get to the optimal set of parameters. For this test, we used 3D real-time propagation hybrid program on a single computing node. All three methods were allowed to execute the program up to 1000 times. The mesh sizes used range from $80 \times 80 \times 80$ to $600 \times 600 \times 600$, all of which could potentially be offloaded to GPU. We tested parameter set PS_3 (with 15 parameters), which control the execution of FFT and the kernels in the subsequent `calcnu`, `calcluy` and `calcluz` functions. The BFS algorithm was used as the baseline. Note that the range of parameters and a small number of allowed executions implies that the BFS algorithm takes the values of parameters with large stride, potentially missing the optimal solution. GD method is used as described, with the learning rate γ initially set to a higher value, which was gradually decreased. We performed the GD for 60 iterations, which amounts to 960 program executions. The GA was run on a population of 100 individuals, for 10 generations, amounting to the same the number of program executions as the

BFS. We have used mutation rate of 5%, and the tournament selection, with the tournament size equal to 10% of the population size. Elitism was also included, with the top 2% of the population copied over. All algorithms were tested five times, to minimize the effects of random initialization of both GD and GA methods. The results are shown in Table 8.1.

Table 8.1: A comparison of three optimization methods. Reported times are given in milliseconds, for a single iteration of the main time-propagation loop, averaged over 50 iterations. The last column contains the minimal execution times obtained by manual tuning. The reported time for each algorithm represents the minimal achieved value in five test runs.

Mesh size	BFS	GD	GA	Best
$80 \times 80 \times 80$	7	6	6	6
$128 \times 128 \times 128$	18	27	16	16
$240 \times 240 \times 240$	154	272	135	126
$256 \times 256 \times 256$	181	199	169	156
$360 \times 360 \times 360$	343	387	312	298
$480 \times 480 \times 480$	981	1049	868	829
$512 \times 512 \times 512$	1452	1628	1312	1242
$600 \times 600 \times 600$	2591	2984	2227	2159

From the results we conclude that the GA was the most effective optimization method of the three approaches. As expected, GA found better solutions than BFS due to large strides BFS had to use. The randomness in the initial population has a big effect on the convergence of the GA method, sometimes enabling it to find the optimal solution after just three generations. Even if the fitness of the initial population is very bad, the GA still converges to very good solutions after 10 generations. On the other hand, GD performed very poorly, mainly because the noise in program execution times has often thrown it in the wrong direction. Also, GD would get stuck in the nearest local minimum, which often was not the global one. When the initial position of GD is near the global minimum it converged to toward the optimal solution, which was rarely the case. For mesh sizes larger than the tested ones (relevant for MPI-based implementations), corresponding to a larger range of parameters, GD would be even less effective. Both BFS and GD can be made more usable if the range of parameters can be narrowed, i.e., if we know the relative performance of the GPU in comparison to the CPU. However, the GD would still be somewhat inefficient, due to its higher susceptibility to noise.

Since the GA method is shown to be superior to the other two approaches, we use GA as our optimization method of choice in the next two sections.

8.2 Testing methodology

All programs were tested on the PARADOX-IV cluster, which is a part of the PARADOX supercomputing facility. This cluster is comprised of computing nodes with two Intel Xeon E5-2670 Sandy Bridge CPUs (with a total of $2 \times 8 = 16$ cores), with 32 GB of RAM and one Nvidia Tesla M2090 GPU with 6 GB of GPU RAM, each connected by InfiniBand QDR interconnect. We used Intel's compiler (version 2016) to compile the serial and OpenMP programs, and CUDA 7.5 for the GPU portions of the CUDA and hybrid programs. MPI-based implementations were compiled with Open MPI (version 1.10), which itself relied on underlying Intel and CUDA compilers. In the case of Hybrid/MPI programs, we performed tests using both FFTW and our own transpose routines. We found the minimal execution times to be about the same for both approaches, but the FFTW transpose would sometimes exhibit very bad performance due to the creation of suboptimal communication plan. For this reason, the execution times reported for Hybrid/MPI implementation are obtained using only our own transpose routines.

The base of all performance evaluations was the measured execution time of critical regions of the programs, i.e., the portions performing wave function propagation in imaginary or in real time. This measurement excluded the time spent in other parts of the programs, e.g., initialization of OpenMP/CUDA/MPI environment, memory allocation and deallocation, creation and destruction of FFTW plans, initialization of variables and I/O operations. Measuring average execution time of a single iteration of the main time-propagation loop allows us to predict the performance and total execution time of a given simulation, as the number of iterations is specific to the problem at hand and may vary significantly between different simulations. All measurements were collected using high precision timers based on `clock_gettime` POSIX function on the CPU side, and CUDA event API on the GPU side. The execution time of a single iteration of serial programs depends on the mesh size, controlled by variables `Nx`, `Ny` and `Nz`. For parallel programs, we can measure speedup and scaling efficiency as a function of the varying number of processing elements (OpenMP threads or MPI processes).

We tested performance of 1D, 2D and 3D programs on a range of mesh sizes, for a varying number of OpenMP threads and MPI processes, as shown in Table 8.2. Mesh sizes were chosen from the corresponding range, and we did not focus solely on mesh sizes which maximize performance of the programs, to obtain a more realistic assessment. We varied the number of OpenMP threads from 1 to 16, in increments of one thread. Similarly, we tested the MPI-based implementations by varying number of MPI processes, each bound to a different cluster node. In this way we tested MPI-based implementations on 2, 4, 8, 12, 16, 20, 24, 28 and 32 computing nodes. Note that varying the number of processing elements is not applicable to the single-GPU implementations of the programs, as they always use all available processing resources on a GPU. Only one of the two 1D and 2D programs were tested (corresponding to x direction in 1D and x-y plane in 2D), because, performance-wise, there is no difference between them (e.g., imag1dX-th vs. imag1dZ-th).

The main performance indicator is the execution time, the wall-clock time of one iteration of the main loop, averaged over 5 executions of 1000 iterations, reported in milliseconds. Using the results obtained, we calculated the speedup of all programs compared to the published serial C implementation. We were also interested in examining the scaling efficiency, or *scalability*, of the OpenMP and MPI programs. We tested both *strong* scaling, when the mesh size stays the same but the number of processing elements varies, and *weak* scaling, when the amount of work each processing element performs stays the same while the number of processing elements increases. More formally, given the execution time of a single iteration of serial programs ($T(1)$), and the corresponding execution time for parallel programs performed with N processing elements ($T(N)$), we calculated speedup as $S(N) = T(1)/T(N)$ and strong scaling efficiency as $E(N) = S(N)/N$. Weak scaling is computed as $E_W(N) = T_W(1)/T_W(N)$, where $T_W(1)$ is the execution time of a program using single processing element performing the work assigned to it, while $T_W(N)$ is the

Table 8.2: Performance testing matrix, showing the mesh sizes and numbers of processing elements (threads or processes) used to test the programs, as well as the baseline program used for comparison.

Program	Mesh size		Processing elements		Baseline
	Min	Max	Min	Max	
OpenMP programs					
imag1dX-th	1000	1000000	1	16	imag1d
real1dX-th	1000	1000000			real1d
imag2dXY-th	1000×1000	15000×15000	1	16	imag2dXY
real2dXY-th	1000×1000	13000×13000			real2dXY
imag3d-th	$50 \times 50 \times 50$	$800 \times 800 \times 800$	1	16	imag3d
real3d-th	$50 \times 50 \times 50$	$800 \times 800 \times 800$			real3d
CUDA programs					
imag2dXY-cuda	1000×1000	15000×15000	1	1	imag2dXY
real2dXY-cuda	1000×1000	13000×13000			real2dXY
imag3d-cuda	$50 \times 50 \times 50$	$600 \times 600 \times 600$	1	1	imag3d
real3d-cuda	$50 \times 50 \times 50$	$540 \times 540 \times 540$			real3d
Hybrid programs					
imag2dXY-hetero	1000×1000	15000×15000	16+1	16+1	imag2dXY-th
real2dXY-hetero	1000×1000	13000×13000			real2dXY-th
imag3d-hetero	$50 \times 50 \times 50$	$800 \times 800 \times 800$	16+1	16+1	imag3d-th
real3d-hetero	$50 \times 50 \times 50$	$800 \times 800 \times 800$			real3d-th
OpenMP/MPI programs					
imag3d-mpi	$480 \times 480 \times 250$	$1920 \times 1920 \times 960$	1×16	32×16	imag3d-th
real3d-mpi	$480 \times 480 \times 250$	$1920 \times 1920 \times 960$			real3d-th
CUDA/MPI programs					
imag3d-mpicuda	$480 \times 480 \times 250$	$1920 \times 1920 \times 960$	1	32	imag3d-cuda
real3d-mpicuda	$480 \times 480 \times 250$	$1920 \times 1920 \times 960$			real3d-cuda
Hybrid/MPI programs					
imag3d-mpihetero	$480 \times 480 \times 250$	$1920 \times 1920 \times 960$	$1\times(16+1)$	$32\times(16+1)$	imag3d-hetero
real3d-mpihetero	$480 \times 480 \times 250$	$1920 \times 1920 \times 960$			real3d-hetero

execution time of a program using N processing elements performing N times more work. We achieve this by increasing the mesh size.

8.3 Performance test results and modeling of single node programs

In this section we present the results obtained for single computing node OpenMP, CUDA and hybrid programs, and compare them to the previously published [13] serial implementation.

Strong scaling performance test results for the OpenMP-based implementation using methodology described in the previous section are given in Table 8.3 and Figure 8.3. They show the obtained execution times, speedups and strong scaling efficiencies for different number of OpenMP threads. Columns $N_{\text{th}} = 1$, $N_{\text{th}} = 2$, $N_{\text{th}} = 4$, $N_{\text{th}} = 8$ and $N_{\text{th}} = 16$ in Table 8.3 correspond to the number of threads used, while the last column shows the obtained speedup $S(16)$ with 16 OpenMP threads compared to one OpenMP thread. Strong scaling efficiency $E(N_{\rm th}) = S(N_{\rm th})/N_{\rm th}$ in Figure 8.3 is calculated as a fraction of the obtained speedup compared to a theoretical maximum. The mesh size used in 1D is 10^5, in 2D $10^4 \times 10^4$, while in 3D the mesh size is $480 \times 480 \times 480$. Execution times and speedups of imag1dZ-th, real1dZ-th, imag2dXZ-th, and real2dXZ-th (not reported here) are similar to those of imag1dX-th, real1dX-th, imag2dXY-th, and real2dXY-th, respectively.

Table 8.3: Wall-clock execution times of a single iteration of the main time-propagation loop of single-node OpenMP programs (in milliseconds) for different number of OpenMP threads $N_{\rm th}$ and speedup $S(16)$ in strong scaling tests. The speedup is calculated w.r.t. the execution times of previously published serial versions of programs [13], given in the second column.

Program	Serial	$N_{\rm th} = 1$	$N_{\rm th} = 2$	$N_{\rm th} = 4$	$N_{\rm th} = 8$	$N_{\rm th} = 16$	$S(16)$
imag1dX-th	9.1	7.1	4.7	3.4	2.9	2.8	2.5
real1dX-th	15.2	14.2	10.5	8.2	7.3	7.2	2.0
imag2dXY-th	13657	7314	4215	2159	1193	798	9.2
real2dXY-th	17281	11700	6417	3271	1730	1052	11.1
imag3d-th	16064	9353	5201	2734	1473	888	10.5
real3d-th	22611	17496	9434	4935	2602	1466	11.9

The change from C2C to R2C FFT routine has a big impact on the execution time of single-threaded ($N_{\rm th} = 1$) programs compared to the previous serial programs. As we can see from the table, these improvements alone yield a speedup of 1.3 to 1.9 in 2D and 3D programs, and somewhat smaller speedup for 1D programs, 1.1 to 1.3. The use of additional threads brings about further speedup (reported in the last column) of 2 to 2.5 for 1D programs, and 9 to 12 for 2D and 3D programs. In Figure 8.3 we see that the efficiency rapidly decreases for 1D programs, even though speedup increases with the number of threads used. This is expected, as parts of the algorithm dealing with the recursive relations for calculation of the CN coefficients are inherently serial. In 1D, already with $N_{\rm th} = 4$ threads we almost achieve the maximal speedup, while still keeping the efficiency around 50%. We also see that, as expected, speedup and efficiency of multidimensional programs behave quite well as we increase the numbers of threads. In particular, we note that the efficiency always remains above 60%, making the use of all available CPU cores worthwhile.

From Figure 8.3 we observe that the speedup of 1D programs saturate quickly due to inherent serial nature of the portion of the algorithm, while in 2D and 3D the speedup behaves almost linearly. Despite their obvious differences, all curves in Figure 8.3 can be successfully modeled based on Amdahl's law [93]. Namely, the measured execution time $T(N_{\rm th})$ of one iteration of the main loop can be expressed as

$$T(N_{\rm th}) = T(1)\left(s + \frac{p}{N_{\rm th}}\right), \tag{8.4}$$

where $N_{\rm th}$ is the number of threads used, $T(1)$ is the execution time of a single-threaded run, s is the serial fraction of the loop code, and p is the corresponding parallel fraction. By definition, $s + p = 1$, and therefore the speedup can be modeled by

$$S(N_{\rm th}) = \frac{T(1)}{T(N_{\rm th})} = \frac{1}{1 - p + p/N_{\rm th}}, \tag{8.5}$$

where the parallel fraction of the main loop code p is the only fit parameter. Table 8.4 gives the obtained model fit parameter values for the data from Figure 8.3. As we can see, the fits match

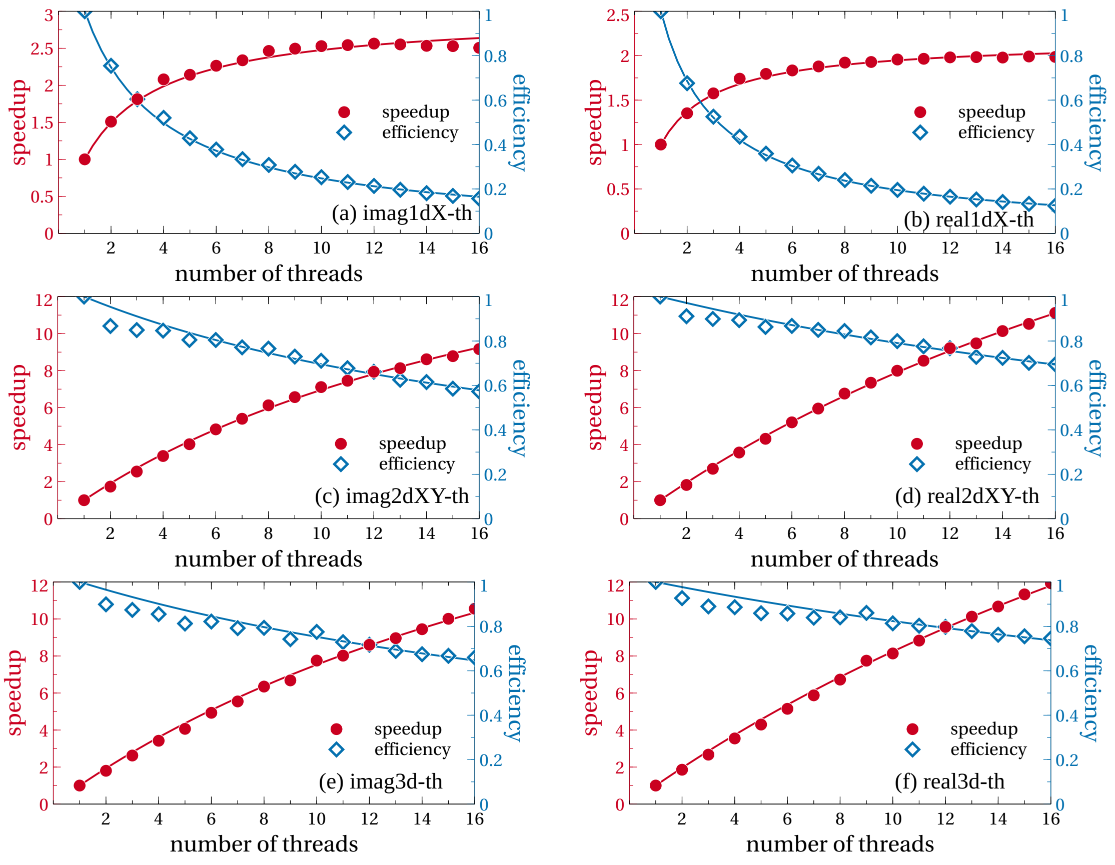


Figure 8.3: Speedup in the execution time and strong scaling efficiency of OpenMP programs compared to single-threaded runs for: (a) imag1dX-th, (b) real1dX-th, (c) imag2dXY-th, (d) real2dXY-th, (e) imag3d-th, (f) real3d-th. Solid lines represent fits to measured data, where fit model functions are given in the text and obtained fit parameters are listed in Table 8.4. Note that the model parameter p is fitted only to the speedup data, and then used to plot the efficiency model curve.

the obtained measurement data very well. Note that the efficiency can be expressed in terms of the speedup model

$$E(N_{\mathrm{th}}) = \frac{S(N_{\mathrm{th}})}{N_{\mathrm{th}}} = \frac{1}{N_{\mathrm{th}}} \frac{1}{1 - p + p/N_{\mathrm{th}}}, \tag{8.6}$$

and is not fitted independently. Figure 8.3 shows that this model with the parameter p fitted on the speedup data matches very well with the efficiency data points.

Table 8.4: Values of obtained strong scaling model fit parameter p and the estimated fit errors for OpenMP-parallelized programs.

Program	p	Δp
imag1dX-th	0.662	0.004
real1dX-th	0.541	0.003
imag2dXY-th	0.9514	0.0007
real2dXY-th	0.9706	0.0004
imag3d-th	0.9635	0.0008
real3d-th	0.9762	0.0006

From the obtained values of the parallel fraction of the algorithm, we see that 2D and 3D programs are almost ideally parallelizable, with p over 95%. In 1D case, however, the parallel fraction is around 66% for imaginary-time propagation and around 54% for real-time propagation, due to the fact that calculation of CN coefficients in the function `calclux` cannot be parallelized. Note that the parallel fraction of imaginary-time 1D program is higher than that of the real-time 1D program due to the larger amount of arithmetic operations required to process complex-valued data in the real-time version of the serial function `calclux`.

OpenMP programs were also tested for weak scalability. We were mostly interested in 3D variants of the programs, which we tested by fixing the amount of work to 6,912,000 spatial points of the mesh, which corresponds to a mesh size of $240 \times 240 \times 120$. By increasing the number of OpenMP threads, we also increase the mesh size to be the multiple of 6,912,000. This means that the mesh has to be increased in such a way that, when divided among the threads, each thread gets 6,912,000 spatial points to process. The mesh sizes that satisfy this requirement cannot always be obtained by multiplying all three array dimensions with the same number, so we have to work with mesh sizes in which the size of each dimension may be different. For such mesh sizes, we tested all possible combinations, however no significant difference has been observed. Table 8.5 contains mesh sizes we used for testing for $N_{\text{th}} = 1, 2, 4, 8, 12, 16$ threads, along with the execution times (in milliseconds) and the weak scaling efficiency. The testing was also done for $N_{\text{th}} = 6$ and 10 threads, but the corresponding mesh sizes are omitted from the table for brevity. However, Figure 8.4 presents complete data collected in this test, for all N_{th} values, averaged over mesh sizes used. The figure shows that the real-time 3D programs have better weak scaling efficiency, which is about 75% at 16 threads, while imaginary-time programs demonstrate smaller, but still significant efficiency of about 60%.

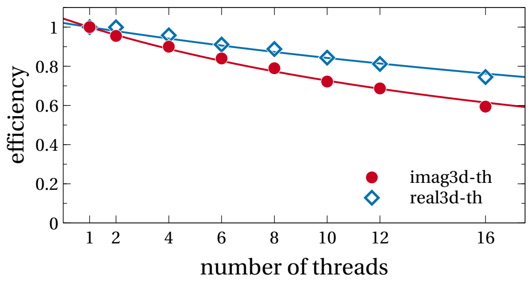


Figure 8.4: Weak scaling efficiency of OpenMP-parallelized 3D programs, averaged over all mesh sizes tested for each value of N_{th}. Solid lines represent fits to measured data, where fit model functions are given in the text.

To model weak scaling efficiency, we compare execution times of a single iteration of the main loop $T_W(N_{\text{th}})$, performed with N_{th} threads, where for each value of N_{th} the total amount of work is N_{th} times the work being performed in a single-threaded run, i.e., the amount of work per thread is constant. The weak scaling efficiency is defined as

$$E_W(N_{\text{th}}) = \frac{T_W(1)}{T_W(N_{\text{th}})}\,, \tag{8.7}$$

where $T_W(1) = T(1)$ is the execution time of a single-threaded run for a given workload (in our case, 6,912,000 spatial points). The expected execution time of a workload assigned to a simulation with

Table 8.5: Weak scaling efficiency of OpenMP-parallelized 3D programs. Wall-clock execution times T_W are given in milliseconds, efficiencies E_W in percents.

Mesh size	imag3d-th		real3d-th	
	T_W	E_W	T_W	E_W
$N_{\text{th}} = 1$				
$120 \times 240 \times 240$	699	98.2	1259	99.5
$240 \times 120 \times 240$	711	96.5	1271	100
$240 \times 240 \times 120$	686	100	1275	98.2
$N_{\text{th}} = 2$				
$240 \times 240 \times 240$	718	95.5	1254	99.9
$N_{\text{th}} = 4$				
$240 \times 240 \times 480$	755	90.9	1312	95.5
$240 \times 480 \times 240$	778	88.2	1301	96.3
$480 \times 240 \times 240$	754	91	1310	95.6
$N_{\text{th}} = 8$				
$240 \times 480 \times 480$	875	78.4	1405	89.9
$480 \times 240 \times 480$	847	81	1393	89.1
$480 \times 480 \times 240$	886	77.5	1431	87.7
$N_{\text{th}} = 12$				
$360 \times 480 \times 480$	1010	67.9	1532	81.7
$480 \times 360 \times 480$	981	69.9	1514	82.7
$480 \times 480 \times 360$	1004	68.4	1581	79.2
$N_{\text{th}} = 16$				
$480 \times 480 \times 480$	1155	59.4	1680	74.5

N_{th} threads, executed in a single-threaded run, is $N_{\text{th}}T(1)$. In weak scaling tests, this workload is executed with N_{th} threads and therefore the expected execution time can be modeled by

$$T_W(N_{\text{th}}) = N_{\text{th}}T(1)\left(1 - p + \frac{p}{N_{\text{th}}}\right) . \tag{8.8}$$

According to this argument, we model the weak scaling efficiency by a single-parameter function

$$E_W(N_{\text{th}}) = \frac{1}{p + (1-p)N_{\text{th}}} . \tag{8.9}$$

We fitted this model to data presented in Figure 8.4 and the obtained fit parameters are given in Table 8.6. As we can see from the figure, the above model is an excellent fit to experimental data, and both 3D OpenMP-parallelized programs have high (above 95%) parallel fraction of the code, in agreement with the results of strong scaling tests (Table 8.4).

Table 8.6: Values of obtained weak scaling model fit parameter p and the estimated fit errors for OpenMP-parallelized programs.

Program	p	Δp
imag3d-th	0.958	0.002
real3d-th	0.9792	0.0009

Overall, we conclude that imaginary-time OpenMP-parallelized programs show smaller speedup and efficiency across all tests, except in 1D. This can be attributed to the additional step of wave function normalization in each iteration of the imaginary-time propagation, as well as the fact that real-time programs work with complex-valued data that require more arithmetic operations for the same mesh size. Since much of the computation inside loops requires simple arithmetic, the throughput of the CPU is often not fully exploited in imaginary-time programs, thus the pressure of memory bandwidth makes these programs less efficient. The real-time programs are also affected by this, however to a somewhat lesser extent.

Next, we consider the CUDA implementation of the shared memory algorithm. GPU functions as a single processing element, therefore we cannot test CUDA implementation by varying both the mesh size and the number of processing elements. However, just varying the mesh size gives us valuable insight into the behavior of this implementation, due to the difference in programming models and the libraries used. Tables 8.7 and 8.8 show the execution times (in milliseconds) for a number of mesh sizes tested, as well as the average speedup compared to the serial programs [13]. Figure 8.5 shows the speedup obtained for all mesh sizes tested, and red horizontal lines represent average speedups obtained for each program. Note that the dispersion of data is due to the use of `FFTW_ESTIMATE` flag in library calls to FFTW in the serial programs. Use of this flag results in a choice of suboptimal FFT algorithm for some mesh sizes. The vertical lines in Figure 8.5 denote the change in `POTMEM` parameter. The speedups left of the first vertical line are obtained with `POTMEM=2` and thus demonstrate the best speedup. Second group of results, between the two vertical lines, is obtained with `POTMEM=1`, and we note that the speedup decreases slightly, while the results right of the second vertical line are obtained with `POTMEM=2`, and show the smallest speedup due to the use of mapped memory. The 2D programs in x-z plane exhibited very similar performance to those of x-y plane, and therefore we did not include them in the figures.

Table 8.7: Wall-clock execution times of a single iteration of the main time-propagation loop of single-node 2D CUDA programs (in milliseconds) for different mesh sizes, and average speedup w.r.t. to the execution times of serial 2D programs [13].

Program	2000^2	4000^2	6000^2	8000^2	10000^2	12000^2	Avg. speedup
imag2dXY-cuda	24.1	104.3	235.2	386.1	657.1	1150.4	10
real2dXY-cuda	29.9	112.4	266.4	444.0	749.0	1528.3	14

Table 8.8: Wall-clock execution times of a single iteration of the main time-propagation loop of single-node 3D CUDA programs (in milliseconds) for different mesh sizes, and average speedup w.r.t. to the execution times of serial 3D programs [13].

Program	100^3	200^3	300^3	400^3	500^3	Avg. speedup
imag3d-cuda	10.6	79.3	298.8	674.5	1260.2	7.1
real3d-cuda	10.9	84.1	302.5	682.4	1467.4	13.5

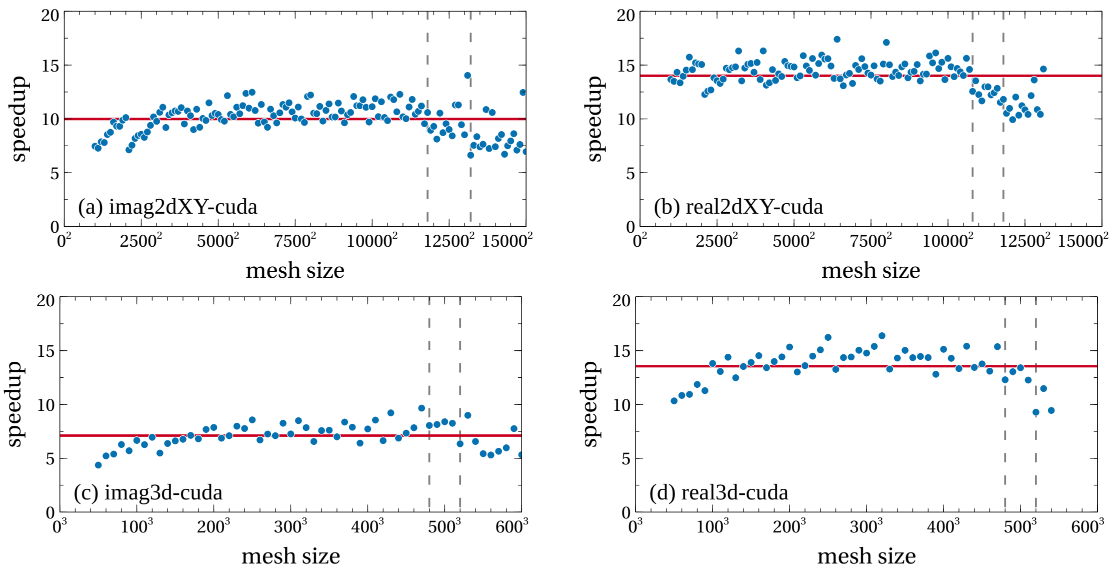


Figure 8.5: Speedup in the execution time of CUDA programs compared to the serial programs for all tested mesh sizes: (a) imag2dXY-cuda, (b) real2dXY-cuda, (c) imag3d-cuda, (d) real3d-cuda. Red horizontal lines represent the average speedup, while dashed vertical lines represent different values of `POTMEM` parameter (see text).

For small mesh sizes, the GPU remains underutilized, resulting in a smaller speedup. For large mesh sizes, where the GPU memory usage approaches the limit, we also see declining speedup. This is due to the inevitable use of `POTMEM` parameter, which keeps some arrays in the host memory when they cannot fit in the memory of the GPU. Overall, the CUDA implementation shows execution times similar to the OpenMP implementation for imaginary-time propagation, and slightly lower for real-time propagation. We stress that these results strongly depend on the type of GPU used and that the obtained speedups may be even better for newer-generation GPUs (e.g., *Kepler*-, *Maxwell*-, *Pascal*- and *Volta*-based GPUs).

Hybrid OpenMP/CUDA implementation was tested on a range of mesh sizes, similarly to the OpenMP- and CUDA-based implementations. Since OpenMP implementation showed that all cores should be used for 2D and 3D programs, we kept the number of OpenMP threads fixed at 16 in our hybrid algorithm. The parameters governing the amount of data offloaded to GPU were optimized using our GA method from Section 8.1. An illustrative subset of the results for 3D programs is shown in Table 8.9, which gives execution times for a single iteration of the main loop, for different mesh sizes. A comparison of execution times of hybrid and pure OpenMP programs with $N_{\mathrm{th}} = 16$ threads is given in Figure 8.6(a), while Figure 8.6(b) compares the performance of hybrid and pure CUDA programs, for all tested mesh sizes.

Table 8.9: Wall-clock execution times of a single iteration of the main time-propagation loop of single-node 3D hybrid programs (in milliseconds) for different mesh sizes. Offload parameters are optimized using the GA optimization method.

Program	80^3	200^3	320^3	440^3	560^3	680^3	800^3
imag3d-hetero	6.3	67.1	271.4	678.9	1493.3	2750.9	4461.6
real3d-hetero	8.0	94.7	374.1	922.8	1985.2	4020.2	6676.6

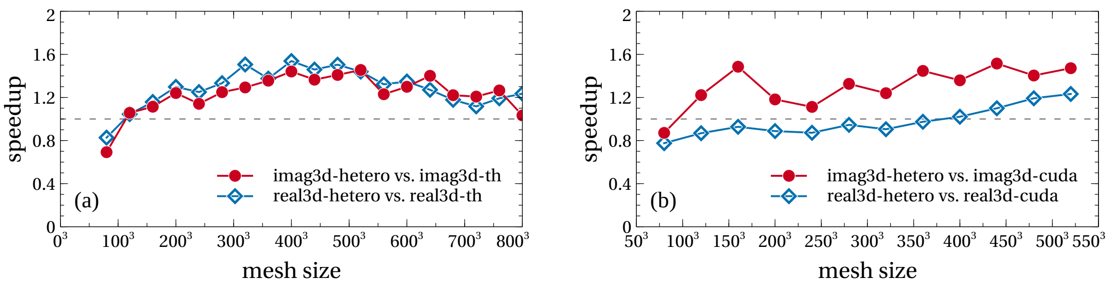


Figure 8.6: Speedup in the execution time of 3D hybrid programs compared to: (a) OpenMP programs with $N_{\rm th} = 16$ threads, (b) CUDA programs. Dashed horizontal lines correspond to a speedup value $S = 1$.

From Figure 8.6 we see that the hybrid implementation outperforms the OpenMP one for all mesh sizes except for the smallest one. The same applies to the comparison of imaginary-time hybrid and CUDA programs, while real-time hybrid program outperforms the corresponding CUDA program only for mesh sizes larger than $400\times400\times400$. Although one would expect that the speedup of the optimized hybrid algorithm is always equal to or larger than one, we see that this is not the case for all mesh sizes in Figure 8.6. This is due to the fact that the hybrid FFT algorithm is always employed in hybrid programs, meaning that even if no or all data are offloaded to GPU, the splitting of FFT will still take place. Therefore, in these limiting cases we do not obtain pure OpenMP or CUDA algorithms that would yield maximal performance. Splitting the computation of FFT along one direction in `calcpsidd2` function disables some of the potential optimizations that libraries like FFTW and cuFFT exploit, and introduces data copies between host and device which cannot be fully offset using CUDA streams. The hybrid algorithm can compensate for this if the amount of offloaded data to GPU is sufficiently large, which is not the case for the smallest mesh size.

Given that pure CUDA programs performing imaginary-time propagation demonstrated smaller speedup than the corresponding OpenMP programs (Figure 8.5 vs. Figure 8.3), it is expected that CUDA portion of the hybrid imaginary-time propagation programs would yield smaller improvement, which is evident when we compare hybrid implementation with the OpenMP one in Figure 8.6(a), except for the largest mesh sizes. We can reach the same conclusion if we consider that, in real-time propagation, CUDA implementation shows the highest speedup of all single-node programs, and therefore the corresponding hybrid programs show better speedup than imaginary-time propagation programs parallelized with OpenMP. On the other hand, when compared with the CUDA implementation in Figure 8.6(b), the situation is reversed, and we see much better performance of the hybrid imaginary-time propagation programs, while the hybrid real-time propagation programs achieve speedup larger than one only for mesh sizes larger than $400 \times 400 \times 400$. This result can be attributed to the fact that in real-time propagation the amount of data copied between host and device is larger, given that some of the copied arrays are complex-valued.

Also, we can observe that the speedup in Figure 8.6(a) declines with mesh size. This is due to the memory saturation of the GPU device, as only a small portion of the data can be offloaded when the mesh is large. The amount of data offloaded to GPU for all tested mesh sizes is given in Table 8.10, where we can see that the total amount of data processed by the GPU declines with increasing mesh size. As a result, the computation is unbalanced, given that the host has to work with a much larger portion of the data. This is reflected in the optimization step, where our GA quickly converges to the highest possible mesh size that can be offloaded to the device, implying that a greater amount of device memory would be required to get a balanced computation.

Table 8.10: Optimal fraction of total data offloaded to GPU for different mesh sizes in hybrid 3D programs, obtained using the GA optimization method.

Mesh size	GPU portion of data [%]							
	imag3d-hetero				real3d-hetero			
	PS_1	PS_2	PS_3	PS_4	PS_1	PS_2	PS_3	PS_4
$80 \times 80 \times 80$	20	3	60	33	23	3	40	10
$120 \times 120 \times 120$	12	10	25	28	23	10	25	27
$160 \times 160 \times 160$	15	7	31	24	44	19	34	23
$200 \times 200 \times 200$	38	27	37	28	44	26	46	26
$240 \times 240 \times 240$	38	28	40	30	38	28	47	25
$280 \times 280 \times 280$	40	31	30	33	46	22	37	25
$320 \times 320 \times 320$	33	38	29	36	45	28	69	44
$360 \times 360 \times 360$	41	33	27	36	49	33	43	31
$400 \times 400 \times 400$	38	25	30	38	46	24	47	21
$440 \times 440 \times 440$	31	35	29	39	45	35	43	32
$480 \times 480 \times 480$	16	19	22	41	45	33	62	35
$520 \times 520 \times 520$	40	31	28	46	45	25	40	37
$560 \times 560 \times 560$	38	41	30	43	36	24	35	32
$600 \times 600 \times 600$	37	27	30	43	29	29	29	28
$640 \times 640 \times 640$	36	40	30	43	24	22	24	23
$680 \times 680 \times 680$	26	38	28	38	20	16	20	20
$720 \times 720 \times 720$	23	26	25	32	17	13	17	17
$760 \times 760 \times 760$	21	22	25	27	14	14	14	13
$800 \times 800 \times 800$	19	23	23	21	11	8	12	12

To make an optimal choice of programs to be used on a single node for a particular hardware platform, one has to perform detailed tests using the methodology presented in Section 8.2. The results presented in this section apply to hardware comparable to the one available at PARADOX supercomputing facility, but one can expect similar behavior for all modern types of CPU and GPU.

8.4 Performance test results and modeling of MPI programs

Following on from the previous section, here we present the results obtained for MPI-parallelized programs, executed on a computer cluster consisting of varying number of computing nodes, and compare them with the corresponding single-node programs.

MPI programs are highly dependent on the configuration of the cluster, mainly on the speed of interconnect, but also on the distribution of processes and threads, NUMA configuration, MPI configuration, etc. Getting the best performance out of the programs requires some experimentation with several different configurations. The results presented are obtained without extensive tuning of the cluster or MPI runtime, with the aim to show the base performance.

Strong scaling of MPI-based implementations is tested in a similar way to the single-node testing, by varying number of cluster nodes N_p from 2 to 32. On each node, a single MPI process is launched, which then uses the node's resources, either by further spawning 16 OpenMP threads in case of OpenMP/MPI and Hybrid/MPI implementations, or by invoking the CUDA kernels utilizing the node's GPU in case of CUDA/MPI implementation. In this test, the baseline used for comparison was the equivalent single-node implementation, i.e., OpenMP/MPI programs are compared with OpenMP programs executing with $N_\mathrm{th} = 16$ threads, CUDA/MPI programs are compared to the single-node pure CUDA programs, and Hybrid/MPI programs is compared to the single-node hybrid programs.

Mesh size we use in this test is $480 \times 480 \times 250$. Since the values of `Nx` and `Ny` parameters must be divisible by the number of processes N_p, this mesh cannot be distributed over 28 processes. In this case we use slightly modified mesh, taking into account the multiple of 28 that is closest to 480, yielding a mesh size of $476 \times 476 \times 250$. Data is never distributed along z direction, so no such requirement exists for `Nz`. The resulting mesh has slightly fewer spatial points, and thus less work per process, potentially allowing for a better performance, but in our tests the smaller mesh did not have a significant impact on the measured execution time.

Note that strong scaling tests, where the mesh size is fixed, inevitably lead to saturation and decrease in measured speedup and efficiency values. Whatever the chosen mesh size, increase in the number of cluster nodes used will eventually yield insufficient amount of work per MPI process, such that communication will start to dominate over computation. Therefore, in our strong scaling tests we can expect that execution times initially decrease with increasing values of N_p, but eventually performance of MPI-based programs will decline. This issue is addressed by considering weak scaling tests.

The illustrative portion of obtained execution times of a single iteration of the main loop for the three MPI-parallelized implementations are shown in Table 8.11, together with the execution times of the corresponding baseline single-node program. Columns $N_\mathrm{p} = 4$, $N_\mathrm{p} = 8$, $N_\mathrm{p} = 16$, $N_\mathrm{p} = 24$ and $N_\mathrm{p} = 32$ correspond to the number of cluster nodes used, while the last column shows the obtained speedup $S(32)$ with $N_\mathrm{p} = 32$ nodes compared to baseline single-node programs.

Table 8.11: Wall-clock execution times of a single iteration of the main time-propagation loop of OpenMP/MPI, CUDA/MPI and Hybrid/MPI programs (in milliseconds) for different number of MPI processes N_p and speedup $S(32)$ in strong scaling tests. The speedup is calculated w.r.t. the baseline execution times of the corresponding single-node programs (OpenMP, CUDA and hybrid, respectively), given in the second column.

Program	Baseline	$N_\mathrm{p} = 4$	$N_\mathrm{p} = 8$	$N_\mathrm{p} = 16$	$N_\mathrm{p} = 24$	$N_\mathrm{p} = 32$	$S(32)$
imag3d-mpi	1124	541	262	134	89	64	17.5
real3d-mpi	2140	700	358	207	155	98	21.8
imag3d-mpicuda	579	438	210	103	71	59	9.8
real3d-mpicuda	800	609	291	142	95	79	10.1
imag3d-mpihetero	489	299	162	99	84	81	6.0
real3d-mpihetero	613	407	255	154	135	101	6.0

The maximal speedup of OpenMP/MPI implementation ranges from 17 for imaginary-time propagation to 22 for real-time propagation programs for $N_\mathrm{p} = 32$. The complete measurement results for speedup and efficiency are depicted in Figure 8.7(a) and 8.7(b), where we see that the speedup grows linearly with the number of nodes used, while the efficiency remains mostly constant in the range between 40% and 60%, thus making the use of OpenMP/MPI programs highly advantageous for simulations with large mesh size. In general, we can expect even better efficiency for larger mesh sizes.

Similar behavior is observed for CUDA/MPI implementation. The obtained speedup with $N_\mathrm{p} = 32$ nodes ranges from 9 to 10, with the slightly lower efficiency, between 30% and 40%, as shown in Figure 8.7(c) and 8.7(d). Even though the efficiency is lower for this implementation, the speedup still grows linearly and the execution times are lower than for the OpenMP/MPI implementation. This makes CUDA/MPI programs ideal choice for use on GPU-enabled computer clusters. Additional benefit of using CUDA/MPI programs is their low CPU usage (using only one CPU core per cluster node), allowing for the possibility that the same cluster nodes are used for other CPU-intensive simulations in a time-sharing fashion.

As we see in Figure 8.7(e) and 8.7(f), the linear growth of speedup is also present for the Hybrid/MPI implementation, however, as the amount of work per-process shrinks, the efficiency drastically drops down to 20% with $N_\mathrm{p} = 32$ nodes. For the mesh size used in this test, we observe that the Hybrid/MPI implementation performs very well on $N_\mathrm{p} < 16$ processes, providing the lowest execution times, but with $N_\mathrm{p} \geq 16$ processes its execution times become larger than for the CUDA/MPI implementation (especially for real-time propagation programs), and eventually even for the OpenMP/MPI implementation, as illustrated in Figure 8.8. This is again due to the insufficient amount of work each computing node performs after data are distributed among the MPI processes, which can be seen from the amount of data offloaded to GPU for all tested values of N_p in Table 8.12. Similar saturation in performance will eventually happen for the OpenMP/MPI and CUDA/MPI implementations, just for larger values of N_p.

We thus conclude that Hybrid/MPI implementation has the best performance of the three MPI-based implementations if the amount of work per process remains high enough to justify the use of hybrid algorithm. Otherwise, either of the other two MPI-based implementations should be considered first. The energy efficiency of this and other MPI-based implementations was not explored due to the difficulty in making precise measurements. If the energy consumption is not an issue, Hybrid/MPI implementation will yield the best performance, providing the cluster has powerful GPUs installed and the mesh size used is large enough.

We now model the speedup and strong scaling efficiency of MPI-based programs. In general, the execution time of a program can be expressed as

$$T(N_\mathrm{p}) = \alpha + \beta L + \gamma \frac{M}{N_\mathrm{p}} + \beta V \frac{M}{N_\mathrm{p}^2}\,, \tag{8.10}$$

where α represents the average time to perform serial portion of the code, L is the communication latency associated with one MPI message, β is the frequency of MPI messaging, γ is the average time to process one spatial point, V is related to data transfer speed (throughput), and M is the mesh size ($\mathtt{Nx} \times \mathtt{Ny} \times \mathtt{Nz}$). The communication overhead of one all-to-all message passing instance is equal to $L + VM/N_\mathrm{p}^2$, where each of N_p processes communicates its M/N_p part of the mesh evenly to all other processes, leading to a message size of M/N_p^2. Taking into account that in the strong

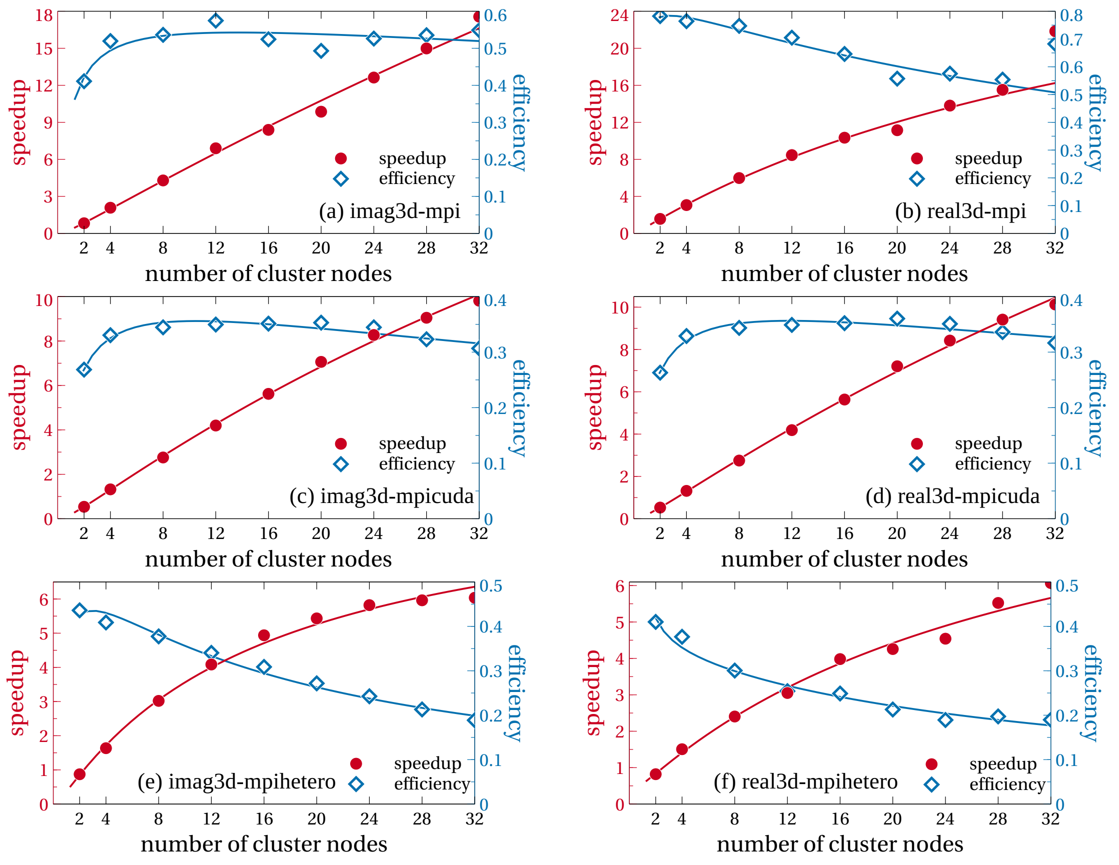


Figure 8.7: Speedup in the execution time and strong scaling efficiency of MPI-based programs compared to single-node runs: (a) imag3d-mpi, (b) real3d-mpi, (c) imag3d-mpicuda, (d) real3d-mpicuda, (e) imag3d-mpihetero, (f) real3d-mpihetero. Solid lines represent fits to measured data, where fit model functions are given in the text and obtained fit parameters are listed in Table 8.13. Note that the model parameters are fitted only to the speedup data, and then used to plot the efficiency model curve.

scaling tests the mesh size is fixed, the above model can be simplified to

$$T(N_\mathrm{p}) = T(1)\left(a + \frac{b}{N_\mathrm{p}} + \frac{c}{N_\mathrm{p}^2}\right), \tag{8.11}$$

while the speedup can be modeled by

$$S(N_\mathrm{p}) = \frac{T(1)}{T(N_\mathrm{p})} = \frac{1}{a + b/N_\mathrm{p} + c/N_\mathrm{p}^2}. \tag{8.12}$$

This model is fitted to the obtained strong scaling measurement data and the results are presented in Table 8.13, while the corresponding model curves are shown as solid lines in Figure 8.7. As we see, the proposed model agrees very well with the experimental data. The only exception is the speedup and efficiency of real-time OpenMP/MPI program (real3d-mpi) on $N_\mathrm{p} = 32$ processes, where FFTW library creates an optimal transform plan that works very well with the given mesh size. However, changing the mesh size even slightly gives the performance comparable to the model prediction. We note that the value of the model parameter c is negative for real-time Hybrid/MPI program (real3d-mpihetero), which should not be the case since is parameter is related to data

Table 8.12: Optimal fraction of total data offloaded to GPU per process for different values of N_p in Hybrid/MPI programs, obtained using the GA optimization method in strong scaling tests.

N_p	GPU portion of data [%]							
	imag3d-mpihetero				real3d-mpihetero			
	PS_1	PS_2	PS_3	PS_4	PS_1	PS_2	PS_3	PS_4
1	30	17.5	34.4	36.7	45.2	32.5	55	35
2	25	10	17.5	16.7	20.8	11.5	23.8	16
4	10	5.2	9.2	9.4	10	5.2	15	6.7
8	5	2.9	5.8	6.7	6.3	2.1	6.7	5
12	3.3	1.7	4.2	4.2	4	1.3	5	3.3
16	2.9	1.3	2.9	2.1	3.1	0.4	2.9	2.1
20	1.7	0.6	2.5	1	2.5	0.4	2.5	1.7
24	1.7	1.3	2.1	1.7	2.1	1.3	2.5	0.8
28	1.7	0.4	2.1	1.3	1.7	0.4	1.7	1.3
32	0.8	0.4	1.7	0.8	1.5	0.4	2.1	0.8

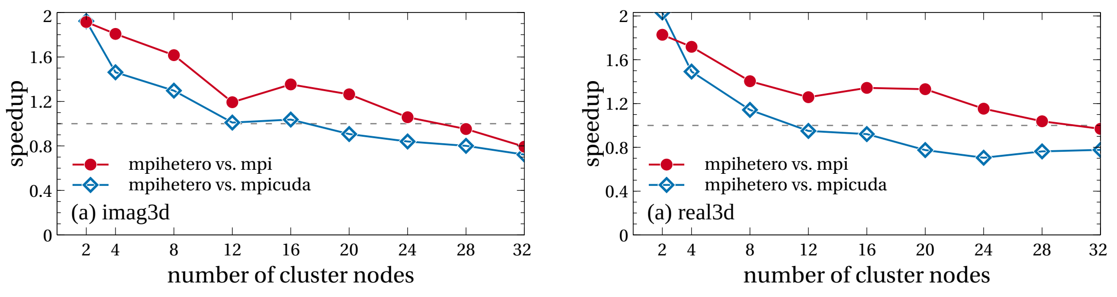


Figure 8.8: Speedup in the execution time of Hybrid/MPI programs in strong scaling tests compared to the other two MPI-based programs: (a) OpenMP/MPI, (b) CUDA/MPI.

transfer speed. However, due to uncertainty of the optimization choice by the GA method and the fact that the fraction of total data offloaded to GPU gradually decreases (Table 8.12), leading from real hybrid algorithm to almost pure OpenMP/MPI one with hybrid FFT, it is not surprising that the obtained value differs from expected. Taking into account relatively large fit error $\Delta c/c = 60\%$ for this parameter, we can still use expression (8.12) to model performance of Hybrid/MPI programs.

Next, we test weak scaling of MPI-based implementations. The same number of cluster nodes (and thus MPI processes) was used as in previous tests, while the starting mesh, which corresponds to a unit workload, had a size of $480 \times 480 \times 480$, amounting to 110,595,000 spatial points. This number of spatial points was kept constant per process, similarly to the weak test of OpenMP programs. There is an exception to this scheme, the case when programs are executed on $N_\mathrm{p} = 28$ nodes, as the scaled number of spatial points cannot be evenly distributed among 28 processes. In this case we compare the weak scalability by scaling up a starting mesh of $476 \times 476 \times 476$, which has 107,850,176 spatial points, just slightly less than for other values of N_p. The base of comparison was the execution time of a single iteration of the main loop with the corresponding MPI-based program running as a single process ($N_\mathrm{p} = 1$). While we used this configuration as the baseline, we do not recommend running MPI-based programs with $N_\mathrm{p} = 1$ processes, because no

Table 8.13: Values of obtained strong scaling model fit parameters a, b and c, as well as the estimated fit errors for MPI-parallelized programs.

Program	a	Δa	b	Δb	c	Δc
imag3d-mpi	0.008	0.006	1.6	0.2	1.4	0.4
real3d-mpi	0.026	0.006	1.1	0.1	0.2	0.3
imag3d-mpicuda	0.025	0.008	2.3	0.2	2.8	0.6
real3d-mpicuda	0.020	0.008	2.3	0.2	2.8	0.6
imag3d-mpihetero	0.104	0.009	1.7	0.2	0.9	0.5
real3d-mpihetero	0.09	0.02	2.7	0.3	-1.0	0.6

special handling of such case has been implemented. This means that MPI-parallelized programs are always transposing the data, which is unnecessary with $N_\mathrm{p} = 1$ as all data is local to the single process. Instead, we recommend using single-node variants of the programs outside cluster environment.

Table 8.14 lists mesh sizes used and execution times obtained for $N_\mathrm{p} = 1$, 2, 4, 8, 12, 16, 24 and 32 cluster nodes. We also tested weak scaling on $N_\mathrm{p} = 20$ and 28, but have excluded them from the table for brevity. A complete comparison is shown in Figure 8.9. The ordering of mesh dimensions had a slightly greater impact than with single-node OpenMP programs, varying up to 10% for the OpenMP/MPI programs, and less for the other two MPI-based implementations. This means that for meshes of the same size, thus implying equal work, execution times were mostly lower for those with larger values of `Nx` (e.g., the execution time for a $960 \times 480 \times 960$ mesh is lower than for a $480 \times 960 \times 960$ one). However, no distinct pattern emerges that would give us a clue as to which order is the most favorable. Upon inspection, we find that the difference in execution times is due to the different FFT plans employed by the FFTW and cuFFT libraries. The communication time remains mostly the same, as is expected since the amount of data exchanged is the same.

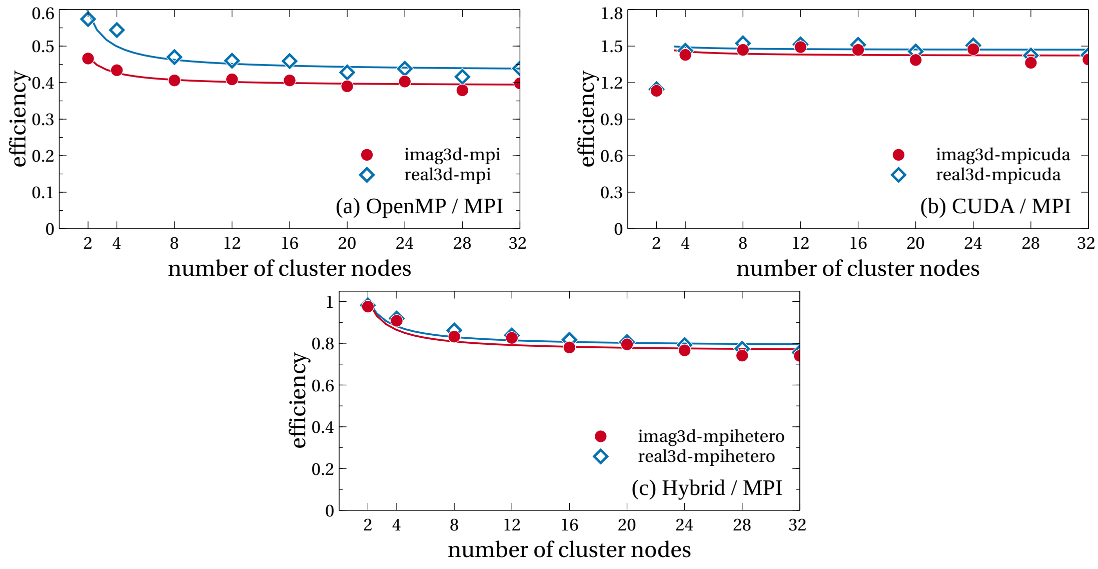


Figure 8.9: Weak scaling efficiency of three MPI-based implementations, averaged over all mesh sizes tested for each value of N_p. Solid lines represent fits to measured data, where fit model functions are given in the text.

Table 8.14: Weak scaling of MPI-based programs. Wall-clock execution times T_W are given in milliseconds, efficiencies E_W in percents.

Mesh size	imag3d-mpi		real3d-mpi		imag3d-mpicuda		real3d-mpicuda		imag3d-mpihetero		real3d-mpihetero	
	T_W	E_W	T_W	E_W	T_W	E_W	T_W	E_W	T_W	E_W	T_W	E_W
$N_\text{p} = 1$												
$480 \times 480 \times 480$	1758	100	2956	100	4338	100	6346	100	1971	100	2877	100
$N_\text{p} = 2$												
$480 \times 480 \times 960$	3858	45.6	4910	60.2	3785	114.6	5430	116.9	2044	96.4	2878	100
$480 \times 960 \times 480$	3704	47.5	5035	58.7	3825	113.4	5573	113.9	1997	98.7	2928	98.3
$960 \times 480 \times 480$	3758	46.8	5506	53.7	3871	112.1	5609	113.1	2019	97.6	2980	96.5
$N_\text{p} = 4$												
$480 \times 960 \times 960$	4267	41.2	6033	49.2	2995	144.8	4278	148.3	2149	91.7	3009	95.6
$960 \times 480 \times 960$	3877	45.4	5223	56.6	3042	142.6	4318	146.9	2218	88.9	3274	87.9
$960 \times 960 \times 480$	4000	44	5063	58.4	3078	140.9	4424	143.4	2148	91.8	3106	92.6
$N_\text{p} = 8$												
$960 \times 960 \times 960$	4327	40.6	6285	47	2952	146.9	4167	152.3	2369	83.2	3338	86.2
$N_\text{p} = 12$												
$1440 \times 960 \times 960$	4400	40	6729	43.9	2902	149.5	4254	149.2	2335	84.4	3504	82.1
$960 \times 1440 \times 960$	4268	41.2	6263	47.2	2953	146.9	4167	152.3	2412	81.7	3440	83.6
$960 \times 960 \times 1440$	4234	40.2	6299	46.9	2865	151.4	4153	152.8	2413	81.7	3351	85.9
$N_\text{p} = 16$												
$1920 \times 960 \times 960$	4356	40.4	6419	46	3001	144.5	4278	148.3	2392	82.4	3434	83.8
$960 \times 1920 \times 960$	4266	41.2	6376	46.4	2970	146.1	4170	152.2	2552	77.2	3525	81.6
$960 \times 960 \times 1920$	4234	40.2	6508	45.4	2888	150.2	4148	153	2639	74.7	3606	79.8
$N_\text{p} = 24$												
$1440 \times 1920 \times 960$	4677	40	6895	42.9	2924	148.4	4266	148.8	2552	77.3	3770	76.3
$1440 \times 960 \times 1920$	4796	38.3	7076	41.8	2984	145.4	4235	149.8	2611	75.5	3699	77.8
$1920 \times 1440 \times 960$	4470	40.6	6439	45.9	3007	144.2	4286	148.1	2546	77.4	3504	82.1
$1920 \times 960 \times 1440$	4450	40.8	6425	46	2984	145.3	4251	149.3	2551	77.3	3550	81.1
$960 \times 1440 \times 1920$	4364	41.1	6846	43.2	2879	150.7	4105	154.6	2682	73.5	3725	77.2
$960 \times 1920 \times 1440$	4346	41	6806	43.4	2878	150.7	4133	153.5	2501	78.8	3562	80.8
$N_\text{p} = 32$												
$1920 \times 1920 \times 960$	4452	39.5	6704	44.1	3183	136.3	4518	140.5	2585	76.3	3715	77.5
$1920 \times 960 \times 1920$	4520	38.9	6802	43.4	3163	137.2	4480	141.7	2632	74.9	3769	76.3
$960 \times 1920 \times 1920$	4301	40.9	6663	44.4	3014	143.9	4351	145.9	2776	71	3933	73.0

The results of weak scaling tests warrant further discussion as the execution times and plotted efficiencies appear misleading. On one hand, OpenMP/MPI programs show poor scaling results, achieving only 40% efficiency, while on the other hand the CUDA/MPI show efficiency well above 100%, a seemingly impossible result. The lower efficiency of the OpenMP/MPI programs is due to the very good performance of the baseline run. Here, the FFTW creates a very good FFT and transpose plans, which exploit the fact that all data are local, and thus have very good execution time. With $N_\mathrm{p} = 2$ nodes the execution time increases significantly, and efficiency drops to about 45-55%. However, adding more work and nodes has much smaller impact on efficiency, which only drops to 40-45% with $N_\mathrm{p} = 32$ nodes. If we use $N_\mathrm{p} = 2$ as the baseline, we see that the efficiency remains above 80%, a very good result.

In contrast to the OpenMP/MPI version, CUDA/MPI version performs very badly when executed on one node. Due to message sizes, MPI implementation used (Open MPI) relies on asynchronous copies through host memory [94] to perform the required communication. This results in bad performance due to a lack of overlap when copying data, as there is only one process involved. With $N_\mathrm{p} = 2$ processes the situation improves, as multiple streams are used, much like how we employ streams to overlap computation and data transfers in implementation of the hybrid algorithm. Between $N_\mathrm{p} = 4$ and 16 processes we get the best performance, and further increasing the number of processes only slightly reduces efficiency due to the size and amount of messages passed. If we compare the weak scaling efficiency with the best result as the baseline (obtained with $N_\mathrm{p} = 16$ processes), we see that the efficiency is lower than for the OpenMP/MPI version, due to the different transpose routine used and divided multidimensional FFT that has to be employed. However, we stress that, in terms of absolute execution times, CUDA/MPI is faster than OpenMP/MPI version for $N_\mathrm{p} \geq 4$.

The Hybrid/MPI version behaves as expected and demonstrates weak scaling efficiency of about 70-75% with $N_\mathrm{p} = 32$ nodes. Since data are transposed only in host memory, this version does not suffer the penalty of memory copies like the CUDA/MPI version, and therefore achieves much better efficiency with small number of cluster nodes. In terms of absolute execution times, Hybrid/MPI version is the fastest of all three implementations for $N_\mathrm{p} \geq 2$ and is therefore the algorithm of choice for distributed memory systems. Of course, it assumes previous optimization of offloading parameters using the GA method, which itself is time-consuming and has to be taken into consideration when making the choice. Table 8.15 provides data on the optimal fraction of total data offloaded to GPU in our weak scaling tests. As the number of processes increases, one expects that the amount of data offloaded to GPU remains constant, which is the case, as we see from the table. The only exceptions are when the value of `Ny` is greater than `Nx` where we see a larger fraction of data offloaded to GPU in PS_2 and PS_4, and when the value of `Ny` is smaller than `Nx` where we see a smaller fraction of data offloaded to GPU in PS_2 and PS_4.

Table 8.15: Optimal fraction of total data offloaded to GPU in weak scaling tests for different mesh sizes in Hybrid/MPI programs, obtained using the GA optimization method.

Mesh size	imag3d-mpihetero				real3d-mpihetero			
	PS_1	PS_2	PS_3	PS_4	PS_1	PS_2	PS_3	PS_4
$N_\mathrm{p} = 1$								
$480 \times 480 \times 480$	35	20	31	44	46	30	40	35
$N_\mathrm{p} = 2$								
$480 \times 480 \times 960$	46	33	33	47	38	29	37	38
$480 \times 960 \times 480$	50	67	17	47	45	60	19	40
$960 \times 480 \times 480$	50	15	70	34	43	15	67	35
$N_\mathrm{p} = 4$								
$480 \times 960 \times 960$	40	65	18	47	40	63	17	37
$960 \times 480 \times 960$	50	10	67	42	40	12	75	33
$960 \times 960 \times 480$	33	29	33	40	45	22	33	40
$N_\mathrm{p} = 8$								
$960 \times 960 \times 960$	53	25	33	47	40	27	40	28
$N_\mathrm{p} = 12$								
$1440 \times 960 \times 960$	37	20	53	45	42	18	60	63
$960 \times 1440 \times 960$	40	38	27	47	45	35	27	35
$960 \times 960 \times 1440$	50	25	38	45	45	25	40	40
$N_\mathrm{p} = 16$								
$1920 \times 960 \times 960$	47	13	67	27	42	13	80	30
$960 \times 1920 \times 960$	47	47	35	37	47	53	20	25
$960 \times 960 \times 1920$	50	20	33	33	47	20	40	27
$N_\mathrm{p} = 24$								
$1440 \times 1920 \times 960$	47	40	30	53	47	33	25	63
$1440 \times 960 \times 1920$	50	17	60	45	47	13	60	70
$1920 \times 1440 \times 960$	40	20	50	40	45	20	53	27
$1920 \times 960 \times 1440$	53	13	60	45	45	10	80	25
$960 \times 1440 \times 1920$	50	30	27	47	45	30	27	27
$960 \times 1920 \times 1440$	40	50	18	45	45	50	20	30
$N_\mathrm{p} = 32$								
$1920 \times 1920 \times 960$	47	27	33	30	47	17	40	27
$1920 \times 960 \times 1920$	50	13	60	40	47	10	80	40
$960 \times 1920 \times 1920$	47	47	40	33	47	40	23	27

To model the obtained weak scaling results, we start from equation (8.10) for the execution time of a single iteration of the main loop. In weak scaling tests, the mesh size is increased proportionally to the number of MPI processes N_p, and therefore the execution time is given by

$$T_W(N_\mathrm{p}) = \alpha + \beta L + \gamma \frac{M N_\mathrm{p}}{N_\mathrm{p}} + \beta V \frac{M N_\mathrm{p}}{N_\mathrm{p}^2} = \alpha + \beta L + \gamma M + \beta V \frac{M}{N_\mathrm{p}} \,. \tag{8.13}$$

The weak scaling efficiency

$$E_W(N_\mathrm{p}) = \frac{T_W(1)}{T_W(N_\mathrm{p})}\,, \tag{8.14}$$

can thus be modeled by

$$E_W(N_\mathrm{p}) = \frac{1}{a + b/N_\mathrm{p}}\,. \tag{8.15}$$

We fitted this model to data presented in Figure 8.9 and the obtained fit parameters are given in Table 8.16. Note that the model is fitted to CUDA/MPI scaling data only for $N_\mathrm{p} \geq 4$. The agreement between the model and measurement data is very good, although we note that the parameter b has negative values, contrary to its expected relation to the communication cost. Therefore, we conclude that the above expression can be successfully used for modeling of the performance of all three MPI-based implementations.

Table 8.16: Values of obtained weak scaling model fit parameters a and b, as well as the estimated fit errors for MPI-parallelized programs.

Program	a	Δa	b	Δb
imag3d-mpi	2.56	0.06	-0.9	0.3
real3d-mpi	2.32	0.05	-1.3	0.2
imag3d-mpicuda	0.70	0.02	-0.1	0.2
real3d-mpicuda	0.68	0.02	-0.1	0.2
imag3d-mpihetero	1.32	0.03	-0.6	0.2
real3d-mpihetero	1.27	0.03	-0.6	0.2

To summarize, in all test results of MPI-based programs, we note an expected declining of the scaling efficiency. This is due to the introduction of distributed transposes of data, creating overhead that negatively impacts scaling efficiency. It is most evident in both strong and weak scaling tests of the CUDA/MPI version, as the transpose algorithm is inferior to the one provided by FFTW, used in OpenMP/MPI implementation. In our tests, all three MPI versions of programs failed to achieve actual speedup ($S(N_\mathrm{p}) > 1$) on less than $N_\mathrm{p} = 4$ nodes, due to the introduction of these transpose routines. We therefore recommend using MPI versions only on $N_\mathrm{p} = 4$ or more cluster nodes.

8.5 Selecting optimal algorithm

In previous sections we presented results of detailed performance tests and associated models for all developed programs. Here we provide general guidelines for obtaining the best performance from each implementation.

We note that the extensive testing performed shows that the best performance can be achieved by evenly distributing the workload among the MPI processes and OpenMP threads, and by using mesh sizes which are optimal for FFT. In particular, the single-node OpenMP programs have the best performance if the number of spatial points in all directions, controlled by parameters `Nx`, `Ny` and `Nz`, is divisible by the number of OpenMP threads used. Similarly, the OpenMP/MPI implementation achieves the best performance if `Nx` and `Ny` are divisible by a product of the number of MPI processes and the number of OpenMP threads used.

On the other hand, CUDA implementation works best if all the data for a given mesh size can fit into the GPU memory with the `POTMEM` parameter set to 2. In the case of a small mesh size, CUDA programs may not be able to saturate all Streaming Multiprocessors (SM) of a given GPU, and in this case the OpenMP programs may be considered first. For CUDA/MPI programs, the best performance is achieved if `Nx` and `Ny` are divisible by a product of the number of MPI processes and the number of SMs in the GPU used. Note that CUDA-based implementations write output files (e.g., density profiles) by transferring data from GPU memory to host memory, where a single thread writes to a file. Therefore, these implementations may not be suitable for simulations that require the output to be written frequently, after a small number of time propagation steps.

Best performance of hybrid implementations can be obtained by following the same guidelines. Furthermore, we recommend using these programs with the optimization methods described in Section 8.1 to optimally divide the work between host and device. If manual tuning of offload parameters is required, we recommend that the guidelines above are followed for both CPU and GPU portions of the mesh. In the case of large disparity in the performance of host or device, hybrid versions will not provide the lowest execution times, and in such cases the pure CPU or GPU implementations with or without MPI could be better suited.

In addition to the guidelines for mesh sizes presented above, all programs benefit from the mesh size which is also optimal for FFT. The best FFT performance on CPU with FFTW library is obtained if `Nx`, `Ny` and `Nz` can be expressed as $2^a 3^b 5^c 7^d 11^e 13^f$, where e and f are either 0 or 1, and the other exponents are non-negative integer numbers [95]. Similarly, for cuFFT the best performance is achieved for transform sizes of the form $2^a 3^b 5^c 7^d$ [96]. In hybrid implementations, the same applies to the host and device portions of `Nx` and `Ny`, i.e., `cpuNx`, `cpuNy`, `gpuNx` and `gpuNy`.

Chapter 9

Demonstration of usability of developed programs

The programs developed as part of this thesis can be used to model and study a variety of systems. In this chapter we demonstrate the versatility of developed programs on a simulation of a BEC. We use the MPI-based programs to simulate the effects of a moving obstacle in an oblate atomic BEC. The obstacle, a repulsive Gaussian laser beam, moves through the condensate and sheds quantum vortices, elementary excitation of a superfluid [10, 11]. The vortices appear only when the obstacle is moving above some critical speed, consistent with the Landau's criterion of superfluidity. At low obstacle velocities above a critical value, vortex dipoles emerge, and as the speed is increased further, individual vortices and rotating vortex pairs are also formed.

In Section 9.1 we document how our programs can be used to simulate the experiment reported in Ref. [97], and compare experimental and numerical results. The experiment measured critical velocity for the emergence of vortex dipoles and rotating vortex pairs in a BEC of sodium atoms (^{23}Na), which do not exhibit the dipolar interaction. Therefore, in this section we also demonstrate how our programs can be modified to switch off the dipolar interaction term when solving GPE. The agreement of the results obtained numerically and experimental observations provides an external check of the correctness of our algorithms and their implementations.

Next, in Section 9.2, we investigate the formation of vortices in a dipolar BEC of dysprosium atoms (^{164}Dy). Our simulations follow the same methodology as the experiment with sodium atoms, only with atomic species exhibiting strong dipolar interaction. We study effects of the dipolar interaction on the critical velocity for the emergence of vortices, as well as the interplay between contact and dipolar interaction. The visualization extensions presented in Chapter 7 have proven to be indispensable during these simulations, as they allow much easier study of the results and control of the simulation.

9.1 Formation of vortices in BEC

BEC is a superfluid quantum liquid and one of its hallmarks are quantized vortices, which appear as elementary excitations of the system. Quantization of vortices is connected to excitation spectrum and, according to Landau's criterion, leads to the existence of a minimal velocity an obstacle moving through the superfluid has to have in order to generate such elementary excitations. This critical velocity v_c can be experimentally measured and, in principle, depends on the experimental protocol used.

Reference [97] reports the study of vortex formation in an oblate BEC of $N_{\rm at} = 3.2 \times 10^6$ sodium atoms. The trapping potential frequencies used are $(\omega_x, \omega_y, \omega_z) = 2\pi \times (9, 9, 400)$ Hz, and the s-wave scattering length determining the strength of contact interaction was $a_s = 51.9\, a_0$, where $a_0 = 0.0529$ nm is the Bohr radius. The moving obstacle is realized by a repulsive Gaussian laser beam corresponding to an additional potential of the form

$$V_{\rm B}(\mathbf{r}; t) = V_0 \exp\left\{ -2\frac{[y - y_0(t)]^2 + z^2}{\sigma^2} \right\}, \tag{9.1}$$

where V_0 represents the strength of the beam, σ is the $1/e^2$ beam waist, while $y_0(t) = vt$ determines the center of the beam $(0,\, y_0(t),\, 0)$. In our simulations, as in the experiment, the initial position of the beam is at the condensate center and then it moves with the velocity v along y direction for 24 μm. Afterwards, the beam is switched off linearly during 0.5 s. Illustration of the experimental setup is given in Figure 9.1. Using imaginary-time propagation, we calculate the ground state of the system with the trapping potential $V(\mathbf{r}) + V_{\rm B}(\mathbf{r}; t)$, where harmonic part $V(\mathbf{r})$ is defined by Eq. (2.2), with the frequencies given above. The experiment measured critical velocity for the emergence of vortex dipoles, pairs of vortices of the opposite sign. We numerically addressed this setup and calculated this critical velocity for the values of the parameters $\sigma = 1.3\, l$ and $V_0 = 250\, \hbar\tilde{\omega}$, where $l = 6.98777\, \mu$m is the harmonic oscillator length for the referent frequency $\tilde{\omega} = \omega_x$.

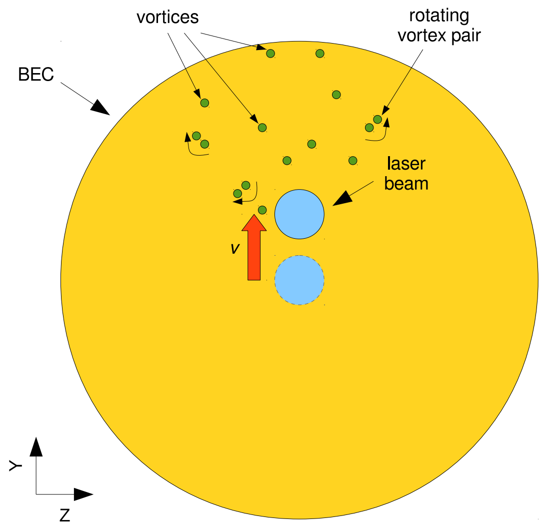


Figure 9.1: Illustration of the experimental setup used in Ref. [97] to study vortex formation in a BEC of sodium atoms. A repulsive Gaussian laser beam is initially at the center of the condensate and moves along y direction with a constant velocity v until it reaches its final position. The beam is then switched off during a period of 0.5 s. For sufficiently large velocity vortices and vortex pairs are generated.

Before it was possible to run simulations with the parameters described above, we had to make changes to the programs to model the experiment precisely. First, we had to exclude the dipolar interaction term from the propagation of the wave function, since sodium atoms have negligible dipole moment. While we can set the corresponding input parameters to zero, this would still result in unnecessary computation. A better approach is to remove (or comment out) the portions of the code relating to the calculation of the dipolar interaction term (calling of `calcpsidd2` function and, optionally, the associated allocations and deallocations), and adjust the `calcnu` function not to include the dipolar interaction term. This is a very simple change, resulting in modification of a few lines of the code, however, the performance impact is significant, due to the code calling the FFT functions no longer being part of the main loop. A single iteration of the main loop now performs roughly 30-40% faster.

Final change was related to the implementation of time-dependent potential $V(\mathbf{r}) + V_{\mathrm{B}}(\mathbf{r};t)$. Initialization of $V(\mathbf{r})$ remains the same, i.e., it is initialized before the main loop and stored in an array, while calculation of $V_{\mathrm{B}}(\mathbf{r};t)$ is implemented in a separate function that is called in each iteration within the main loop. Two distinct phases of $V_{\mathrm{B}}(\mathbf{r};t)$ exist: one relating to the movement of the beam, and the other relating to the beam shutdown during 0.5 s. Each phase is implemented as a separate function, and the active phase is determined by the current time, i.e., iteration number.

Figure 9.2 shows several snapshots of a typical dynamical evolution of the system. Imaginary-time propagation yields ground state of the system, which is shown in Figure 9.2(a), where we plot integrated 2D density profile in y-z plane. To model local inhomogeneities always present in the experiment, we add uniformly distributed random noise to the wave function of the order of 10%. Such modified ground state represents initial state of the system, and the beam starts to move along y direction at time $t = 0$ with the speed $v = 1.26$ mm/s. Figures 9.2(b) and 9.2(c) show 2D density profile of the system at times $t = 52.17$ ms and $t = 358.1$ ms, respectively. In Figure 9.2(b) the beam already reached its final position and its switch-off started. We can observe that several vortices (vortex dipoles) are generated and that the used speed exceeds the critical velocity. Generated vortices are stable and can be seen after extended period of time, Figure 9.2(c). To confirm that we indeed have vortices and not just localized density minima, we plot the $x = 0$ slice of the phase of the wave function in Figure 9.3. We observe characteristic braiding and jumps of the phase in the vicinity of density minima, which is a well-known hallmark of a vortex.

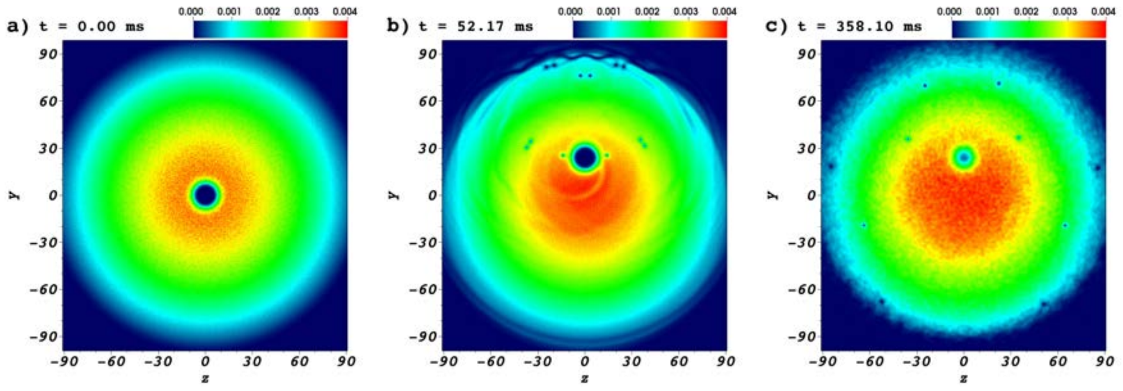


Figure 9.2: Time evolution of 2D density profile of BEC of sodium atoms with an obstacle moving at speed $v = 1.26$ mm/s $> v_c$, when several vortices are generated. Each panel shows integrated 2D density profile in y-z plane at different time t. All lengths are expressed in units of μm, and particle density is given in units N_{at}/l^2.

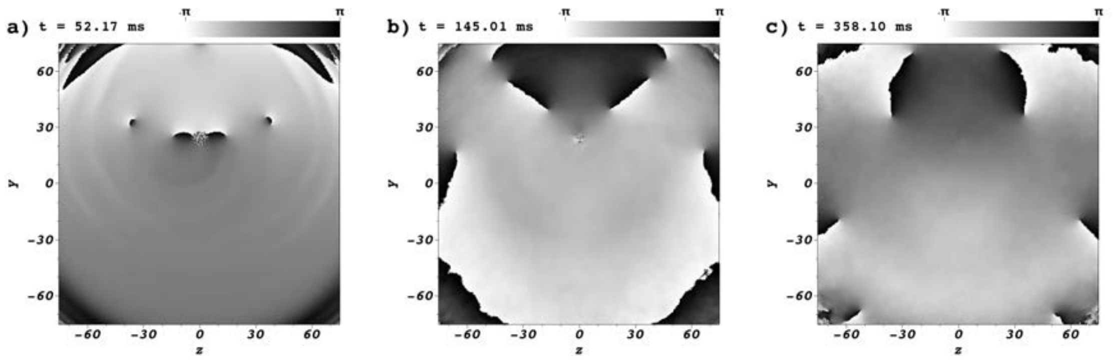


Figure 9.3: Time evolution of the wave function phase of BEC of sodium atoms with an obstacle moving at speed $v = 1.26$ mm/s $> v_c$, when several vortices are generated. Each panel shows $x = 0$ slice of the phase of the wave function in y-z plane at different time t. All lengths are expressed in units of μm.

When the obstacle is moving with an under-critical velocity, no vortices are generated, as in Figure 9.4. The velocity $v = 0.87$ mm/s is just slightly lower than the the critical one, and we can see precursors of vortices at the beam edge in Figures 9.4(a) and 9.4(b). At higher velocities, vortices emerge, as we have seen in Figure 9.2. At even higher velocities, Figure 9.5, rotating vortex pairs are generated, in addition to individual vortices.

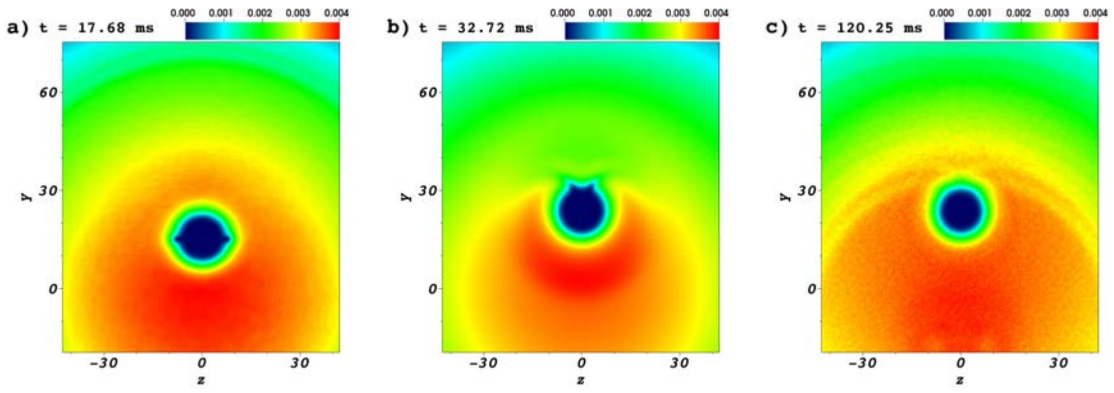


Figure 9.4: Time evolution of 2D density profile of BEC of sodium atoms with an obstacle moving at speed $v = 0.87$ mm/s $\lesssim v_c$, when no vortices are generated. Each panel shows integrated 2D density profile in y-z plane at different time t. All lengths are expressed in units of μm, and particle density is given in units N_{at}/l^2.

The obtained results are in good agreement with experimental findings of Ref. [97] and show that the programs developed within this thesis can be successfully used to model BEC systems with contact interaction, even for the most complex setup, when vortices are generated due to a moving obstacle.

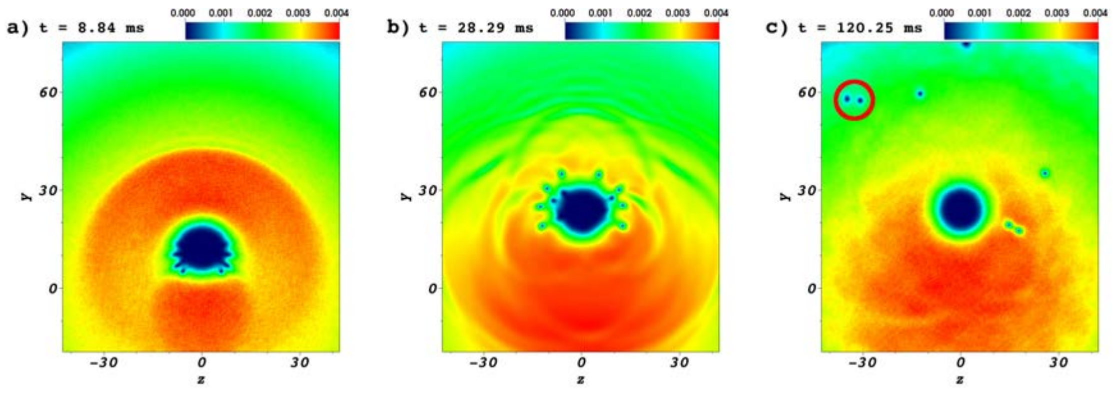


Figure 9.5: Time evolution of 2D density profile of BEC of sodium atoms with an obstacle moving at speed $v = 1.4$ mm/s $\gg v_c$, when rotating vortex pairs are also generated. Each panel shows integrated 2D density profile in y-z plane at different time t. A rotating vortex pair is highlighted in panel (c). All lengths are expressed in units of μm, and particle density is given in units N_{at}/l^2.

9.2 Effects of dipolar interactions on vortex formation in BEC

To further test our programs, we model vortex formation in a BEC of dysprosium atoms (^{164}Dy), which exhibit strong magnetic dipole moment. In particular, we study effects of the dipolar interaction strength for varying s-wave scattering lengths on the critical velocity for the emergence of vortices, which was not investigated experimentally. Dysprosium atoms have the largest magnetic dipole moment ($m = 10\mu_\text{B}$) available in ultracold atom experiments, corresponding the characteristic dipole-dipole interaction length $a_{\text{dd}} = 132\, a_0$. External magnetic field can be used to align all atomic dipoles in the same direction, as well as to tune their strength up to a maximal value given above. The same applies to the s-wave scattering length, which determines the contact interaction strength and which can also be tuned using the Feshbach resonance technique. Following Ref. [98], we take the same range of possible values for a_s as for a_{dd}, i.e., up to a maximal value of $132\, a_0$. With this we show how the developed programs can be used to model and theoretically address new physical phenomena, before they are studied experimentally. Such approach is essential for the design of many upcoming experiments, since otherwise it would be extremely difficult to predict what would be the relevant range of physical quantities to be measured, thus making it very challenging to perform the experiments. Having the results of detailed numerical simulations for the particular system enables experiments to target appropriate range of all relevant quantities, and to focus on discovering new phenomena.

The experiment follows the same methodology as in previous section and uses the same parameters where applicable. Namely, the trap frequencies remain the same, i.e., $(\omega_x, \omega_y, \omega_z) = 2\pi \times (9,\, 9,\, 400)$ Hz, as well as strength of the beam $V_0 = 250\, \hbar\tilde{\omega}$. Due to the strong dipolar interactions affecting the stability of the BEC, we had to use a much smaller number of atoms, $N_{\text{at}} = 8 \times 10^4$. This resulted in a much smaller BEC, so we had to reduce the $1/e^2$ beam waist to $\sigma = l$, where $l = 2.617\, \mu$m is harmonic oscillator length corresponding to dysprosium atoms. Also, we positioned the beam outside of the condensate, and move it all the way to the other side, as illustrated in Figure 9.6. The potential $V_\text{B}(\mathbf{r}; t)$ has the same form (9.1) as in previous section, just the center of the beam is now given by $y_0(t) = y_{00} + v_t$, where $y_{00} = -15$ in units of l.

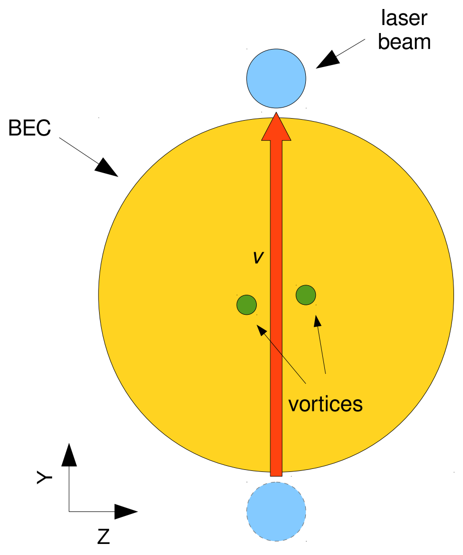


Figure 9.6: Illustration of the setup we use to study vortex formation in a dipolar BEC of dysprosium atoms. A repulsive Gaussian laser beam is initially outside of the condensate and moves along y direction with a constant velocity v until it reaches its final position on the other side. For sufficiently large velocity vortices are generated.

Modifications to the programs presented in the previous section can be reused for this numerical experiment and we only need to re-enable previously disabled calculation of the dipolar term in the main time propagation loop. As before, the ground state of the system is obtained using imaginary-time program, which then serves as the initial state of the real-time propagation program, with uniformly distributed random noise of 10% added. To calculate critical velocity v_c for the emergence of vortices, we search for the minimal speed of the laser beam for which two vortices appear, as illustrated in Figure 9.6.

Figure 9.7 shows several 2D density profiles of a typical dynamical evolution of the system for velocity $v = v_c = 0.16$ mm/s. The ground state, obtained through imaginary-time propagation is shown in Figure 9.7(a). The beam, initially outside of the condensate, moves along the y direction, as seen in Figures 9.7(b) through 9.7(f). In Figure 9.7(c) the precursors of the vortices form on the edges of the beam, which then separate from the beam if $v \geq v_c$, a situation seen in Figure 9.7(d). In Figures 9.7(e) and 9.7(f) we see that the generated vortices are stable, and survive for long propagation times.

Figure 9.8 shows results of our numerical study of the dipolar interaction effects on the critical velocity. We see that for large values of the s-wave scattering length, i.e., for the contact interaction comparable or larger than the dipole interaction, effects of decreasing $a_{\rm dd}$ are very small and probably could not be experimentally measured. On the other hand, when contact interaction is tuned down so that the dipole interaction starts to dominate the behavior of the system, critical velocity depends much stronger on $a_{\rm dd}$ and could be easily measured in future experiments.

The above study represents an example on how our programs can be used to verify and compare results of current experiments, as well as to theoretically investigate new phenomena and plan future experiments.

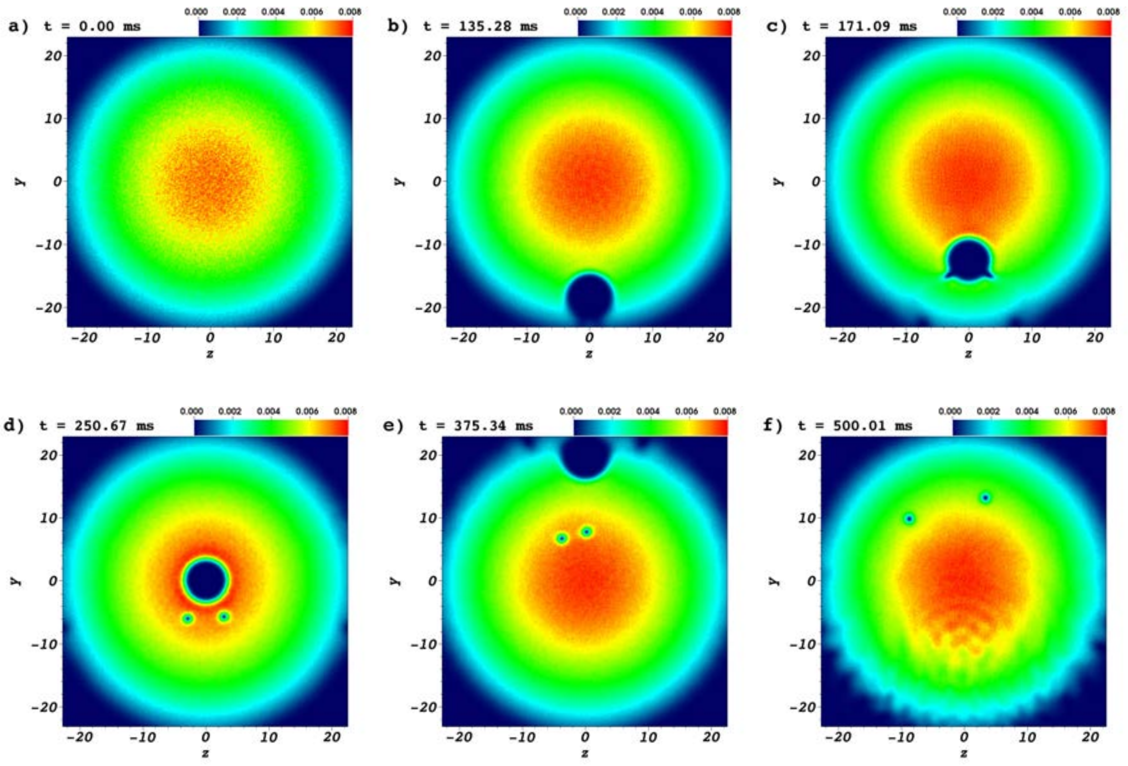


Figure 9.7: Time evolution of 2D density profile of BEC of dysprosium atoms for $a_s = 66a_0$ and $a_{\rm dd} = 44a_0$, with an obstacle moving at speed $v = v_c = 0.16$ mm/s, when two individual vortices are generated. Each panel shows integrated 2D density profile in y-z plane at different time t. All lengths are expressed in units of μm, and particle density is given in units $N_{\rm at}/l^2$.

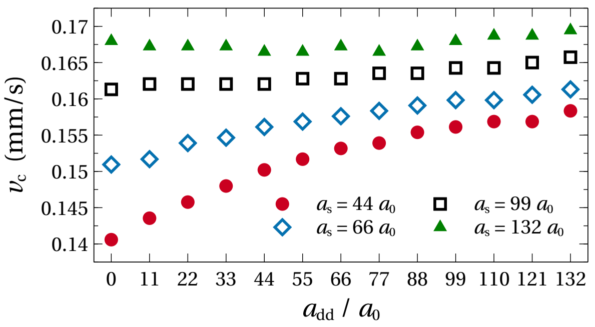


Figure 9.8: Critical velocity v_c for the emergence of vortices generated by a moving obstacle as a function of the characteristic dipolar interaction length $a_{\rm dd}$ (in units of a_0) for varying values of the s-wave scattering length a_s.

Chapter 10

Conclusions and future work

The main contribution of this thesis is the development of parallel algorithms for solving nonlinear differential equations of Gross-Pitaevskii type with a convolution integral kernel, as well as six implementations of the algorithms. The algorithms are based on the split-step, semi-implicit Crank-Nicolson method, while the convolution integral is solved using Fourier transform. Several parallelization approaches were considered, targeting both shared memory and distributed memory systems, with an emphasis on heterogeneous computing platforms. Algorithm for shared-memory architectures based on multi-core processing units is presented in Chapter 3 and its extension to specialized accelerator architecture in the form of GPUs is given in Chapter 4. Combination of the two approaches is given in Chapter 5, where we propose a hybrid algorithm targeting heterogeneous computer architectures, which fully harnesses the power of modern heterogeneous parallel resources of a single computer. While the (discrete) Fourier transform can be computed using the specialized FFT libraries that target specific hardware architecture, in a hybrid environment this is not easily achievable as no FFT library for heterogeneous architectures exists. Therefore, as part of the hybrid implementation, we have developed a heterogeneous algorithm for discrete Fourier transform, which combines the specialized multi-core and GPU FFT libraries. Our heterogeneous FFT algorithm remains general enough that it can easily be used outside the programs we developed, potentially simplifying the development of future heterogeneous algorithms which rely on Fourier transform.

The shared memory algorithms were further extended using MPI, allowing them to be executed in a distributed computing environment, e.g., on a computer cluster. This work is presented in Chapter 6. The benefit of distributed memory programs is twofold: they can be used to speed up the computation of simulations that can be executed on a single computer, and enable the simulations of higher resolution, which due to large memory requirements cannot fit in the memory of a single computer. Three implementations are given, targeting different computer cluster installations, from the ones based entirely on multi-core CPUs or GPUs, to the heterogeneous ones where the high-performing CPU and GPU are installed on the same computing node.

Two additional important contributions of this thesis are presented in Chapter 8. The first is the parameter optimization method for heterogeneous algorithms, which we developed to optimize the performance of our hybrid implementations. We have studied three approaches, based on brute-force search, gradient descent and genetic algorithm, and found the optimizer based on a genetic algorithm to be best suited to the problem. This method allows us to quickly find the optimal division of work between CPU host and GPU device, and we note that it can be adapted to work with other heterogeneous algorithms. The second contribution relates to the assessment of the efficiency of all developed programs through extensive performance evaluation. Testing results confirm that optimization of all programs described throughout the thesis yield highly efficient implementations. This chapter also includes detailed performance models for programs at different parallelization approaches. The developed methodology can be directly used in the evaluation of the expected performance of our programs on different computing resources.

The larger volume of data produced by the higher-resolution simulations could negatively impact user's ability to analyze the data and extract new information. We have addressed this by providing extensive integration with the VisIt visualization system, presented in Chapter 7. *In situ* visualization enables higher level of interaction with the simulation, through modification of the simulation's parameters while the simulation is running, visualization of its current state and control of its course. More traditional, *post-mortem* visualization is also possible, using standardized visualization file format. The usefulness of the developed integrations can be seen in Chapter 9, in which we demonstrate the usability of the developed programs on the example of study of vortex formation in BECs with a moving obstacle. Here, the visualization of high-resolution simulations was integral to the observations and measurements that were made.

In the future, we plan to address several important topics that include multi-component BEC systems, fast-rotating BECs, as well as other types of interaction, such as spin-orbit coupling. The main challenge for the development of multi-component programs is their increased memory requirement and therefore a distributed memory approach seems suitable to address this problem. Fast-rotating BECs are described by a Hamiltonian that includes additional, first-order partial derivatives w.r.t. spatial coordinates, which will require substantial changes in the implementation of Crank-Nicolson method in all approaches. Finally, different forms of the spin-orbit coupling, which represents a hot topic in the physics of ultracold atoms, will require combining of multi-component algorithm with the improved Crank-Nicolson method to allow study of novel physical systems that exhibit this type of interaction.

Bibliography

[1] S. Bose, “Plancks gesetz und lichtquantenhypothese,” *Zeitschrift für Physik*, vol. 26, pp. 178–181, 1924.

[2] A. Einstein, “Quantentheorie des einatomigen idealen gases,” *Sitzungsber. Kgl. Preuss. Akad. Wiss.*, vol. 1924, p. 261, 1924.

[3] M. H. Anderson, J. R. Ensher, M. R. Matthews, C. E. Wieman, and E. A. Cornell, “Observation of Bose–Einstein condensation in a dilute atomic vapor,” *Science*, vol. 269, pp. 198–201, 1995.

[4] C. C. Bradley, C. A. Sackett, J. J. Tollett, and R. G. Hulet, “Evidence of Bose–Einstein condensation in an atomic gas with attractive interactions,” *Phys. Rev. Lett.*, vol. 75, pp. 1687–1690, 1995.

[5] K. B. Davis, M. O. Mewes, M. R. Andrews, N. J. van Druten, D. S. Durfee, D. M. Kurn, and W. Ketterle, “Bose–Einstein condensation in a gas of sodium atoms,” *Phys. Rev. Lett.*, vol. 75, pp. 3969–3973, 1995.

[6] O. Morsch and M. Oberthaler, “Dynamics of Bose–Einstein condensates in optical lattices,” *Rev. Mod. Phys.*, vol. 78, pp. 179–215, 2006.

[7] I. Bloch, “Quantum coherence and entanglement with ultracold atoms in optical lattices,” *Nature*, vol. 453, pp. 1016–1022, 2008.

[8] R. P. Feynman, “Simulating Physics with Computers,” *Int. J. Theor. Phys.*, vol. 21, pp. 467–488, 1982.

[9] A. Griesmaier, J. Werner, S. Hensler, J. Stuhler, and T. Pfau, “Bose–Einstein condensation of chromium,” *Phys. Rev. Lett.*, vol. 94, p. 160401, 2005.

[10] L. P. Pitaevskii and S. Stringari, *Bose–Einstein Condensation.* Oxford University Press, 2003.

[11] C. J. Pethick and H. Smith, *Bose-Einstein Condensation in Dilute Gases.* Cambridge University Press, 2008.

[12] P. Muruganandam and S. Adhikari, “Fortran programs for the time-dependent Gross–Pitaevskii equation in a fully anisotropic trap,” *Comput. Phys. Commun.*, vol. 180, pp. 1888–1912, 2009.

[13] D. Vudragović, I. Vidanović, A. Balaž, P. Muruganandam, and S. K. Adhikari, “C programs for solving the time-dependent Gross–Pitaevskii equation in a fully anisotropic trap,” *Comput. Phys. Commun.*, vol. 183, pp. 2021–2025, 2012.

[14] B. Satarić, V. Slavnić, A. Belić, A. Balaž, P. Muruganandam, and S. K. Adhikari, "Hybrid OpenMP/MPI programs for solving the time-dependent Gross–Pitaevskii equation in a fully anisotropic trap," *Comput. Phys. Commun.*, vol. 200, pp. 411–417, 2016.

[15] L. E. Young-S., D. Vudragović, P. Muruganandam, S. K. Adhikari, and A. Balaž, "OpenMP Fortran and C programs for solving the time-dependent Gross–Pitaevskii equation in an anisotropic trap," *Comput. Phys. Commun.*, vol. 204, pp. 209–213, 2016.

[16] R. K. Kumar, L. E. Young-S., D. Vudragović, A. Balaž, P. Muruganandam, and S. Adhikari, "Fortran and C programs for the time-dependent dipolar Gross–Pitaevskii equation in an anisotropic trap," *Comput. Phys. Commun.*, vol. 195, pp. 117–128, 2015.

[17] V. Lončar, A. Balaž, A. Bogojević, S. Škrbić, P. Muruganandam, and S. K. Adhikari, "CUDA programs for solving the time-dependent dipolar Gross–Pitaevskii equation in an anisotropic trap," *Comput. Phys. Commun.*, vol. 200, pp. 406–410, 2016.

[18] V. Lončar, L. E. Young-S., S. Škrbić, P. Muruganandam, S. K. Adhikari, and A. Balaž, "OpenMP, OpenMP/MPI, and CUDA/MPI C programs for solving the time-dependent dipolar Gross–Pitaevskii equation," *Comput. Phys. Commun.*, vol. 209, pp. 190–196, 2016.

[19] P. Wittek and L. Calderaro, "Extended computational kernels in a massively parallel implementation of the Trotter–Suzuki approximation," *Comput. Phys. Commun.*, vol. 197, pp. 339–340, 2015.

[20] L. J. O'Riordan, T. Morgan, and N. Crowley, "GPUE: Phasegineering release." `https://github.com/mlxd/GPUE`, 2016. DOI:10.5281/zenodo.57968.

[21] R. Caplan, "NLSEmagic: Nonlinear Schrödinger equation multi-dimensional Matlab-based GPU-accelerated integrators using compact high-order schemes," *Comput. Phys. Commun.*, vol. 184, pp. 1250–1271, 2013.

[22] X. Antoine and R. Duboscq, "GPELab, a Matlab toolbox to solve Gross–Pitaevskii equations I: Computation of stationary solutions," *Comput. Phys. Commun.*, vol. 185, pp. 2969–2991, 2014.

[23] X. Antoine and R. Duboscq, "GPELab, a Matlab toolbox to solve Gross–Pitaevskii equations II: Dynamics and stochastic simulations," *Comput. Phys. Commun.*, vol. 193, pp. 95–117, 2015.

[24] M. Caliari and S. Rainer, "GSGPEs: A MATLAB code for computing the ground state of systems of Gross–Pitaevskii equations," *Comput. Phys. Commun.*, vol. 184, pp. 812–823, 2013.

[25] Ž. Marojević, E. Göklü, and C. Lämmerzahl, "ATUS-PRO: A FEM-based solver for the time-dependent and stationary Gross–Pitaevskii equation," *Comput. Phys. Commun.*, vol. 202, pp. 216–232, 2016.

[26] G. Vergez, I. Danaila, S. Auliac, and F. Hecht, "A finite-element toolbox for the stationary Gross–Pitaevskii equation with rotation," *Comput. Phys. Commun.*, vol. 209, pp. 144–162, 2016.

[27] M. Suzuki, "Decomposition formulas of exponential operators and lie exponentials with some applications to quantum mechanics and statistical physics," *J. Math. Phys.*, vol. 26, pp. 601–612, 1985.

[28] H. A. van der Vorst, "Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems," *SIAM J. Sci. Stat. Comp.*, vol. 13, pp. 631–644, 1992.

[29] P. Wittek, "Comparing three numerical solvers of the Gross–Pitaevskii equation." `http://peterwittek.com/gpe-comparison.html`, 2016. Accessed: 2017-04-10.

[30] J. Crank and P. Nicolson, "A practical method for numerical evaluation of solutions of partial differential equations of the heat-conduction type," *Math. Proc. Cambridge Philos. Soc.*, vol. 43, p. 50, 1947.

[31] W. F. Ames, *Numerical Methods for Partial Differential Equations.* Elsevier Science, 1992.

[32] R. Dautray and J.-L. Lions, *Mathematical Analysis and Numerical Methods for Science and Technology*, vol. 6, ch. XX, pp. 45–74. Springer, 1993.

[33] G. C. Wick, "Properties of Bethe–Salpeter wave functions," *Phys. Rev.*, vol. 96, pp. 1124–1134, 1954.

[34] G. Blelloch, "Scans as primitive parallel operations," *IEEE Trans. Comput.*, vol. 38, pp. 1526–1538, 1989.

[35] G. E. Blelloch, "Prefix sums and their applications," tech. rep., School of Computer Science, Carnegie Mellon University, 1990.

[36] D. Vudragović, I. Vidanović, A. Balaž, P. Muruganandam, and S. K. Adhikari, "GP-SCL package." `http://cpc.cs.qub.ac.uk/summaries/AEDU_v2_0.html`, 2012. Accessed: 2017-04-10.

[37] V. Lončar, L. E. Young-S., S. Škrbić, P. Muruganandam, S. K. Adhikari, and A. Balaž, "Repository of DBEC-GP family of programs." `https://git.ipb.ac.rs/vloncar/DBEC-GP`, 2016. Accessed: 2017-04-10.

[38] International Organization for Standardization (ISO), "The ANSI C standard (C99)," tech. rep., ISO/IEC, 1999.

[39] M. Frigo and S. G. Johnson, "FFTW – Fastest Fourier Transform in the West." `http://www.fftw.org/`, 2016. Accessed: 2017-04-10.

[40] R. L. Burden and J. D. Faires, *Numerical Analysis*, ch. 4, pp. 185–190. Brooks Cole Pub Co, 2010.

[41] S. Pop, A. Cohen, C. Bastoul, S. Girbal, G.-A. Silber, and N. Vasilache, "GRAPHITE: Polyhedral analyses and optimizations for GCC," in *Proceedings of the 2006 GCC Developers Summit*, p. 2006, 2006.

[42] UPC consortium, "Unified parallel C." `https://upc-lang.org/`, 2013. Accessed: 2017-04-10.

[43] D. Bonachea and G. Funck, "UPC language and library specifications, version 1.3," tech. rep., Lawrence Berkeley National Laboratory, 2013.

[44] Intel, "Intel Cilk Plus." `https://www.cilkplus.org/`, 2010. Accessed: 2017-04-10.

[45] OpenMP Architecture Review Board, *OpenMP Application Programming Interface*, 4.5 ed., 2015.

[46] V. Lončar, L. E. Young-S., S. Škrbić, P. Muruganandam, S. K. Adhikari, and A. Balaž, "DBEC-OMP-CUDA-MPI package." `https://data.mendeley.com/datasets/j3z9z379m8/2`, 2016.

[47] Nvidia Corporation, "CUDA home page." `http://www.nvidia.com/object/cuda_home_new.html`, 2007. Accessed: 2017-04-10.

[48] Khronos Group, "OpenCL home page." `https://www.khronos.org/opencl/`, 2009. Accessed: 2017-04-10.

[49] A. Danalis, G. Marin, C. McCurdy, J. S. Meredith, P. C. Roth, K. Spafford, V. Tipparaju, and J. S. Vetter, "The scalable heterogeneous computing (SHOC) benchmark suite," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, ACM Press, 2010.

[50] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE T. Comput.*, vol. C-21, pp. 948–960, 1972.

[51] D. Kirk and W. mei Hwu, *Programming Massively Parallel Processors.* Elsevier LTD, Oxford, 2016.

[52] J. Cheng, M. Grossman, and T. McKercher, *Professional CUDA C Programming.* John Wiley & Sons Inc, 2014.

[53] N. Wilt, *The CUDA Handbook.* Pearson Education, 2013.

[54] Nvidia Corporation, *cuFFT Library User's Guide*, 7.5 ed., 2017.

[55] M. Harris, "CUDA Pro Tip: Write Flexible Kernels with Grid-Stride Loops." `https://devblogs.nvidia.com/parallelforall/cuda-pro-tip-write-flexible-kernels-grid-stride-loops/`, 2013. Accessed: 2017-04-10.

[56] Stack Overflow discussion, "atomicAdd() for double on GPU." `http://stackoverflow.com/questions/16077464/`, 2013. Accessed: 2017-04-10.

[57] M. Harris, "Optimizing parallel reduction in CUDA." `http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf`, 2007. Accessed: 2017-04-10.

[58] Stack Overflow discussion, "How can I add up two 2d (pitched) arrays using nested for loops?." `http://stackoverflow.com/questions/6137218/`, 2011. Accessed: 2017-04-10.

[59] V. Lončar, A. Balaž, A. Bogojević, S. Škrbić, P. Muruganandam, and S. K. Adhikari, "DBEC-GP-CUDA package." `https://data.mendeley.com/datasets/s858p8zsgp/1`, 2016.

[60] J. Lee, M. Samadi, Y. Park, and S. Mahlke, "Transparent CPU–GPU collaboration for data-parallel kernels on heterogeneous systems," in *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, (Piscataway, NJ, USA), pp. 245–256, IEEE Press, 2013.

[61] M. Papadrakakis, G. Stavroulakis, and A. Karatarakis, "A new era in scientific computing: Domain decomposition methods in hybrid CPU–GPU architectures," *Comput. Methods Appl. Mech. Eng.*, vol. 200, pp. 1490–1508, 2011.

[62] Y. Li, J. R. Diamond, X. Wang, H. Lin, Y. Yang, and Z. Han, "Large-scale fast Fourier transform on a heterogeneous multi-core system," *Int. J. High Perform. Comput. Appl.*, vol. 26, pp. 148–158, 2012.

[63] Y. Ogata, T. Endo, N. Maruyama, and S. Matsuoka, "An efficient, model-based CPU-GPU heterogeneous FFT library," in *2008 IEEE International Symposium on Parallel and Distributed Processing*, IEEE, 2008.

[64] S. Chen and X. Li, "A hybrid GPU/CPU FFT library for large FFT problems," in *2013 IEEE 32nd International Performance Computing and Communications Conference (IPCCC)*, IEEE, 2013.

[65] E. Chu and A. George, *Inside the FFT Black Box.* CRC Press, 1999.

[66] Miroway, "Comparison between NVIDIA GeForce and Tesla GPUs." `https://www.microway.com/knowledge-center-articles/comparison-of-nvidia-geforce-gpus-and-nvidia-tesla-gpus/`, 2016. Accessed: 2017-04-10.

[67] Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard Version 3.1," tech. rep., University of Tennessee, Knoxville, Tennessee, 2015.

[68] O. Ayala and L.-P. Wang, "Parallel implementation and scalability analysis of 3D Fast Fourier Transform using 2D domain decomposition," *Parallel Comput.*, vol. 39, pp. 58–77, 2013.

[69] N. Li and S. Laizet, "2DECOMP&FFT – a highly scalable 2D decomposition library and FFT interface," in *Cray User Group 2010 conference, Edinburgh*, 2010.

[70] Nvidia Corporation, "MPI solutions for GPUs." `https://developer.nvidia.com/mpi-solutions-gpus`, 2017. Accessed: 2017-04-10.

[71] Open MPI project, "Open MPI home page." `https://www.open-mpi.org/`. Accessed: 2017-04-10.

[72] U. Ayachit, *The ParaView Guide: A Parallel Visualization Application.* Kitware, Incorporated, 2015.

[73] H. Childs, E. Brugger, B. Whitlock, J. Meredith, S. Ahern, D. Pugmire, K. Biagas, M. Miller, C. Harrison, G. H. Weber, H. Krishnan, T. Fogal, A. Sanderson, C. Garth, E. W. Bethel, D. Camp, O. Rübel, M. Durant, J. M. Favre, and P. Navrátil, "VisIt: An end-user tool for visualizing and analyzing very large data," in *High Performance Visualization–Enabling Extreme-Scale Scientific Insight*, pp. 357–372, CRC Press, 2012.

[74] Lawrence Livermore National Laboratory, *Silo User's Guide*, 2014. LLNL-SM-654357.

[75] W. Schroeder, K. Martin, and B. Lorensen, *Visualization Toolkit: An Object-Oriented Approach to 3D Graphics, 4th Edition.* Kitware, 2006.

[76] The Qt Company, "Qt application framework." `www.qt.io`. Accessed: 2017-04-10.

[77] J. Snyman, *Practical Mathematical Optimization.* Springer, 2005.

[78] C. Darken, J. Chang, and J. Moody, "Learning rate schedules for faster stochastic gradient search," in *Neural Networks for Signal Processing II Proceedings of the 1992 IEEE Workshop*, IEEE, 1992.

[79] T. Schaul, S. Zhang, and Y. LeCun, "No more pesky learning rates," in *Proceedings of the 30th International Conference on Machine Learning* (S. Dasgupta and D. McAllester, eds.), vol. 28 of *Proceedings of Machine Learning Research*, (Atlanta, Georgia, USA), pp. 343–351, PMLR, 2013.

[80] J. C. Spall, "Implementation of the simultaneous perturbation algorithm for stochastic optimization," *IEEE T. Aero. Elec. Sys.*, vol. 34, pp. 817–823, 1998.

[81] J. C. Spall, "Stochastic optimization and the simultaneous perturbation method," in *Proceedings of the 31st conference on Winter simulation Simulation—a bridge to the future - WSC 1999*, ACM Press, 1999.

[82] J. H. Holland, *Adaptation in Natural and Artificial Systems.* MIT University Press Group Ltd, 1992.

[83] M. Mitchell, *An Introduction to Genetic Algorithms.* MIT Press, 1998.

[84] M. Gen and R. Cheng, *Genetic Algorithms and Engineering Optimization.* John Wiley & Sons Inc, 1999.

[85] A. Kumar, "Encoding schemes in genetic algorithm," *Int. J. Adv. Res. Inf. Technol. Eng.*, vol. 2, pp. 1–7, 2013.

[86] L. J. Eshelman, R. A. Caruana, and J. D. Schaffer, "Biases in the crossover landscape," in *Proceedings of the Third International Conference on Genetic Algorithms*, (San Francisco, CA, USA), pp. 10–19, Morgan Kaufmann Publishers Inc., 1989.

[87] V. M. Spears and K. A. D. Jong, "On the virtues of parameterized uniform crossover," in *Proceedings of the Fourth International Conference on Genetic Algorithms*, pp. 230–236, 1991.

[88] J. N. Richter, *On mutation and crossover in the theory of evolutionary algorithms.* PhD thesis, Montana State University, 2010.

[89] D. R. White and S. Poulding, "A rigorous evaluation of crossover and mutation in genetic programming," in *Lecture Notes in Computer Science*, vol. 5481, pp. 220–231, Springer Berlin Heidelberg, 2009.

[90] A. P. A. da Silva and D. M. Falcao, “Fundamentals of genetic algorithms,” in *Modern Heuristic Optimization Techniques: Theory and Applications to Power Systems* (K. Y. Lee and M. A. El-Sharkawi, eds.), ch. 2, pp. 25–42, John Wiley & Sons Inc, 2008.

[91] S. Luke and L. Spector, “A comparison of crossover and mutation in genetic programming,” in *Genetic Programming 1997: Proceedings of the Second Annual Conference* (J. R. e. a. Koza, ed.), pp. 240–248, Morgan Kaufmann, 1997.

[92] S. Luke and L. Spector, “A revised comparison of crossover and mutation in genetic programming,” in *Genetic Programming 1998: Proceedings of the Third Annual Conference* (J. R. e. a. Koza, ed.), pp. 208–213, Morgan Kaufmann, 1998.

[93] G. M. Amdahl, “Validity of the single processor approach to achieving large scale computing capabilities,” in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference - AFIPS ’67 (Spring)*, ACM Press, 1967.

[94] Open MPI project, “FAQ: Running CUDA-aware Open MPI.” `https://www.open-mpi.org/faq/?category=runcuda`, 2016. Accessed: 2017-04-10.

[95] M. Frigo and S. G. Johnson, “FFTW 3 documentation, version 3.3.6.” `http://www.fftw.org/fftw3_doc/`, 2017. Accessed: 2017-04-10.

[96] See reference [54], Section 2.12.

[97] W. J. Kwon, G. Moon, S. W. Seo, and Y. Shin, “Critical velocity for vortex shedding in a Bose–Einstein condensate,” *Phys. Rev. A*, vol. 91, p. 053615, 2015.

[98] H. Kadau, M. Schmitt, M. Wenzel, C. Wink, T. Maier, I. Ferrier-Barbut, and T. Pfau, “Observing the Rosensweig instability of a quantum ferrofluid,” *Nature*, vol. 530, pp. 194–197, 2016.

Prošireni sažetak

Algoritmi za rešavanje parcijalnih diferencijalnih jednačina, posebno paralelni numerički algoritmi, danas predstavljaju veoma značajnu oblast istraživanja, posebno uzimajući u obzir da su praktično svi današnji računari zasnovani na sistemima sa više procesorskih jedinica. Višejezgarni procesori predstavljaju osnovni nivo paralelizma koji postoji u svim modernim računarima, a grafički akceleratori (engl. Graphics Processing Unit, GPU) i druge vrste koprocesora (npr. Intel Xeon Phi kartice) su sledeći nivo u ovoj hijerarhiji. Računarski klasteri, koji se sastoje od velikog broja računara povezanih brzom mrežnom infrastrukturom, uz odgovarajuća softverska rešenja koja omogućavaju da se ceo sistem koristi na paralelan način (npr. pomoću Message Passing Interface, MPI paradigme), kompletiraju ovu veoma heterogenu hijerarhiju. Efikasno korišćenje velikih računarskih sistema zahteva domen-specifična znanja i razvoj paralelnih algoritama koji uzimaju u obzir hardverske osobine sistema, kao i mogućnosti paralelizacije numeričkih metoda koji se koriste za rešavanje datog problema.

Jedna od oblasti gde su ovakvi algoritmi od velikog značaja je fizika ultrahladnih atoma, koja proučava osobine materije na veoma niskim temperaturama, reda nanokelvina. Na tim temperaturama gasovi nekih elemenata ili jedinjenja doživljavaju fazni prelaz pod nazivom Boze-Ajnštajn kondenzacija i prelaze u novo stanje, koje se odlikuje koherentnim kvantnim kolektivnim ponašanjem. Ovakvi sistemi su od velikog interesa jer pružaju ranije nezamislivu kontrolu nad eksperimentalnim parametrima i omogućavaju primene u kvantnom računarstvu, kao i simuliranje drugih kvantnih sistema (Fajnmanov kvantni simulator), opisanih sličnim jednačinama. U poslednjoj deceniji su posebno proučavani sistemi u kojima se, pored sveprisutne kontaktne interakcije, pojavljuje i dipol-dipol interakcija, a eksperimentalna realizacija Boze-Ajnštajn kondenzacije sa atomima velikog dipolnog momenta (erbijum, disprozijum) u poslednjih par godina je otvorila novu istraživačku oblast u kojoj se proučavaju dominantni efekti anizotropne dipolne interakcije.

Osnovni opis ovakvih fizičkih sistema daje teorija srednjeg polja. Dok za uobičajene kvantne sisteme možemo da koristimo linearnu parcijalnu diferencijalnu Šredingerovu jednačinu, u slučaju Boze-Ajnštajn-kondenzovanih sistema teorija srednjeg polja daje nelinearnu Šredingerovu jednačinu, odnosno Gros-Pitaevski jednačinu (GPJ). U prisustvu dipol-dipol interakcije ona sadrži nelinearni integralni kernel konvolucionog tipa, i rešavanje ovakvih jednačina zahteva primenu jednog ili više od nekoliko poznatih numeričkih metoda. U poglavlju 1 dat je kratak uvod u oblast, kao i formulacija osnovnih problema.

Ova teza se bavi razvojem i implementacijom paralelnih algoritama za rešavanje Gros-Pitaevski jednačine sa dipolnom interakcijom, bazirane na CUDA (na GPU) i OpenMP paralelizaciji (na višejezgarnim procesorima) u okviru jednog računara, MPI paralelizaciji na nivou računarskog klastera, kao i kombinacije ovih pristupa. Razvijeni algoritmi omogućavaju ne samo rešavanje vremenski zavisne GPJ, nego i vremenski nezavisne jednačine, prelaskom na propagaciju u imaginarnom vremenu. Ovakvi algoritmi omogućavaju efikasno rešavanje GPJ u velikom broju primena u fizici ultrahladnih atoma, nelinearnoj optici, primenjenoj fizici, astrofizici, kao i u kvantitativnim finansijama, u kojima se takođe koriste jednačine sličnog tipa. Tokom izrade disertacije su razvijena softverska rešenja u programskom jeziku C i detaljno izmerena poboljšanja koja ona donose u odnosu na druge pristupe. Takođe je razmotren problem *in situ* vizuelizacije podataka dobijenih tokom izvršavanja numeričkih simulacija i rešavanja vremenski zavisne i vremenski nezavisne GPJ. Sve implementacije opisane u ovoj tezi su otvorenog koda, i dostupne su za preuzimanje [37].

Za rešavanje GPJ i jednačina sličnog tipa koristi se veliki broj metoda, a kratak pregled relevantne literature je dat u poglavlju 2. Analitički se koriste različite aproksimativne metode, kao što je Tomas-Fermi aproksimacija, koja zanemaruje kinetičku energiju sistema, odnosno Laplasijan u GPJ. Koriste se i varijacioni metodi, kod kojih se za rešavanje GPJ pretpostavi funkcionalni oblik talasne funkcije, a onda se na osnovu ekstremizacionog uslova nalazi optimalno rešenje. Oba ova pristupa su popularna jer omogućavaju analitički tretman problema, ali, sa druge strane, ne daju mogućnost kontrole niti ocene greške takve aproksimacije.

Perturbativni razvoj (na primer, aproksimacija Bogoljubova, Hartri-Fok-Bogoljubova, Hartri-Fok-Bogoljubov-Popova) predstavlja sledeći nivo u analitičkom tretmanu ovakvih sistema i pruža mogućnost ocene greške, ali uz izuzetnu matematičku kompleksnost rešenja, koja u mnogim slučajevima praktično onemogućava korišćenje ovog pristupa. Postoje i malobrojni neperturbativni metodi, ali je njihova primenljivost veoma sužena na mali broj konkretnih problema (npr. solitonska rešenja u jednodimenzionalnom slučaju).

Pored analitičkih metoda, veoma značajnu ulogu imaju i numerički metodi za rešavanje GPJ, koji u velikom broju slučajeva predstavljaju jedinu mogućnost za proučavanje datih fizičkih sistema. U ove pristupe spadaju spektralne metode, metode podeljenog koraka, kao i metode konačne razlike elemenata. U slučaju GPJ sa dipolnom interakcijom, konvolucioni integral se obično rešava višestrukim korišćenjem diskretne Furijeove transformacije. Svaki od ovih metoda ima prednosti i mane kada se razmatraju njihova stabilnost i preciznost, a jedan od pristupa koji garantuje apsolutnu stabilnost i odlično skaliranje grešaka diskretizacije je semi-implicitni Krenk-Nikolson metod podeljenog koraka (KN). Ranije je slučaj kontaktne interakcije razvijen serijski algoritam koji implementira ovaj metod [12], kao i njegova paralelizovana verzija pomoću OpenMP pristupa [13]. Za slučaj GPJ sa dipol-dipol interakcijom ranije je razvijen samo serijski algoritam [16]. Ovo predstavlja odlično rešenje za jednodimenzionalne probleme, dok u dve i tri prostorne dimenzije vreme izvršavanja simulacija za sisteme u razumnoj diskretizaciji veoma brzo postaje suviše veliko da bi ovi algoritmi mogli praktično da se koriste. Zbog toga je jedan od otvorenih problema razvoj efikasnih paralelnih algoritama na heterogenim platformama koji će omogućiti brzo rešavanje GPJ za sisteme sa finom diskretizacijom u dve i tri prostorne dimenzije. Pošto je izračunavanje konvolucionog interakcionog integrala bazirano na korišćenju diskretne Furijeove transformacije, od velike koristi je upotreba GPU kartica, koje mogu značajno da ubrzaju izvršavanje ovakvih algoritama. Takođe, u trodimenzionalnim slučajevima velika količina podataka koja je neophodna za opis sistema često zahteva distribuiranje izračunavanja na više računara, odnosno primenu MPI paralelizacije. Svi ovi problemi su predmet aktivnog istraživanja u ovoj oblasti.

U poglavlju 3 predstavljen je algoritam za rešavanje GPJ sa dipolnim članom namenjen sistemima sa deljenom memorijom. Algoritam se oslanja na KN metod podeljenog koraka i omogućava efikasne implementacije na sistemima gde je celokupan sadržaj memorije vidljiv svim računskim resursima. Opis algoritma započinjemo definicijom diskretizacione šeme, koja definiše način na koji su prostor i vreme diskretizovani pomoću konačnog broja tačaka. Razlikujemo tri slučaja, u zavisnosti od broja prostornih dimenzija problema koji rešavamo. Srž algoritma je petlja u kojoj jedna iteracija odgovara jednom vremenskom koraku propagacije. Unutar petlje, vrednosti talasne funkcije se računaju u odnosu na različite delove Hamiltonijana, u skladu sa šemom podeljenog koraka opisanom u poglavlju 2. Svaki korak glavne petlje podeljen je na više manjih koraka. U prvom od njih se računa vrednost dipolnog člana GPJ i na egzaktan način propagira vrednost talasne funkcije u odnosu na članove Hamiltonijana koji ne sadrže prostorne izvode, dok se u preostalim koracima talasna funkcija propagira u odnosu na delove Hamiltonijana sa prostornim izvodima pomoću KN metoda. Vrednost dipolnog člana se računa korišćenjem diskretne Furijeove transformacije. Prilikom propagacije u imaginarnom vremenu, neophodan je dodatni korak normalizacije talasne funkcije u svakoj iteraciji glavne petlje.

S obzirom na to da se u svakoj iteraciji više puta računa vrednost talasne funkcije u svim tačkama diskretizacione šeme, lako se može zaključiti da je algoritam računski izuzetno zahtevan. Ovo je dodatno izraženo u dve i tri dimenzije, gde ukupan broj diskretizacionih tačaka brzo raste povećanjem rezolucije, te se javlja potreba za paralelizacijom.

Osim algoritma, u poglavlju 3 su opisane i serijska i paralelna implementacija. Serijska implementacija je napisana u programskom jeziku C, sadrži 10 programa, i oslanja se na prethodne implementacije u programskim jezicima Fortran i C [12, 13, 16]. Implementirani su zasebni programi za rešavanje GPJ u jednoj, dve i tri dimenzije, u realnom i imaginarnom vremenu. Za računanje vrednosti dipolnog člana GPJ korišćena je FFTW biblioteka. Radi preglednosti, koraci u glavnoj petlji su implementirani u zasebnim funkcijama, što olakšava buduće izmene programa u skladu sa potrebama eksperimenata i drugih primena. Takođe, da bi se smanjila količina numeričkih operacija u svakoj iteraciji, vrednosti koje se ne menjaju (potencijal zamke i dipol-dipol interakcije, koeficijenti KN šeme, itd.) se izračunavaju samo jednom i smeštaju u memoriju, a zatim se po potrebi koriste tokom propagacije vrednosti talasne funkcije.

Paralelna implementacija algoritma za sisteme sa deljenom memorijom je zasnovana na OpenMP tehnologiji. Ova tehnologija omogućava paralelizaciju korišćenjem niti koje se mogu izvršavati paralelno na višejezgarnim procesorima. Nitima se ne manipuliše direktno, već se putem instrukcija u programskom kodu označavaju regioni koji će biti paralelizovani, na osnovu kojih kompajler generiše paralelni kod. Paralelizacija petlji u kojima pojedinačne iteracije nemaju međusobnih zavisnosti na ovaj način postaje veoma jednostavna. Međutim, u slučaju da se u petlji javljaju rekurzivne relacije, kao što je slučaj u funkcijama koje implementiraju KN šemu, paralelizacija je moguća jedino uz obimnu izmenu algoritma, od koje smo zbog kompleksnosti odustali. Ovo znači da u jednoj dimenziji nije moguće ostvariti paralelizaciju svakog koraka u iteraciji, što nije velik problem imajući u vidu da broj tačaka u diskretizacionoj šemi nije velik u ovim programima. U programima koji rešavaju GPJ u dve ili tri dimenzije problem rekurzivnih relacija efektivno nestaje, jer se rekurzije javljaju samo u unutrašnjoj petlji što ostavlja mogućnost da se paralelizuje spoljašnja petlja. U ovom pristupu neophodno je jedino obezbediti zaseban prostor u memoriji za privremene promenljive koje svaka nit koristi. Posledica ovakvog pristupa je veće zauzeće memorije zbog privremenih promenljivih, međutim ukupno povećanje nije veliko, a svakako je opravdano značajno boljim performansama koje ova implementacija ima u odnosu na serijsku, što je pokazano u poglavlju 8. Osim petlji u funkcijama, neophodno je paralelizovati i računanje vrednosti dipolnog člana GPJ, za koji se koristi FFTW biblioteka. Ova biblioteka već podržava OpenMP, čime je paralelizacija ovog koraka značajno olakšana.

Algoritam za sisteme sa deljenom memorijom se može prilagoditi i implementirati na grafičkim procesorima (GPU), što je tema poglavlja 4. Iako algoritam nije značajno izmenjen, zbog različitog programskog modela koji GPU koristi implementacija je značajno kompleksnija. Naša implemetacija za grafičke procesore je zasnovana na CUDA tehnologiji, koju koriste Nvidia grafički procesori. GPU arhitektura se fundamentalno razlikuje od tradicionalne arhitekture koje poseduju centralni procesori računara, i njeno razumevanje je neophodno da bi se iskoristio pun potencijal grafičkih procesora. Dok su centralni procesori dizajnirani da izvršavaju raznorodne zadatke korišćenjem relativno malog broja jezgara, grafički procesori su dizajnirani isključivo za *floating-point* operacije, za šta koriste velik broj GPU niti. Niti se izvršavaju po modelu *Single Instruction Multiple Thread*, gde sve izvršavaju istu instrukciju nad različitim podacima, uz značajno manju kontrolu nad međusobnom sinhronizacijom. Grafički procesori poseduju svoju RAM memoriju, odvojenu od glavne memorije računara, i svi podaci kojima GPU efikasno pristupa moraju biti dostupni u njegovoj memoriji. Iako novi grafički procesori imaju i do 24 GB memorije, to je ipak osetno manje od glavne memorije najmodernijih računara, koji često imaju i po 128 GB. Iz tog razloga, optimalno korišćenje GPU memorije je ključno za njihovu efikasnu primenu.

CUDA grafički procesori se programiraju pomoću ekstenzije programskog jezika C, nazvanog CUDA C. CUDA pravi razliku između koda koji se izvršava na glavnom procesoru i koji je pisan u neizmenjenom programskom jeziku C, i koda koji se izvršava na grafičkom procesoru za koji je neophodno korišćenje proširenog jezika C i specijalizovanog kompajlera. Centralno mesto u CUDA programskom modelu zauzima koncept *kernel*-a, odnosno funkcija koje programer definiše, a koje se izvršavaju na grafičkom procesoru. Kerneli se pozivaju korišćenjem specijalne sintakse, i prilikom poziva CUDA kreira željeni broj niti koje izvršavaju telo funkcije. Unutar kernela je moguće napraviti razliku između pojedinačnih niti korišćenjem posebnih varijabli preko kojih se može izračunati njihov broj ili mesto u hijerarhiji izvršavanja.

Grafički procesori se, iako međusobno imaju sličnu hardversku arhitekturu, značajno razlikuju po svojim računskim kapacitetima. Da bi se omogućio razvoj programa koji ne zavise od računskog kapaciteta grafičkog procesora, CUDA model izvršavanja se oslanja na logičku hijerarhiju na dva nivoa. Na prvom nivou se nalazi skup svih niti koje će se izvršiti, nazvan *grid*, koji je podeljen u niz blokova iste veličine, dok drugi nivo čine niti unutar blokova. Veličina grida i blokova se definiše pri pozivu kernela. Prilikom izvršenja programa, CUDA određuje koliko blokova će se paralelno izvršavati, u zavisnosti od kapaciteta grafičkog procesora. Bitna posledica ovakvog modela je da ne postoji garancija o broju blokova koji se izvršavaju ili njihovom redosledu, što znači da sinhronizacija između njih nije moguća. Sinhronizacija niti je moguća jedino unutar jednog bloka. Ovo u značajnoj meri utiče na dizajn algoritama za grafičke procesore, koji se često moraju značajnije menjati da bi mogli da se efikasno implementiraju.

Implementacija našeg algoritma na GPU je zahtevala značajne izmene u odnosu na OpenMP paralelizaciju. S obzirom na to da se implementacija algoritma u značajnoj meri oslanja na korišćenje petlji bez međuzavisnosti njihovih iteracija, implementacija nije zahtevala veće izmene algoritma i upotrebu sinhronizacije. Jedino je pomoćni algoritam za numeričku integraciju, baziran na Simpsonovom pravilu, morao da se izmeni u značajnijoj meri. S druge strane, implementacija predstavlja veliki tehnički izazov jer zahteva korišćenje CUDA tipova podataka za reprezentaciju kompleksnih brojeva, specifičan način (de)alokacije memorije i pristup indeksima nizova, aritmetiku sa pokazivačima, izmenu biblioteke za Furijeovu transformaciju, itd. Takođe, posebna pažnja je bila posvećena optimizaciji upotrebe memorije, odnosno smanjenju ukupne memorije koju program zauzima na grafičkom procesoru, što je postignuto reorganizacijom računskih operacija i naprednom upotrebom pokazivača. Na ovaj način, u CUDA implementaciji smo uspeli da smanjimo upotrebu memorije i do 50% u odnosu na OpenMP implementaciju.

Moderni računarski sistemi često poseduju snažan višejezgarni centralni procesor povezan sa grafičkom karticom sličnih računskih performansi. Korišćenjem programa iz poglavlja 3 ili 4 se može koristiti samo jedan od ovih resursa. Bolji pristup je razvoj hibridnog algoritma i implementacije koji će objediniti oba resursa u jedan heterogeni sistem, što se razmatra u poglavlju 5. U ovom poglavlju je opisan hibridni algoritam, od načina podele podataka između dva resursa do podele obrade podataka na svakom od njih, a takođe je predstavljena implementacija i optimizacija dobijenog rešenja.

Prvi korak u dizajnu hibridnog algoritma je razvoj efikasne šeme podele podataka i posla između dva resursa. Imajući u vidu da su memorije centralnog i grafičkog procesora odvojene i različitog su kapaciteta, nije opravdano šemu bazirati na modelu koji zahteva da sadržaj obe memorije bude istovetan. Umesto toga, potrebno je logički podeliti podatke na deo koji će obrađivati grafički procesor i deo koji će obrađivati centralni procesor. U slučaju da je jednom resursu potreban deo podataka na kojima radi drugi resurs, ove podatke je potrebno preraspodeliti kopiranjem u odgovarajuću memoriju. Pritom treba omogućiti da se simultano računanje na oba resursa nesmetano nastavi, kao i da ne dođe do problema sa sinhronizacijom. Upravo takvu šemu koristi naš hibridni algoritam i ona predstavlja važan doprinos ove teze. Korisnost šeme je takođe vidljiva kroz hibridni algoritam za Furijeovu transformaciju, koji je razvijen za potrebe programa, ali se može lako iskoristiti i za druge hibridne algoritme koji zahtevaju ovu transformaciju. Ovaj algoritam je primenljiv na višedimenzionu Furijeovu transformaciju i funkcioniše tako što deli transformaciju na dva dela: deo koji se može izračunati na svakom resursu sa podacima koji se nalaze u lokalnoj memoriji, i deo za koji je potrebna distribucija podataka po dimenziji koja nije lokalna.

Hibridna implementacija osnovnog algoritma je zasnovana na delovima koda OpenMP i CUDA implementacija. Glavne funkcije za računanje vremenske propagacije talasne funkcije se mogu iskoristiti uz male izmene granica petlji. Međutim, ovako naivna implementacija bi bila veoma neefikasna, jer bi oba resursa bila neiskorišćena tokom preraspodele podataka. Implementacija se može unaprediti tako što se računske operacije nad jednim delom podataka podele u manje celine i povežu u jedan tok (engl. *stream*) koji ima definisan redosled izvršavanja. Upotrebom više tokova moguće je u potpunosti preklopiti raspodelu podataka i računanje nad njima, i time značajno povećati efikasnost hibridne implementacije. U ovom slučaju nastaje problem da za maksimalno iskorišćenje oba resursa moramo precizno da odredimo količinu podataka koju će svaki procesor obraditi, kao i broj tokova koji će biti korišćen, što nije jednostavan zadatak jer umnogome zavisi od performansi oba resursa. Rešenje ovog problema je predstavljeno u poglavlju 8, u formi metoda za automatsku optimizaciju parametara podele podataka.

Algoritmi predstavljeni u poglavljima 3, 4 i 5 namenjeni su izvršavanju na jednom računaru, gde mogu da koriste sve dostupne resurse. Sledeći korak je njihovo proširenje na sisteme sa distribuiranom memorijom, odnosno na računarske klastere, što je tema poglavlja 6. Karakteristika ovakvih sistema je model memorije u kome nijedan računarski resurs ne poseduje kompletan skup podataka. Pristup podacima koji pripadaju nekom drugom resursu zahteva njihov prenos kroz mrežni interfejs, što je značajno sporije od rada sa lokalnim podacima. Algoritmi za ovakve sisteme treba da poseduju šemu distribucije podataka koja maksimalno iskorišćava lokalnost podataka, a podacima udaljenih resursa pristupa samo kada je to neophodno. U okviru poglavlja 6 predstavljeno je proširenje glavnog algoritma na sisteme sa distribuiranom memorijom, kao i tri implementacije. Algoritam koristi model razmene poruka da pristupi podacima na drugim računarima u klasteru, dok se za implementaciju koristi *Message Passing Interface* (MPI). Sve tri implementacije koriste MPI samo za razmenu podataka, dok se za paralelno računanje nad lokalnim podacima koriste algoritmi koje smo prethodno razvili. Tri implementacije su:

1. Implementacija koja radi samo sa centralnim procesorima, zasnovana na algoritmu za deljenu memoriju, odnosno njegovoj OpenMP paralelizaciji iz poglavlja 3 (OpenMP/MPI),

2. Implementacija koja radi samo sa grafičkim procesorima, a zasnovana je na implementaciji iz poglavlja 4 (CUDA/MPI),

3. Hibridna implementacija zasnovana na algoritmu predstavljenom u poglavlju 5 (Hybrid/MPI).

Sve implementacije koriste istu šemu distribucije podataka i imaju sličnu strukturu. Prilikom razvoja, fokus je stavljen na programe koji rešavaju GPJ u tri dimenzije, jer je potreba za njihovom paralelizacjiom najveća. Korišćenje ovih implementacija na sistemima sa distribuiranom memorijom pomaže nam da ostvarimo dva cilja, tj. omogućava nam da upotrebom više računarskih resursa skratimo vreme izvršavanja simulacija za datu rezoluciju, dok nam takođe omogućavaju i da izvršavamo simulacije sa značajno većom rezolucijom, za koje kapacitet memorije jednog računara nije dovoljan i koje na drugi način ne bi ni mogle da budu izvršene.

Predstavljeni algoritam za distribuirane sisteme se oslanja na podelu podataka po jednoj dimenziji, što znači da svaki resurs (ili proces u MPI terminologiji) ima u svojoj lokalnoj memoriji podatke u preostale dve dimenzije. Inicijalno su podaci podeljeni između procesa po x koordinati, što omogućava da svaki proces u svojoj lokalnoj memoriji propagira talasnu funkciju u odnosu na delove Hamiltonijana koji sadrže prostorne izvode po y i z koordinati. Da bi talasnu funkciju propagirali u odnosu na deo Hamiltonijana sa prostornim izvodima po x koordinati neophodno je transponovati podatke, tako da se obezbedi lokalnost podataka po celokupnoj x koordinati u svakom procesu. Operacija transponovanja podataka zauzima centralno mesto u ovim programima i njihove performanse su direktno povezane. U tezi smo ispitivali dva načina transponovanja podataka: korišćenjem FFTW biblioteke koja ovu operaciju pruža kao svoju dodatnu funkcionalnost, i putem rutine koju smo sami razvili za ovu potrebu. Uzimajući u obzir da OpenMP i hibridna implementacija već koriste FFTW prilikom računanja vrednosti dipolnog clana GPJ, upotreba ove biblioteke za transponovanje podataka ne unosi dodatne zavisnosti u program, odnosno jedini preduslov je da je FFTW biblioteka kompajlirana sa podrškom za MPI. Međutim, CUDA implementacija se ne oslanja na ovu biblioteku, već koristi cuFFT biblioteku koja radi isključivo sa grafičkim procesorima, i koja ne poseduje ekvivalentnu operaciju za transponovanje, niti bilo kakvu podršku za MPI. Da bi rešili ovaj problem, razvili smo novu rutinu za ovu operaciju, koja koristi MPI tipove podataka i komunikaciju između svih procesa da bi ostvarila neophodnu razmenu podataka. Uz upotrebu MPI implementacije koja pristupa memoriji grafičkog procesora direktno, bez potrebe za prebacivanjem u glavnu memoriju, ova rutina ima odlične performanse. Takođe, pošto dva navedena pristupa operaciji transponovanja kao rezultat imaju identičan raspored podataka u memoriji, u Hybrid/MPI implementaciji su oba pristupa omogućena i korisnik prilikom kompajliranja programa može da odabere koju implementaciju želi.

Osim operacije transponovanja koja nam je neophodna da bi izvršili propagaciju talasne funkcije u odnosu na odgovarajući deo Hamiltonijana, potreban nam je i algoritam za računanje vrednosti dipolnog člana GPJ. Da bi se njegova vrednost izračunala, neophodno je koristiti algoritam za diskretnu Furijeovu transformaciju koji radi na sistemima sa distribuiranom memorijom. U slučaju OpenMP/MPI implementacije, ovo je jednostavan zadatak, jer FFTW biblioteka ima podršku za MPI i koristi identičan model podele podataka između procesa. U Hybrid/MPI programima bilo je neophodno proširiti algoritam za Furijeovu transformaciju iz poglavlja 5. Osnovni koncept podele po dimenzijama i korišćenja tokova podataka ostaje nepromenjen, s tim što je u ovom slučaju neophodno i dodatno transponovati podatke i obezbediti odgovarajući raspored podataka u memoriji. Kao i u slučaju hibridnog algoritma za Furijeovu transformaciju na jednom računaru, i ovaj algoritam predstavlja dodatni doprinos teze koji se može primeniti izvan okvira programa za koje je razvijen. U slučaju CUDA/MPI implementacije može se primeniti ista ideja podele podataka, dodatno olakšana činjenicom da su svi podaci jednog procesa u memoriji grafičkog procesora, te nije neophodno korišćenje tokova podataka i njihovo prebacivanje u glavnu memoriju računara.

Korišćenje MPI programa omogućava simulacije eksperimenata značajno veće rezolucije, te se javlja problem ispisa vrednosti koje program računa. Sa jedne strane, ukoliko bi MPI procesi unutar programa ispisivali svoje lokalne podatke u zasebne fajlove, korisniku bi ostao nezavidan zadatak da te fajlove objedini na korektan način i analizira. Sa druge strane, tekstualni zapis koji programi iz poglavlja 3, 4 i 5 koriste postaje neadekvatan za simulacije visoke rezolucije usled ogromne veličine fajlova. MPI programi koje smo razvili ovaj problem rešavaju korišćenjem MPI *input/output* operacija. Umesto u zasebne fajlove, svi procesori pišu u jedan, binarni fajl koji zadržava preciznost podataka iz programa, i pritom zauzima manje prostora na disku od tekstualnog zapisa.

Sa povećanim obimom izlaznih podataka, javio se i problem njihove analize i vizuelizacije, kako nakon završetka izvršavanja simulacije, tako i tokom samog izvršavanja. Ovim problemom se bavi poglavlje 7. Vizuelizacija podataka tokom izvršavanja simulacije, tzv. *in situ* vizuelizacija, je veoma korisna jer omogućava značajno bržu analizu rezultata, a ukoliko dodatno pruža mogućnost uticaja na tok simulacije (na primer, kroz modifikaciju ulaznih parametera dok se simulacija izvršava), takođe značajno smanjuje i vreme koje je korisniku potrebno da putem simulacija ispita neki fizički fenomen. Pored vizuelizacije ove vrste, veoma je korisno i pojednostaviti vizuelizaciju podataka nakon završetka simulacije korišćenjem standardnog formata. Oba navedena slučaja korišćenja smo pokrili kroz integraciju razvijenih programa sa *VisIt* sistemom za analizu i vizuelizaciju podataka.

VisIt je program otvorenog koda koji omogućava vizuelizaciju podataka iz velikih simulacija, čije napredne funkcionalnosti predstavljaju poslednju reč tehnologije u oblasti analize i vizuelizacije podataka. Oslanja se na klijent-server arhitekturu, gde se server izvršava paralelno na sistemu sa distribuiranom memorijom, te se uz pomoć njega može prikazati veliki skup podataka poput onog iz naših MPI programa. VisIt može da pristupi podacima na više načina od kojih smo mi koristili dva pristupa: putem fajlova zapisanih u VTK formatu koje VisIt može da pročita i putem `libsim` biblioteke koja omogućava *in situ* vizuelizaciju.

Proširenje programa da pored dosadašnjeg tekstualnog ili binarnog zapisa svoje podatke zapisuje i u VTK formatu je olakšano postojanjem više biblioteka za ovu namenu, dok se za osnovne potrebe može iskoristiti postojeći slobodan kod koji ne unosi dodatne zavisnosti u program. Na ovaj način smo omogućili programima da ispisuju profile gustine u tri dimenzije i integrisane dvodimenzione profile gustine talasne funkcije koje se kao VTK fajlovi mogu lako vizuelizovati ne samo u VisIt-u već i u drugim programima koji podržavaju ovaj format.

Integracija sa VisIt-om putem `libsim` biblioteke omogućava da se podacima u memoriji pristupi kao da su resursi koje VisIt može da prikaže na ekranu, bez potrebe za njihovim upisom na disk. Pošto putem ove integracije komponenta VisIt sistema zadužena za vizuelizaciju radi zajedno sa procesom koji poseduje podatke, nema dodatne potrebe za razmenom podataka između procesa, te se ovaj pristup izuzetno dobro skalira prilikom izvršavanja na velikom broju računara u klasteru. Putem ove biblioteke smo omogućili vizuelizaciju stanja svih relevantnih promenljivih u programu. Osim vizuelizacije, `libsim` je iskorišćen i za kontrolu same simulacije. VisIt i `libsim` poseduju više mehanizama za kontrolu toka simulacije. Na osnovnom nivou, VisIt omogućava da se definišu funkcije u programu koje se pozivaju pritiskom na dugme u korisničkom interfejsu VisIt-a, što smo iskoritili za osnovne komande pokretanja, zaustavljanja, prekida i ponovnog pokretanja simulacije. Naprednija funkcionalnost je moguća kroz razvoj dodatnog modula koji VisIt pretvori u korisnički interfejs koji se prikazuje korisniku. Kroz ovaj mehanizam moguća je naprednija kontrola simulacije putem prosleđivanja parametra, ali potpuna kontrola trenutno ipak nije moguća, i predmet je aktivnog istraživanja za proširenje u budućoj verziji VisIt-a. I pored nedostataka, kroz ovaj mehanizam smo omogućili izmenu svih bitnih parametera i obezbedili kontrolu toka simulacije do nivoa pojedinačne iteracije glavne petlje. Osim dva navedena pristupa, moguće je napraviti i konzolni interfejs, putem kojeg se programu iz VisIt-a šalju tekstualne naredbe. Ovaj mehanizam je ograničen samo sposobnošću programa da tumači tekstualne naredbe, i uz dovoljno napredan parser moguće je ostvariti potpunu kontrolu nad simulacijom. Ovaj interfejs smo iskoristili da obezbedimo istu funkcionalnost koja je dostupna i preko specijalizovanog grafičkog interfejsa. Razvijeni sistem kontrole simulacije, kao i vizuelizacija podataka nakon završetka simulacije bili su od ogromne koristi tokom istraživanja predstavljenog u poglavlju 9.

Performanse svih implementacija su izmerene u detaljnim testovima predstavljenim u poglavlju 8. Kao osnovna metrika je korišćeno vreme izvršavanja, odnosno mereno je vreme izvršavanja jedne iteracije glavne petlje, usrednjeno nad 1000 iteracija, za različite veličine diskretizacione šeme. U opštem slučaju, vreme trajanja jedne simulacije najviše zavisi od veličine diskretizacione šeme, koja kontroliše broj tačaka a samim tim i broj iteracija u petljama glavnih funkcija. Naravno, vreme izvršavanja simulacija zavisi linearno od ukupnog broja iteracija glavne petlje, koja kontroliše fizičko vreme propagacije i koja je specifična za eksperiment koji se simulira. Imajući ovo u vidu, predložena metrika nam omogućava da steknemo sliku o performansama programa i predvidimo njegovo vreme izvršavanja za buduće simulacije. Svi testovi su izvršeni na PARADOX-IV klasteru Laboratorije za primenu računara u nauci, u okviru Nacionalnog centra izuzetnih vrednosti za izučavanje kompleksnih sistema Instituta za fiziku u Beogradu. PARADOX klaster se sastoji od preko 100 računara, svaki sa dva Sandy Bridge Xeon procesora (ukupno 16 jezgara po računaru), 32 GB RAM memorije i Nvidia Tesla M2090 grafičkom karticom.

Testirano je skaliranje programa koji se izvršavaju na jednom računaru u odnosu na serijsku implementaciju, kao i skaliranje MPI implementacija u odnosu na paralelizacije na jednom računaru. Proučavano je i *jako* i *slabo* skaliranje, gde se jako skaliranje odnosi na slučaj kada je veličina problema (diskretizacione šeme) konstantna, a povećava se broj paralelnih resursa, dok je kod slabog skaliranja količina računskih operacija po pojedinačnom resursu konstantna, a povećava se broj paralelnih resursa, što je praćeno i povećanjem diskretizacione šeme na odgovarajući način. Na osnovu dobijenih rezultata, za sve programe je izračunato ubrzanje, kao odnos vremena koje jedan računarski resurs provede u jednoj iteraciji glavne petlje u odnosu na vreme za istu operaciju u paralelnom programu, kao i efikasnost skaliranja, koja predstavlja ostvareno skaliranje u odnosu na idealno teorijsko linearno skaliranje. Osim predstavljenih rezultata, razvijeni su i modeli performansi većine programa.

Pre ocenjivanja samih performansi, proučavali smo optimizaciju hibridnih implementacija što je poseban problem koji se pojavljuje samo kod hibridnih algoritama. Za razliku od ostalih algoritama kod kojih se optimizacija završava na programskom nivou, tokom implementacije, performanse hibridnih implementacija dodatno, na značajan način, zavise od iskorišćenosti centralnog i grafičkog procesora, koju određuju parametri podele podataka. Izbor optimalnih vrednosti ovih parametara nije lak zadatak, jer zahteva duboko razumevanje performansi hardvera koji se koristi, i u opštem slučaju veoma je teško ručno izabrati optimalne parametre. To je zahtevalo razvoj metoda koji će biti u stanju da izabere optimalne parametre na osnovu pokretanja programa i merenja njegovog vremena izvršavanja, bez intervencije korisnika. Ispitivana su tri pristupa, od naivnog pretraživanja najbolje konfiguracije putem iscrpne pretrage, preko metoda zasnovanog na algoritmu opadajućeg gradijenta, do metaheurističke metode zasnovane na genetskom algoritmu.

U programima koji rešavaju GPJ u tri prostorne dimenzije postoji ukupno 33 parametra koji određuju podelu i obradu podataka na oba resursa, što kao posledicu ima veoma velik prostor pretraživanja. Problem se može olakšati ako parametre grupišemo u četiri grupe čiji se izbor prirodno nameće, na osnovu tačaka u algoritmu na kojim je neophodno da se oba resursa sinhronizuju. Svaki skup parametara je relevantan samo do tačke naredne sinhronizacije, te se može pretražiti zasebno.

I pored navedene podele problema u manje grupe, iscrpno pretraživanje nije odgovarajuće zbog velikog broja kombinacija koje treba proveriti. Ukoliko se umesto svake vrednosti svakog parametra isproba svaka n-ta vrednost, možemo značajno smanjiti vreme ovakve pretrage, ali uz rizik da odabrano rešenje ne bude zaista optimalno, odnosno da se u pretraživanju slučajno „preskoči" optimalna kombinacija parametara.

Drugi metod koji smo ispitivali, algoritam opadajućeg gradijenta, je zasnovan na računanju gradijenta funkcije koju minimizujemo, i koja u našem slučaju predstavlja vreme izvršavanja datog programa. Pošto ova funkcija ne može da se napiše u analitičkoj formi, njen gradijent možemo da izračunamo jedino pomoću numeričke aproksimacije. Ovo se pokazalo kao suviše problematično, jer je šum pri merenju vrednosti funkcije značajan, odnosno dva ili više pokretanja programa sa istim vrednostima parametara neće imati identična vremena izvršavanja, pri čemu razlike imaju veliki uticaj na izvršavanje algoritma pretrage. To znači da algoritam često može pretragu da usmeri u pogrešnom pravcu. Takođe, pošto ovakav algoritam u osnovi radi sa realnim brojevima, a vrednosti naših parametara mogu biti samo celobrojne, javlja se i problem izbora novih vrednosti parametara na osnovu rezultata jedne iteracije algoritma. Ove probleme smo pokušali da rešimo na više načina, međutim nijedan pokušaj nije uspeo da u potpunosti otkloni sve manjkavosti ovog pristupa, što nas je navelo na zaključak da algoritam opadajućeg gradijenta nije pogodan za problem koji smo pokušali da rešimo.

Treći pristup je baziran na genetskom algoritmu (GA) koji je dizajniran da oponaša prirodnu selekciju, po uzoru na biološku evoluciju u prirodi. GA funkcioniše tako što kreira populaciju od individualnih probnih rešenja problema, koju potom oceni, modifikuje i kreira novu. Ovaj proces se ponavlja dok se ne dođe do željenog rešenja. Inicijalna populacija se obično formira slučajnim izborom, što omogućava ovom algoritmu da započne pretragu prostora iz više tačaka. Svako individualno rešenje u populaciji se ocenjuje uz pomoć funkcije koju zadaje korisnik. Na osnovu ocena, najbolje individue se izaberu za „reprodukciju”, i na osnovu njih se formira nova generacija, a proces se ponavlja. U našem slučaju, individue u populaciji predstavljaju jednu (validnu) kombinaciju parametara podele posla u hibridnom algoritmu, dok je funkcija kojom se ocenjuju vreme izvršavanja programa sa parametrima koje definišu individuu. Kvalitet GA zavisi od implementacije tri ključna operatora: selekcije, reprodukcije i mutacije. Svaki od operatora može biti implementiran na više načina, i za sva tri smo razvili i testirali više potencijalnih implementacija.

Selekcija diktira način na koji se biraju najbolje individue iz populacije. Ovaj izbor može biti proporcionalan njihovoj oceni, a može biti zasnovan i na nekoj drugoj metrici ili pak proizvoljan. Treba imati u vidu da je moguće da dođe do prerane konvergencije ka neoptimalnom rešenju ukoliko se biraju samo najbolje individue, odnosno potrebno je napraviti operator selekcije koji ne zanemaruje sva rešenja i daje evolutivnu mogućnost nekim od slabije ocenjenih individua. Ovo se najčešće radi tako što se relaksira selekcija u prvih nekoliko generacija, da bi neke individue sa slabijom ocenom nakon reprodukcije omogućile bolje pretraživanje prostora. Isprobali smo tri oblika selekcije i među njima utvrdili da oblik selekcije poznat pod nazivom *turnir* daje najbolje rezultate. Ovakva selekcija podrazumeva da se slučajno izabere određen broj jedinki iz populacije i iz ovog skupa odabere najbolja. Kontrolom veličine turnira se može smanjiti pritisak na izbor najboljih jedinki i sprečiti prerana konvergencija.

Reprodukcija je operator koji određuje način na koji će se parametri od dve izabrane individue kombinovati da se napravi nova jedinka, a da se pritom dobije validna kombinacija, odnosno skup parametara pomoću kojih se program može izvršiti. Mi smo ispitali više implementacija, međutim nijedna se nije pokazala kao značajno bolja od drugih.

Poslednji operator, mutacija, podrazumeva izmenu slučajno izabranog parametra u jedinki i ekvivalentan je slučajnom prolazu kroz prostor pretraživanja. Mutacija funkcioniše sa malom verovatnoćom, u našem slučaju do 5%. Uticaj mutacije u GA je diskutabilan i često osporavan, međutim u našem slučaju je bio koristan za sprečavanje konvergencije ka neoptimalnom rešenju u slučaju kada se koristi mala populacija.

Poređenjem tri metoda optimizacije na različitim veličinama diskretizacione šeme utvrdili smo da GA daje najbolje rezultate. Metod iscrpnog pretraživanja je uspevao da nađe skup parametara blizak optimalnom, ali predloženo rešenje obično nije bilo optimalno, usled prevelikog broja koraka koji bi ovaj metod morao da koristi da bi uspeo da ispita veći deo prostora pretrage. Metod zasnovan na algoritmu opadajućeg gradijenta je pokazao najgore performanse, iz razloga koje smo već naveli. GA je pronalazio odlična rešenja, veoma blizu najboljeg rešenja otkrivenog ručnom proverom, i jedini njegov nedostatak je to što osim za slučaj najmanje diskretizacione šeme, nije uspevao da konvergira ka egzaktno najboljem rešenju. Umesto toga, u poslednjoj generaciji GA nalazilo se nekoliko veoma sličnih rešenja koja su sva veoma blizu najbolje kombinacije parametara, što je dovoljno dobar rezultat za naše potrebe. Pošto se GA pokazao kao bolje rešenje od preostala dva pristupa, izabrali smo njega za optimizaciju hibridnih programa i pristupili testiranju.

Ubrzanje i jako skaliranje OpenMP programa je određeno poređenjem njihovog vremena izvršavanja sa prethodno objavljenim serijskim programima. Pritom su testirane performanse sa različitim brojem OpenMP niti, od jedne do 16 niti. Imajući u vidu da je ova implementacija unapređena u odnosu na serijsku, najviše u delu gde se koristi Furijeova transformacija, već sa jednom niti se dobijaju bolja vremena izvršavanja, dok se povećanjem broja niti do 16 dobija dodatno ubrzanje od 11 do 12 puta, u zavisnosti od programa. Efikasnost skaliranja je veoma dobra, od 60-80%. Programi koji rade u jednoj prostornoj dimenziji ostvaruju manje ubrzanje, usled dela programa koji je ostao serijski. U ovom slučaju maksimalno ubrzanje je između 2 i 3, dostupno već sa 4 niti, i daljim povećanjem broja niti se ne dobija značajnije ubrzanje. Dobijeni rezultati su saglasni sa Amdalovim zakonom i mogu se modelirati na osnovu njega, što je detaljnije prikazano u poglavlju 8. Slabo skaliranje OpenMP programa je takođe testirano, pri čemu smo se fokusirali na programe koji rešavaju GPJ u tri prostorne dimenzije. Efikasnost ovog skaliranja je takođe veoma dobra, gde programi za propagaciju u realnom vremenu ostvaruju efikasnost od oko 75%, dok programi u imaginarnom vremenu ostvaruju nešto nižu efikasnost, oko 60%. Ova razlika je usled upotrebe kompleksnih brojeva u propagaciji u realnom vremenu, što zahteva više računskih operacija, dok se pri radu sa realnim brojevima puno iskorišćenje procesora ne ostvaruje usled ograničenog kapaciteta protoka memorije. Pored toga, u imaginarnom vremenu je obavezan i dodatan korak normalizacije talasne funkcije, što takođe negativno utiče na performanse ovih programa. Ovaj oblik skaliranja smo takođe modelirali na jednostavan način, funkcijom sa jednim parametrom, koji odgovara frakciji koda koji se izvršava paralelno.

CUDA programi su testirani malo drugačije, jer grafički procesor funkcioniše kao jedan, paralelan računarski resurs. Samim tim, nije moguće testiranje ove implementacije promenom broja paralelnih računskih resursa. Međutim, promenom veličine diskretizacione šeme možemo dobiti uvid u ponašanje grafičkog procesora i predvideti njegove performanse. U našim testovima, CUDA implementacija je pokazala ubrzanje od 10 do 14 za programe u dve prostorne dimenzije, u zavisnosti od tipa vremenske propagacije. Programi koji rešavaju GPJ u tri prostorne dimenzije su pokazali nešto manje ubrzanje, od 7 za programe u imaginarnom vremenu do 14 za programe u realnom vremenu. Najveća ubrzanja su ostvarena za diskretizacione šeme koje nisu ni suviše male, a ni previše velike. Rad sa malim diskretizacionim šemama ne zahteva dovoljno računskih operacija da bi se grafički procesor u potpunosti iskoristio, dok se u slučaju velikih šema, koje u potpunosti popunjavaju memoriju grafičkog procesora, takođe vidi pad performansi usled potrebe za prebacivanjem podataka iz glavne memorije računara.

Proučavanje performansi programa koji se izvršavaju na jednom računaru završeno je testom hibridne implementacije. S obzirom na to da su testovi OpenMP programa pokazali da je potrebno koristiti svih 16 niti u programima koji rešavaju GPJ u više od jedne prostorne dimenzije, u ovim testovima smo uvek koristili maksimalni broj niti (16), dok je optimalna raspodela podataka između centralnog i grafičkog procesora dobijena korišćenjem genetskog algoritma za zadatu veličinu diskretizacione šeme. Osim u slučaju malih diskretizacionih šema, ova implementacija pruža najbolje performanse od svih programa koji se izvršavaju na jednom računaru. Za male diskretizacione šeme, nakon podele posla, nijedan od resursa nema dovoljno podataka za obradu i ne može da bude u potpunosti iskorišćen, te se ne ostvaruje ubrzanje. Takođe, do pada performansi dolazi i za velike diskretizacione šeme, jer u ovom slučaju ograničenje dostupne memorije na grafičkom procesoru onemogućava dalje ubrzanje. Algoritam za optimizaciju parametara za velike diskretizacione šeme brzo konvergira ka modelu raspodele posla gde je memorija grafičkog procesora u potpunosti iskorišćena, što je znak da nam za bolje performanse treba više memorije na GPU.

MPI programi su testirani na različitom broju računara u klasteru, od 2 do 32. Na svakom računaru je pokrenut jedan MPI proces, koji u slučaju OpenMP/MPI i Hybrid/MPI implementacije pokreće svojih 16 niti, dok u slučaju CUDA/MPI implementacije pristupa instaliranom grafičkom procesoru. Osnova za poređenje tri MPI implementacije je bila ekvivalentna implementacija za jedan računar, tj. OpenMP/MPI programi su poređeni sa OpenMP programima koji se izvršavaju sa 16 niti, CUDA/MPI programi su poređeni sa CUDA implementacijom koja koristi jednu grafičku karticu, dok su Hybrid/MPI programi upoređeni sa hibridnim programima namenjenim izvršavanju na jednom računaru. Performanse svakog MPI programa, pa tako i programa koje smo mi razvili, umnogome zavise od konkretnog klastera na kom se izvršavaju, najviše od brzine protoka podataka kroz mrežu. Pronalaženje najbolje konfiguracije klastera za konkretan problem zahteva dosta truda i može da stvori pogrešnu sliku o performansama programa. Rezultati koje smo predstavili su dobijeni bez specijalnog podešavanja klastera, sa ciljem da se pokažu osnovne performanse na osnovu kojih bi se, uz razvijene modele, moglo predvideti ponašanje programa na drugim klasterima. Zbog toga razvijena metodologija testiranja performansi programa predstavlja jedan od značajnih doprinosa ove teze.

Prilikom testiranja jake efikasnosti MPI programa za fiksnu veličinu problema, odnosno bez izmene veličine diskretizacione šeme, jasno je da će na dovoljno velikom broju računara performanse početi da opadaju, usled nedovoljne količine posla po jednom procesu i prevelike cene razmene podataka. Prema tome, iako povećanjem broja MPI procesa odnosno računara očekujemo da će vremena izvršavanja glavne petlje početi da se smanjuju, takođe očekujemo i da će za dovoljno veliki broju procesa doći do zasićenja, nakon čega će efikasnost početi da opada. Ovo znači da iako na malom broju procesa možemo da uočimo linearno skaliranje, modeli performansi ne mogu biti bazirani na linearnim funkcijama i moraju da uzmu u obzir i parametre vezane za brzinu mreže. Modeli koje smo razvili razmatraju ove detalje i pokazuju dobro slaganje sa eksperimentalnim rezultatima za performanse sve tri MPI implementacije.

U testovima jakog skaliranja, OpenMP/MPI programi su pokazali odlično ubrzanje, koje ide do 17 za programe u imaginarnom vremenu i do 22 za programe u realnom vremenu, koristeći 32 računara i ukupno 512 jezgara. Efikasnost ostaje uglavnom konstantna i kreće se u rasponu od 40% do 60%. Pritom, ubrzanje raste skoro linearno, što čini ovu implementaciju veoma pogodnom za simulacije sa veoma velikim diskretizacionim šemama. Slično ponašanje je uočeno i kod CUDA/MPI programa, gde je ubrzanje manje, od 9 do 10, uz manju efikasnost, od 30% do 40%. Iako je ubrzanje manje, treba imati u vidu da su apsolutna vremena izvršavanja CUDA/MPI programa manja od OpenMP/MPI programa, što ovu implementaciju čini veoma pogodnom za klastere koji sadrže grafičke procesore, naročito manje klastere. Na Hybrid/MPI programima je uočljiv pad efikasnosti sa povećanjem broja procesa, pošto je količina posla po samom procesu veoma mala, te dalja podela između centralnog i grafičkog procesora ne donosi bolje performanse. Međutim, ako posmatramo samo vremena izvršavanja, na manje od 16 procesa Hybrid/MPI implementacija ima najbolje performanse, što znači da je ova implementacija najpogodnija ako količina posla po procesu ostane dovoljno velika da opravda upotrebu hibridnog algoritma. U suprotnom, treba razmotriti bilo koju od druge dve implementacije.

U testovima slabog skaliranja korišćena je slična metodologija kao i u testovima OpenMP programa. Kao osnovu za poređenje koristili smo vreme izvršavanja jedne iteracije glavne petlje MPI programa pokrenutog sa jednim procesom. Pokretanje MPI programa sa samo jednim procesom se ne preporučuje jer se operacija transponovanja podataka izvršava iako nema potrebe za njom. Ovo takođe može da utiče na rezultate testova, što je vidljivo u rezultatima OpenMP/MPI i CUDA/MPI implementacija. OpenMP/MPI programi, koji se oslanjaju na FFTW biblioteku za transponovanje podataka, pokazuju veoma dobre performanse kad se koristi samo jedan proces, jer FFTW prepozna ovaj slučaj i podatke transponuje na efikasniji način. Usled toga, prividno se dobija efikasnost od samo 40%. Međutim, ukoliko bi se kao osnova za poređenje koristilo vreme izvršavanja programa sa dva procesa, videlo bi se da efikasnost ostaje iznad 80% prilikom upotrebe 32 procesa, što je odličan rezultat. Sa druge strane, CUDA/MPI implementacija ostvaruje veoma loše performanse kad se koristi samo jedan proces, što je posledica upotrebe memorije centralnog procesora za operaciju transponovanja u ovom slučaju. Sa dva ili više procesa ovi programi ostvaruju mnogo bolje vreme izvršavanja, pa efikasnost može da bude čak i veća nego kod OpenMP/MPI programa, što nije očekivano jer je algoritam za transponovanje inferioran u odnosu na rutine koje pruža FFTW biblioteka. U ovom slučaju potrebno je uzeti apsolutno vreme izvršavanja na 16 procesa kao osnovu za poređenje da bi se dobila prava slika o efikasnosti skaliranja ove implementacije, koja je manja nego kod OpenMP/MPI implementacije. U testovima Hybrid/MPI implementacije se dobije efikasnost skaliranja od oko 75% i, usled upotrebe glavne memorije za transponovanje, ova implementacija nema probleme kao CUDA/MPI implementacija i pogodna je za izvršavanje na bilo kom broju procesa. Ako se porede samo vremena izvršavanja, ova implementacija se nameće kao najbolje rešenje, jer količina posla po procesu ostaje dovoljno velika da opravda upotrebu hibridnog algoritma. Naravno, podrazumeva se da se pre izvršavanja simulacija optimizuju parametari podele podataka pomoću GA, i ovo optimizaciono vreme takođe treba uzeti u razmatranje prilikom izbora algoritma.

Da rezimiramo, u svim testovima MPI programa koji su obavljeni, uočena je očekivana opadajuća efikasnost skaliranja, prouzrokovana potrebom za transponovanjem podataka u sistemu sa distribuiranom memorijom. Imajući sve rezultate u vidu, poglavlje 8 se završava savetima za izbor najboljeg algoritma za zadati problem.

Programi razvijeni u okviru ove teze se mogu koristiti za modeliranje i proučavanje raznih fizičkih sistema. U okviru poglavlja 9 demonstrirana je njihova upotrebljivost za proučavanje formiranja vorteksa u Boze-Ajnštajn kondenzatima. Korišćeni su MPI programi da bi se simulirali efekti prolaska prepreke kroz kondenzat. Prepreka, u formi odbojnog laserskog snopa, prolazi kroz kondenzat, pri čemu njeno kretanje dovodi do formiranja kvantnih vorteksa. Odvajanje vorteksa od prepreke se dešava jedino ukoliko se ona kreće brzinom većom od kritične, u skladu sa Landauovim kriterijumom za superfluidnost. Pošto se kretanje prepreke simulira kao promena potencijala u GPJ jednačini, programi su morali da budu izmenjeni tako da rade sa dinamičkim, vremenski zavisnim potencijalom da bi se ovo istraživanje omogućilo. Izmene programa su bile veoma jednostavne, što ide u prilog algoritmu podeljenog koraka koji nam omogućava da sve izmene budu lokalizovane u funkcijama koje su jednostavne za razumevanje i lake za paralelizaciju, i pri tom su odvojene od Krenk-Nikolson šeme.

Pomoću izmenjenih programa smo najpre simulirali rezultate ostvarene u skorašnjem eksperimentu [97], u kojem je merena kritična brzina na kojoj se formiraju rotirajući parovi vorteksa u Boze-Ajnštajn kondenzatu sačinjenom od atoma natrijuma. Pošto ovi atomi nemaju dipolni moment, programi su dodatno izmenjeni i uklonjeno je računanje vrednosti dipol-dipol interakcije.

Pokazana je saglasnost između rezultata dobijenih numeričkim simulacijama i u eksperimentu, što predstavlja jaku nezavisnu proveru ispravnosti naših algoritama i njihovih implementacija.

Osim navedene simulacije kondenzata sa atomima bez dipolnog momenta, takođe smo istraživali i formiranje vorteksa u kondenzatu od atoma disprozijuma, kod kojih se mora uzeti u obzir snažna dipol-dipol interakcija. Korišćena je ista metodologija za određivanje kritične brzine neophodne za formiranje vorteksa i proučen je uticaj jačine kontaktne i dipol-dipol interakcije na kritičnu brzinu. Zavisnost kritične brzine od jačine dipol-dipol interakcije do sada nije proučavana eksperimentalno, pa rezultati predstavljeni u poglavlju 9 predstavljaju originalni naučni doprinos koji će biti osnova za dodatna istraživanja u narednom periodu.

Budući pravci istraživanja uključuju razvoj paralelnih numeričkih algoritama za višekomponentne Boze-Ajnštajn-kondenzovane sisteme, za brzo rotirajuće sisteme, kao i za druge vrste interakcija između atoma. Glavni izazov u razvoju algoritama za višekomponentne sisteme predstavljaće njihovi povećani memorijski zahtevi, pa očekujemo da će algoritmi za računarske klastere sa distribuiranom memorijom biti osnova za dalje istraživanje u ovom pravcu. Sa druge strane, brzo rotirajući sistemi će zahtevati izmenu Krenk-Nikolson šeme, zbog prisustva dodatnih prostornih izvoda u Hamiltonijanu. Novi programi koji će omogućiti rešavanje GPJ uopštene na ovaj način, posebno uz implementaciju dodatnih tipova interakcije, biće od interesa za širu naučnu zajednicu koja u istraživanju koristi rešavanje ovakvih parcijalnih diferencijalnih jednačina.

Short biography

Vladimir Lončar was born on 28 October 1985 in Novi Sad. He started his undergraduate studies in computer science at the Faculty of Sciences of the University of Novi Sad in 2004, and graduated in 2009. He continued his master studies at the same faculty and obtained a MSc degree in 2011, and immediately afterward he enrolled in PhD studies in computer science. Between 2012 and 2014 he actively participated in the development of the information system of the Faculty of Sciences in Novi Sad, where he was employed. Since 2015 he is employed at the Scientific Computing Laboratory, within the National Center of Excellence for the Study of Complex Systems of the Institute of Physics Belgrade, working on the national research project ON171017 "Modeling and numerical simulations of complex many-body systems". He coauthored two papers in peer-reviewed journals (category M21a), one conference paper (category M33) and two posters at international conferences (category M34). One of the journal papers is marked as a Highly Cited Paper in the Web of Science since its publishing in March 2016 and has a total of 21 citations.

Kratka biografija

Vladimir Lončar je rođen 28. oktobra 1985. godine u Novom Sadu. Osnovne studije na Prirodno-matematičkom fakultetu Univerziteta u Novom Sadu, smer diplomirani informatičar - poslovna informatika, upisao je 2004. godine, a završio 2009. godine. Master studije na istom fakultetu, na smeru informacioni sistemi, završio je 2011. godine. Školske 2011/2012. godine je upisao doktorske studije informatike na istom fakultetu. Od 2012. do kraja 2014. godine je aktivno učestvovao u razvoju informacionog sistema Prirodno-matematičkog fakulteta u Novom Sadu, gde je bio i zaposlen. Od 2015. je zaposlen u Laboratoriji za primenu računara u nauci, u okviru Nacionalnog centra izuzetnih vrednosti za izučavanje kompleksnih sistema Instituta za fiziku u Beogradu, na projektu osnovnih istraživanja ON171017 „Modeliranje i numeričke simulacije složenih višečestičnih sistema". Objavio je dva rada kategorije M21a, jedno saopštenje kategorije M33 i dva saopštenja kategorije M34. Jedan od radova kategorije M21a je u Web of Science označen kao Highly Cited Paper za period od objavljivanja u martu 2016. godine do danas i ima ukupno 21 citat.

University of Novi Sad
Faculty of Sciences
Key Words Documentation

Accession number:	
NO	
Identification number:	
INO	
Document type:	Monograph documentation
DT	
Type of record:	Textual printed material
TR	
Contents code:	Doctoral dissertation
CC	
Author:	Vladimir Lončar
AU	
Advisor:	Dr. Antun Balaž and Dr. Srđan Škrbić
MN	
Title:	Hybrid parallel algorithms for solving nonlinear Schrödinger equation
TI	
Language of text:	English
LT	
Language of abstract	Serbian/English
LA	
Country of publication:	Serbia
CP	
Locality of publication:	Vojvodina
LP	
Publication year:	2017
PY	
Publisher:	Author's reprint
PU	
Publ. place:	Novi Sad, Trg D. Obradovića 4
PP	
Physical description: (no. of chapters/pages/bib. refs/tables/figures/listings/appendices)	10/161/98/16/43/41/0
PO	
Scientific field:	Informatics
SF	

Scientific discipline:	Computer Science
SD	
Subject/Key words:	Numerical simulations, parallel algorithms, hybrid algorithms, nonlinear Schrödinger equation, GPU, OpenMP, MPI
SKW	
UC	
Holding data:	
HD	
Note:	
N	

Abstract:

Numerical methods and algorithms for solving of partial differential equations, especially parallel algorithms, are an important research topic, given the very broad applicability range in all areas of science. Rapid advances of computer technology open up new possibilities for development of faster algorithms and numerical simulations of higher resolution. This is achieved through parallelization at different levels that practically all current computers support.

In this thesis we develop parallel algorithms for solving one kind of partial differential equations known as nonlinear Schrödinger equation (NLSE) with a convolution integral kernel. Equations of this type arise in many fields of physics such as nonlinear optics, plasma physics and physics of ultracold atoms, as well as economics and quantitative finance. We focus on a special type of NLSE, the dipolar Gross-Pitaevskii equation (GPE), which characterizes the behavior of ultracold atoms in the state of Bose-Einstein condensation.

We present novel parallel algorithms for numerically solving GPE for a wide range of modern parallel computing platforms, from shared memory systems and dedicated hardware accelerators in the form of graphics processing units (GPUs), to heterogeneous computer clusters. For shared memory systems, we provide an algorithm and implementation targeting multi-core processors using OpenMP. We also extend the algorithm to GPUs using CUDA toolkit and combine the OpenMP and CUDA approaches into a hybrid, heterogeneous algorithm that is capable of utilizing all available resources on a single computer.

Given the inherent memory limitation a single computer has, we develop a distributed memory algorithm based on Message Passing Interface (MPI) and previous shared memory approaches. To maximize the performance of hybrid implementations, we optimize the parameters governing the distribution of data and workload using a genetic algorithm. Visualization of the increased volume of output data, enabled by the efficiency of newly developed algorithms, represents a challenge in itself. To address this, we integrate the implementations with the state-of-the-art visualization tool (VisIt), and use it to study two use-cases which demonstrate how the developed programs can be applied to simulate real-world systems.

AB	
Accepted by Scientific Board on:	May 12, 2016

AS
Defended:
DE
Dissertation Defense Board:
(Degree/first and last name/title/faculty)
DB

Univerzitet u Novom Sadu
Prirodno-matematički fakultet
Ključna dokumentacijska informacija

Redni broj:	
RBR	
Identifikacioni broj:	
IBR	
Tip dokumentacije:	Monografska dokumentacija
TD	
Tip zapisa:	Tekstualni štampani materijal
TZ	
Vrsta rada:	Doktorska disertacija
VR	
Autor:	Vladimir Lončar
AU	
Mentor:	dr Antun Balaž i dr Srđan Škrbić
MN	
Naslov rada:	Hibridni paralelni algoritmi za rešavanje nelinearne Šredingerove jednačine
NR	
Jezik publikacije:	Engleski
JP	
Jezik izvoda:	Srpski/Engleski
JI	
Zemlja publikovanja:	Srbija
ZP	
Uže geografsko područje:	Vojvodina
UGP	
Godina:	2017
GO	
Izdavač:	autorski reprint
IZ	
Mesto i adresa:	Novi Sad, Trg D. Obradovića 4
MA	
Fizički opis rada:	10/161/98/16/43/41/0
(broj poglavlja/strana/lit. citata/tabela/slika/listinga/priloga)	
FO	
Naučna oblast:	Informatika
NO	

Naučna disciplina:	Računarske nauke
ND	
Predmetna odrednica/Ključne reči:	Numeričke simulacije, paralelni algoritmi, hibridni algoritmi, nelinearne parcijalne difirencijalne jednačine, GPU, OpenMP, MPI
PO	
UDK	
Čuva se:	
ČU	
Važna napomena:	
VN	

Izvod: Numerički metodi i algoritmi za rešavanje parcijalnih diferencijalnih jednačina, naročito paralelni algoritmi, predstavljaju izuzetno značajnu oblast istraživanja, uzimajući u obzir veoma široku primenljivost u svim oblastima nauke. Veliki napredak informacione tehnologije otvara nove mogućnosti za razvoj bržih algoritama i numeričkih simulacija visoke rezolucije. Ovo se ostvaruje kroz paralelizaciju na različitim nivoima koju poseduju praktično svi moderni računari.

U ovoj tezi razvijeni su paralelni algoritmi za rešavanje jedne vrste parcijalnih diferencijalnih jednačina poznate kao nelinearna Šredingerova jednačina sa integralnim konvolucionim kernelom. Jednačine ovog tipa se javljaju u raznim oblastima fizike poput nelinearne optike, fizike plazme i fizike ultrahladnih atoma, kao i u ekonomiji i kvantitativnim finansijama. Teza se bavi posebnim oblikom nelinearne Šredingerove jednačine, Gros-Pitaevski jednačinom sa dipol-dipol interakcionim članom, koja karakteriše ponašanje ultrahladnih atoma u stanju Boze-Ajnštajn kondenzacije.

U tezi su predstavljeni novi paralelni algoritmi za numeričko rešavanje Gros-Pitaevski jednačine za širok spektar modernih računarskih platformi, od sistema sa deljenom memorijom i specijalizovanih hardverskih akceleratora u obliku grafičkih procesora, do heterogenih računarskih klastera. Za sisteme sa deljenom memorijom, razvijen je algoritam i implementacija namenjena višejezgarnim centralnim procesorima korišćenjem OpenMP tehnologije. Ovaj algoritam je proširen tako da radi i u okruženju grafičkih procesora korišćenjem CUDA alata, a takođe je razvijen i predstavljen hibridni, heterogeni algoritam koji kombinuje OpenMP i CUDA pristupe i koji je u stanju da iskoristi sve raspoložive resurse jednog računara.

Imajući u vidu inherentna ograničenja raspoložive memorije koju pojedinačan računar poseduje, razvijen je i algoritam za sisteme sa distribuiranom memorijom zasnovan na Message Passing Interface tehnologiji i prethodnim algoritmima za sisteme sa deljenom memorijom. Da bi se maksimalizovale performanse razvijenih hibridnih implementacija, parametri koji određuju raspodelu podataka i računskog opterećenja su optimizovani korišćenjem genetskog algoritma. Poseban izazov je vizualizacija povećane količine izlaznih podataka, koji nastaju kao rezultat efikasnosti novorazvijenih algoritama. Ovo je u tezi rešeno kroz integraciju implementacija sa najsavremenijim alatom za vizualizaciju (VisIt), što je omogućilo proučavanje dva primera koji pokazuju kako razvijeni programi mogu da se iskoriste za simulacije realnih sistema.

IZ	
Datum prihvatanja teme od strane NN veća:	12. maj 2016.
DP	
Datum odbrane:	
DO	
Članovi komisije: (Naučni stepen/ime i prezime/zvanje/fakultet)	
KO	
Predsednik:	dr Dragan Mašulović, redovni profesor, Prirodno-matematički fakultet, Univerzitet u Novom Sadu
Mentor:	dr Srđan Škrbić, vanredni profesor, Prirodno-matematički fakultet, Univerzitet u Novom Sadu
Mentor:	dr Antun Balaž, naučni savetnik, Institut za fiziku u Beogradu
Član:	dr Nataša Krejić, redovni profesor, Prirodno-matematički fakultet, Univerzitet u Novom Sadu
Član:	dr Miljko Satarić, akademik, redovni profesor Fakultet tehničkih nauka, Univerzitet u Novom Sadu