Numerical Radiation Transport Algorithms for Emergent Computer Architectures

Dissertation
zur Erlangung des Doktorgrades
an der Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Physik
der Universität Hamburg

vorgelegt von

Viktoria Wichert

Hamburg
2018

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Gutachter/innen der Dissertation:	Prof. Dr. Peter Hauschildt Prof. Dr. Michael Hinze
Zusammensetzung der Prüfungskommission:	Prof. Dr. Peter Hauschildt Prof. Dr. Ingenuin Gasser Prof. Dr. Robi Banerjee Prof. Dr. Jochen Liske Prof. Dr. Marcus Brüggen
Vorsitzende/r der Prüfungskommission:	Prof. Dr. Jochen Liske
Datum der Disputation:	26.04.2019
Vorsitzender Fach-Promotionsausschusses PHYSIK:	Prof. Dr. Michael Potthoff
Leiter des Fachbereichs PHYSIK:	Prof. Dr. Wolfgang Hansen
Dekan der Fakultät MIN:	Prof. Dr. Heinrich Graener

Numerical Radiation Transport Algorithms for Emergent Computer Architectures

Abstract

The objective of this thesis is to research effective techniques for solving the 3D radiative transfer problem and to equip the radiative transfer algorithm of the PHOENIX/3D atmosphere modeling code with an alternative parallel solver.
Due to its inherently parallel design and mirroring of the radiative transfer problems' mathematical properties, e.g., narrow-bandedness, a modified parallel Gauss algorithm was selected. It was implemented as a stand-alone OpenCL code, as well as a MPI code, which is already a functioning part of PHOENIX/3D.
Both implementations produce correct results. However, at the moment, the MPI-implementation of the modified Gauss method needs significantly longer to finish execution than the original parallel Jacobi solver of PHOENIX/3D. On the other hand, it requires less iterations to converge, which is a favorable property, especially for large problems.
In summary, the modified parallel Gauss method does indeed work as a solver for the 3D radiative transfer problem, although further optimization is necessary for it to become a reasonable alternative to the original in-use solver of PHOENIX/3D.

Numerische Strahlungstransport-Algorithmen für moderne Computerarchitekturen

Zusammenfassung

Das Ziel dieser Arbeit ist es, effektive Methoden zur Lösung des 3D Strahlungstransportproblems zu finden, und den PHOENIX/3D Code zur Atmosphärenmodellierung mit einem alternativen, parallelen Löser dafür auszustatten.
Es wurde ein modifizierter, paralleler Gauss-Algorithmus ausgewählt, da er grundsätzlich parallel entworfen wurde und mathematische Eigenschaften des Strahlungstransportproblems, wie z.B. seine Band-Struktur, widerspiegelt. Der Algorithmus wurde sowohl als eigenständiger OpenCL-Code, als auch als MPI-Code implementiert, welcher bereits in PHOENIX/3D integriert ist.
Beide Implementationen liefern korrekte Ergebnisse. Allerdings benötigt der modifizierte Gauss-Löser signifikant länger als der bereits in PHOENIX/3D vorhandene parallele Jacobi-Löser. Andererseits braucht er weniger Iteration, um ein konvergiertes Ergebnis zu erhalten, was gerade für große Probleme eine vorteilhafte Eigenschaft ist.
Zusammengefasst ist die modifizierte Gauss-Methode tatsächlich ein funktionierender Löser für das 3D Strahlungstransportproblem, obwohl noch weitere Optimierungen notwendig sind, damit er eine echte Alternative zum bereits vorhandenen PHOENIX/3D Strahlungstransport-Löser ist.

Contents

Chapter 1

Introduction

The scope of numerical mathematics lies in developing methods to calculate sufficiently good approximate solutions when analytical solutions are hard to evaluate, unknown or even non-existent. Throughout the history of mathematics, numerical methods were developed in close alignment with the needs of its areas of application (see, e.g., [5]). Advancements in numerical methods often occurred with the corresponding advancements in, e.g., physics, astronomy and engineering. Nevertheless, numerical mathematics was not regarded as a separate mathematical discipline until the 20th century. During this time, numerical methods faced revived popularity due to the development of modern computers and their rapidly increasing computing capacities.
Apart from developing algorithms, modern numerical analysis also addresses error estimates, uniqueness and existence theorems, as well as convergence properties. Still, the field is driven by the demands of modern applications: more complex problems in science and industry lead to the invention of more sophisticated numerical methods. Currently, the possibilities of parallel computing give rise to parallel numerical algorithms, thus computing results faster. In this context, the speed-up is not necessarily a goal in itself, but allows to find solutions to even larger, more complex models in a broad variety of applications.

One recurring challenge in modern applications is the radiative transfer problem. It plays a role in combustion physics, medicine, climatology and astrophysics. Efficient numerical methods for the radiative transfer equation, therefore, approach a broad variety of possible applications. Furthermore, applying exactly tailored parallel numerical algorithms to the radiative transfer problem will speed-up the numerical computations. Carefully considering the portability of code will also make those algorithms accessible on a broad range of computer architectures.
Three-dimensional atmosphere modeling is one of the applications where realistic models depend heavily on modern computing resources, as well as parallel numerical radiative transfer algorithms. It is an inverse problem, in the sense that fitting observational data to model atmospheres allows conclusions on stellar and planetary parameters, e.g., their chemical composition.

The objective of this thesis is to research effective techniques for solving the

radiative transfer problem and to equip the radiative transfer algorithm of the PHOENIX/3D code with an alternative parallel solver. PHOENIX/3D is a tool for modeling stellar and planetary atmospheres (see, e.g., [18]). This includes, among others, finding a numerical solution to the radiative transfer equation via the Operator Splitting method, while simultaneously computing the corresponding level populations of the atoms and molecules in regard.
Due to the complex physics and the large size of the models, parallel computing is necessary to obtain insightful results. Another crucial criterion is portability, so that the code can be executed on different, current and emergent, computer architectures. Implementing an alternative parallel solver into the PHOENIX/3D radiative transfer algorithm should not only result in a speed-up of current set-ups, but also enable even more sophisticated model atmospheres.

Analyzing the computing time of a "typical" parallel PHOENIX/3D run shows that the radiative transfer calculations take up a significant amount of overall runtime, in some circumstances up to 80%. The high computational cost stems from the complexity of the radiative transfer problem, as well as the large amount of dimensions involved, i.e., spacial, angular and wavelength-dimensions. Therefore, it makes sense to approach further parallelization efforts in atmosphere modeling via effectively parallelizing the radiative transfer algorithm.
There are several current efforts to enhance the parallelization of the PHOENIX/3D radiative transfer algorithm. In [4], [35], [28] and [39], some of them are presented. These approaches already resulted in a decrease of runtime and will allow larger, more detailed models as the efforts on this topic proceed. However, none of them focus on solving the emerging linear system of equations. As a consequence, solving the radiative transfer problem effectively gains further relevance as the overall runtime decreases.

In the PHOENIX/3D code, the radiative transfer calculations alternate with the estimation of occupation numbers at every wavelength until the model converges. During each radiative transfer iteration, a similar system of equations is solved for the angle-averaged intensities. The large linear system is generated by the Operator Splitting approach. Between iterations, only the right-hand-side (RHS) of the equations changes; the matrix is constant throughout the calculation. Solving these recurring systems of equations effectively in parallel is the main focus of this work. Therefore, a modified parallel Gauss algorithm was implemented. The algorithm employs a similar approach as the well-known classic Gauss elimination algorithm: a reduced system with favorable properties is computed from the original problem. After solving the reduced system, the solution to the original system of equations can be reconstructed via back-substitution. The reduced system is much smaller than the large original system, e.g., has a matrix size of a few thousands instead of millions. Also, the matrix factorization only needs to be computed once. For all subsequent problems with the same matrix, the factorization can be re-used. This is especially beneficial for the radiative transfer problem, since it requires to repeatedly solve similar systems of equations.
Furthermore, the modified Gauss algorithm, that will be presented in this work,

has other properties that match the radiative transfer algorithm: it is designed for sparse, narrow-banded problems. During the Operator Splitting step in the PHOENIX/3D radiative transfer code, an approximate operator is introduced and for physical and computational reasons chosen to be narrow-banded.

Altogether, the modified Gauss approach is promising to effectively further parallelize the radiative transfer calculations of the PHOENIX/3D atmosphere modeling code: it is designed as a parallel algorithm from scratch, in contrast to the common parallelization of originally serial solvers, and it takes into account that similar systems of equations have to be solved repeatedly. In addition, the algorithm caters precisely to the mathematical properties of the problem, i.e., the narrow-bandedness and diagonal dominance. The new solver promises a speed-up in calculations, especially for complicated physical set-ups that need a large number of iterations, e.g., situations with heavy scattering.

Two versions of the modified Gauss solver have been implemented for this thesis: one stand-alone OpenCL version and one MPI version that is already part of the PHOENIX/3D radiative transfer algorithm.
Both implementations reproduce results correctly and can be used on a broad range of computer architectures and devices. However, their currently high memory requirements severely limit the problem sizes that they can be applied to. Additionally, the execution time of the modified Gauss solver's MPI implementation is high compared to the other solvers currently used in PHOENIX/3D. So there is definitely the need to optimize the implementations with regard to memory management and execution time before they are fit to be used as alternatives to the current solvers in PHOENIX/3D.
On the other hand, the MPI-implementation shows promising convergence behavior, in that it needs fewer iterations to find a solution with the required accuracy.

The remainder of this thesis is structured as follows: Chapter 2, "Parallel Numerical Mathematics", introduces approaches for successfully parallelizing applications, and gives an overview of numerical algorithms for solving linear systems of equations. In Chapter 3, "3D Radiative Transfer in Stellar Atmospheres", radiative transfer is treated both from the physical as well as the mathematical viewpoint, including a description of the numerical radiative transfer algorithms used in PHOENIX/3D. Afterwards, Chapter 4, "Numerical Methods" proposes a different approach to solve the radiative transfer problem numerically. Therefore, the modified parallel Gaussian Elimination algorithm is introduced as alternative solver. After a short detour to an additional group of linear solvers, the Krylov-Subspace Methods, the chapter ends with a summary of how the modified parallel Gauss algorithm was implemented. In Chapter 5, "Tests", those implementations are tested with regard to several requirements, before Chapter 6, "Discussion and Outlook", concludes the thesis with a summary and discussion of results in the context of effective parallel algorithms in radiative transfer applications, as well as future opportunities in this field.

Chapter 2

Parallel Numerical Mathematics

This chapter introduces the basic concepts of parallel numerical mathematics. It focuses on how developing algorithms for parallel execution differs from designing serial programs.
The first section addresses parallel computing with topics such as hardware architecture, measuring performance and parallel programming concepts. Thereafter, an introduction to parallel solvers for linear systems of equations is featured, illustrated by the class of operator splitting methods as they are currently used in PHOENIX/3D.

2.1 Introduction to Parallel Computing

The goal of parallel computing is to use existing computational resources more efficiently. Through adapting code to support parallel execution and employing several processing elements, existing problems can be solved significantly faster. It is well worth a thought to use the decreased execution-time to compute even more physically accurate model scenarios in now realistic timeframes.
To illustrate, Fig. 2.1 shows the execution-times for computing the radiative transfer problem for one wavelength point in different configurations (also see [41]). As expected, the parallel execution is faster than the serial computation. Still, there are significant differences in the run-times of the parallel codes, one parallelized using MPI, the other using OpenCL. A comparison of execution-times on a CPU shows that the code parallelized with OpenCL runs faster than the MPI-parallelized code-version, which is still significantly faster than the serial code. This behavior is consistent on each of the three tested CPU models. Additionally, the OpenCL code was run on a AMD Radeon GPU, which achieved the lowest run-time in this test but is not compatible with MPI code, and an Intel MIC Acceleration card, which also had a low run-time with the OpenCL code and theoretically is capable of executing MPI code.
The way of parallelization that is appropriate for a problem, therefore, depends, among other things, on the kind of devices on which the parallel program is intended to be run on, as well as the degree of parallelism that is intended, namely

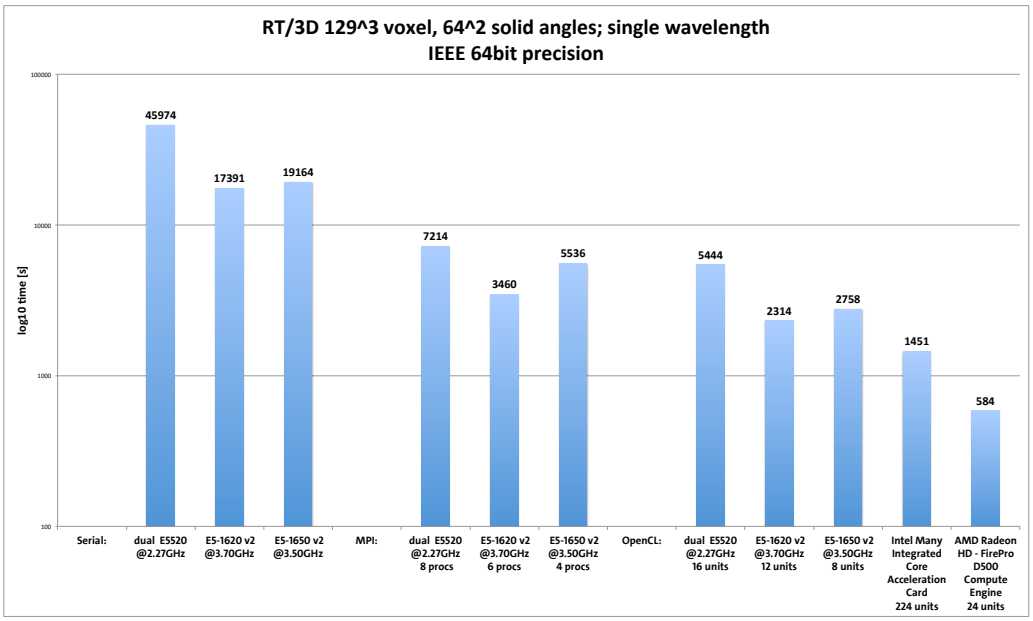


Figure 2.1: Timing comparison of a single wavelength RT computation on different devices (see [41])

if it is run on a massively parallel cluster with thousands of processing elements or, e.g., a GPU in an office computer with several hundreds of processing elements. However, not every task can be parallelized. Sometimes, data- or control-dependencies might be present, which results in a situation where one step cannot be executed until the data is updated or a previous task has been accomplished.

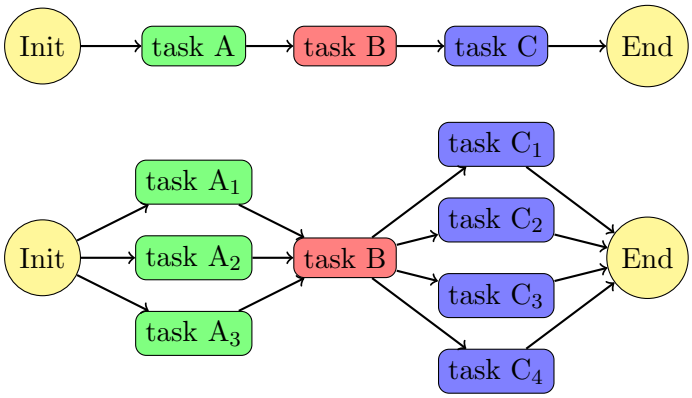


Figure 2.2: Schematic depiction of parallelizing an algorithm.

An illustration of parallelizing an algorithm is depicted in Fig. 2.2. In this example A, B, and C represent tasks which together make up the algorithm. The upper figure shows the serial execution of that algorithm, whereas the lower gives an example for the algorithm's parallelization. B is exemplary for a task that cannot be parallelized. Therefore, only tasks A and C are executed in parallel with different degrees of parallelism. Between tasks, some kind of synchronization takes

place. Depending on the parallelization concept and the intended hardware, this might happen through explicit communication between the processing elements or through shared variables.

Developing effective parallel code hence requires an understanding of not only the algorithms, but also of machine architectures and concepts of parallel programming. A short overview of those is given next.

Common Concepts & Architectures The first microprocessors were build in the 1970s. Since then, their memory and clock-speed increased, while their size decreased. For an overview of the stages of microprocessor development consult, e.g., [8]. Nowadays however, manufacturers strain to design and build yet faster processors, due to physical and power consumption constraints. A tactic to still achieve higher performance is to use several cooperating processors in parallel. Some common approaches are presented here.
Clusters, for instance, are defined as a collection of interconnected stand-alone computers, called *nodes*, that work together as a single integrated computing resource.
Another approach is to design so-called *Multi-core processors*, CPUs containing several independent processing units called cores. A modern multi-core processor typically is composed of two to sixteen cores. The term *many-core processor* is commonly used to describe multi-core processors with a higher number of cores, especially the many-integrated-core products by Intel with typically 64 to 72 cores.
Equally prevalent are architectures containing additional processing units besides the CPU. Work-intensive tasks thereby can be assigned by the CPU to those additional processing units, which might either be very specialized to certain kinds of tasks, e.g., GPUs, or might be comparably generalized as CPUs.
Simultaneous Multi-threading or *Hyper-threading*, on the other hand, is a function of some processors which enables the system to address several virtual cores per actually existing processor-core (see, e.g., [8]). This is partly realized by multiplying parts of the hardware and partly through software-implementation. As a result, the processor can exploit times that one core is suspended due to memory access by using the processor's shared resources on working on another core's instructions. This internal procedure is cheaper and simpler to construct than an actual multi-core processor and, therefore, available on a broad variety of processors.
More detailed data on the devices used for testing the implementations presented in this work is given in Chapter 5.

Despite the broad diversity of hardware architectures, any parallel implementation of an algorithm ought to be as portable as possible, independently of the exact system's setup. Therefore, parallel programming often addresses abstract types of hardware instead of specific devices.
One method to coarsely classify computer architectures is by its memory architecture: different processors either use shared or distributed memory. In the distributed memory case, each processor has private local memory. When sharing

data with cooperating processors, explicit communication is necessary. The programmer of parallel software is responsible for the data to be available for the active processing elements at the right time. This might make parallel programming challenging. Furthermore, the explicit communication is a bottleneck for distributed memory machines, since transporting data between processors takes time, especially in cases where several communication operations happen simultaneously.
In contrast, in the shared memory model processing elements share a global memory space. Communication and synchronization then happen through shared variables that can be read and written by all processing elements.
In both cases, however, memory access timing is a crucial factor for the execution time of a (parallel) program. Terms to describe the features of a system in this regard are *latency* and *bandwidth*. Latency describes the time needed until one memory access operation is complete. Bandwidth, on the other hand, specifies the number of data elements that can be read per time unit, typically given in MB/s or GB/s.
To decrease the average overall memory access times, *caches* were introduced: these small, fast memory elements sits between a processing element and the chip's main memory to hold variables and instructions that are repeatedly accessed. Cache coherency algorithms ensure that all cached versions of a variable agree with its global value. In addition to the decrease in average memory access times, specific combinations of problem sizes and the number of processing elements can lead to a favorable cache use and therefore to a significant decrease in execution time. Unfortunately, this effect can also occur the other way and increase execution times for unfavorable combinations. These effects are hard to predict and lead to unexpected behavior when examining the correlation between execution time and number of processing elements or problem size.
Independently of the memory model, *race conditions* can occur in a parallel code. This unwanted behavior leads to a dependence of the parallel algorithm's result on the order of memory accesses from the processing elements. Race conditions might arise due to errors in the communication between processing elements or when several processing elements access a variable at the same time.

Another method to group hardware architectures is the *Flynn classification*. It divides hardware into four categories: SISD, MISD, SIMD and MIMD. SISD stands for *single instruction, single data* and describes a sequential program. The other classes characterize parallel programs. MISD, *multiple instructions, single data* refers to a theoretical concept, while SIMD, *single instruction, multiple data*, and MIMD, *multiple instructions, multiple data* describe common concepts for parallel software. SIMD stands for a concept of parallelism where one instruction is simultaneously applied to several pieces of data, also referred to as *data parallelism*. An example for hardware operating according to this method are GPUs. MIMD, on the other hand, describes a case where each processing element has an independent set of instructions, which it executes on its own piece of data. This operating principle is used by multi- and many-core processors. Multi-threading also belongs to this category, which sometimes is referred to as *task parallelism*.

Measuring Performance There are several ways to evaluate the quality of a numerical algorithm's implementation. The most common ones are presented here, with regard to assessing a parallel program in comparison to its serial counterpart. It is difficult to separate the performance of code from the performance of the system on which the code is executed. Factors that influence a program's execution time are various, such as machine architecture, compilers, operating system and, of course, the method and even personal style of (parallel) programming. Measures for performance are foremost response time, throughput and computational cost. The *response time* T is defined as the time between the start of a program and its end. Compared to the *CPU-time* T_{CPU} it also includes idle-times. In the case of a program executed in parallel, the *parallel run-time* T_p describes the parallel execution time, from the start of the program until all processes on all processing elements are completed. This includes local computing operations on individual processing elements, communication or global data access (depending on the memory architecture), idle times in case of unbalanced workload, and time for synchronization. According to [30] the influences on execution time listed here are coarsely ordered after their share of overall runtime.
The *throughput* is a measure for machine performance, usually given in MFlops, million floating-point operations per second. To ensure comparability, the throughput is usually given for a set of benchmark programs.
Computational costs C, on the other hand, are defined as the time that all participating processors together need to finish a program's execution. C is determined as the parallel run-time T_p multiplied with the number of involved processors p, $C = T_p p$, and describes the amount of work done by all processors combined. A parallel program is *cost-optimal* if the combined execution time of all processing elements equals the execution time needed by the fastest serial program solving the same problem, $C = T_s$.

With these definitions in mind, the performance of a parallel program can now be judged in comparison to a serial program solving the same problem.
The *speed-up* $S_p(n)$ is given by the ratio of the execution time of the fastest serial program and the parallel run-time, applied to a given problem of size n:

$$S_p(n) = \frac{T_s}{T_p} \tag{2.1}$$

Finding the fastest serial program to solve a given problem can be difficult, so the speed-up is often calculated through a comparison of one serial and one parallel implementation of the same algorithm.
In theory, $S_p(n) \leq p$ should hold, although in applications super-linear speed-ups are possible in cases of favorable cache use. On the other hand, $S_p(n) < p$ often happens due to overhead in the parallelization of an algorithm, such as communication and synchronization operations. A parallel program is cost-optimal if $S_p(n) = p$.
Another measure for the quality of a parallelization is *efficiency* $E_p(n)$, defined as

$$E_p(n) = \frac{S_p(n)}{p}. \tag{2.2}$$

Therefore, theoretically $E_p(n) \leq 1$ applies, except for cases of super-linear speed-up.
The *scalability* of a program is a quantity often used to describe its performance in case of an increasing number of processing elements. Ideally, a parallel program's execution time decreases with an increasing number of processing elements p. A scalable method therefore has a constant efficiency when increasing n and p simultaneously (see, e.g. [30]).

There are also approaches to assess the parallel potential of a problem. Due to data or task dependencies, most parallel programs contain serial sections. A simple form of Amdahl's law estimates the effects of serial program parts on the possible speed-up of a parallelization

$$S_p(n) \leq \frac{1}{f}, \tag{2.3}$$

where f is the ratio of a program that needs to be executed serially (see [30] for details). This does not give any information about the speed-up with increasing problem size n, though.

Parallel Programming When solving a problem in parallel, usually one or several sequential algorithms already exist. The goal of parallelization then is to decrease the program's runtime through parallelization in comparison to the best serial algorithm. Alternatively, one aspires to compute more sophisticated models in terms of resolution or more realistic physical effects in a fixed amount of time. A challenge for an effective parallelization is the need to rethink serial structures in the code. Only "translating" the best serial algorithm into parallel code will most likely not result in a significant increase of computational speed, whereas designing numerical algorithms with advantageous parallel structures might.
The details of implementing a parallel algorithm and the preferred programming language depend on the intended application, e.g., on the architecture of the designated machine, the memory model of said machine, and the intended level of parallelization. There are various ways of programming parallel code. PHOENIX/3D can be run in parallel with MPI, OpenCL and OpenMP. The latter can be applied to shared memory models, while OpenCL and MPI allow for execution on machines with distributed memory. Usually, PHOENIX/3D uses a hybrid parallelization consisting of MPI or OpenCL combined with OpenMP.

OpenCL OpenCL stands for *Open Computing Language*. It is a software standard that was released in 2008 and is maintained by the Khronos Group (see, e.g., [16]). Various vendors provide OpenCL implementations for their hardware.

The standard targets heterogeneous parallel platforms, namely platforms containing different computational elements, e.g. CPUs, GPUs, and accelerators, operating jointly through a distributed memory model. The goal of OpenCL is high portability for different heterogeneous platforms, which, in this case, has the consequence that programmers are in detailed control of many low-level settings.
An abstract view of the OpenCL programming model is that a *host* sets up the program and then off-loads tasks as *kernels* onto computational devices for parallel execution. The host is also responsible for I/O, user interaction, loading *memory objects* onto devices and reading modified memory objects back from possibly several OpenCL devices. [16] defines a memory object as "a set of objects in memory that are visible to OpenCL devices and contain values that can be operated on by instances of a kernel".
The host code is written in the C programming language, while kernels are written in the OpenCL programming language, an extended subset of the ISO C99 standard. Usually, kernels are compiled at run-time by the OpenCL compiler to accommodate different devices. Alternatively, pre-compiled kernel-binaries can be loaded at program start. This approach results in a decreased execution time compared to compiling kernels at run-time but restricts portability severely, as binaries are not compatible across vendors and OpenCL implementations (see e.g. [16]). Also, when compiling all kernels at once through the `clCreateKernelsInProgram` command, the order in which kernel-handles are saved in a list can vary depending on the OpenCL implementation. This might result in errors when using the code across different vendor's platforms.
There are several restrictions to the OpenCL kernel programming language, such as the lack of recursion, the lack of pointers to functions, and only a limited set of standard C libraries is available to the kernels. During run-time, the host submits a kernel for execution on a specific device and a defined number of instances of that kernel, called *work-items*, is created and executed in parallel on the device. The work-items can be distinguished by a global ID which is assigned to each work-item upon creation.
Administration of the devices and events concerning them is done by the host via an *OpenCL context*. A context contains OpenCL devices, kernels, memory objects and program objects holding the kernel source code. All interactions between the host and the devices happen through command queues. These interactions include, among others, loading or reading memory objects and executing kernels. A command queue is tied to one specific device, while a context can hold several devices.
Memory objects can be defined as global, constant, local or private. Global means that all work-items can access the memory objects, while constant describes read-only access for all work-items. Local memory objects, in contrast, grant access for all work-items in a *work-group*, whereas private memory is only visible to one work-item. A work-group is a set of work-items. The behavior of local memory when no work-groups are explicitly defined is implementation-dependent: some implementations use one work-group per work-item as default, some define one work-group for all existing work-items as default.
In OpenCL, data parallelism is achieved by assigning different sets of data to

the work-items, while task parallelism is possible via out-of-order execution of different, independent kernels. Synchronization of data is only possible between work-items of one work-group, but not across work-groups. It happens via the `barrier` command and forces all work-items to wait for each other's execution up to the barrier. Afterwards, the execution continues with synchronized global and/or local memory.

MPI MPI, a *Message-passing library interface specification*(see [15]), provides interfaces for communication commands. Bindings are available for Fortran and C. In Fortran, MPI routines are called as subroutines, while in C they are called as functions. The MPI standard is defined and extended by the Message-Passing Interface Forum, which includes, among others, hardware vendors and computer scientists. It defines the MPI syntax and the effect a command must have, but the internal realization is hardware- and vendor-dependent.
MPI targets distributed memory environments but shared-memory implementations are possible. Its goal is to provide efficient communication in heterogeneous environments and portability between different architectures in a user-friendly and thread-safe way.
An MPI program contains processes that can communicate with each other. *Communicators* serve as a way to organize them: the set of processes in a communicator can exchange messages with each other. The default MPI communicator `MPI_COMM_WORLD` contains all processes in a program. Each process in a communicator is assigned a rank. Since a process can be part of several communicators, it might have different ranks for each communicator. Upon program start the number of processes in a MPI program is defined, e.g., as a variable in the program call, although it might change due to existing processes spawning new processes during program execution.
There are several different modes of communication in MPI: it can be either blocking or non-blocking, synchronous or asynchronous. A blocking communication command prevents the sending and receiving processes from returning before the communication is completed, while a non-blocking operation immediately returns. Synchronous communication describes a situation with exactly one sender and one receiver. On the other hand, MPI allows asynchronous communication, where a message send by one process can be broadcasted among several receivers or one process can gather data from several senders. These collective operations also include global reduction operations to, e.g. find global minima, maxima, or sum data up over all processes in a communicator.
In MPI, data parallelism can be obtained by distributing the data over several processes with a common communicator. Task parallelism is possible through disjoint communicators.

After introducing the theoretical and computational aspects of parallel numerical mathematics, the following section will present a frequent application, namely linear solvers.

2.2 Parallel Solvers

This chapter will present parallel numerical methods for the solution of linear systems of equations, focusing on the solvers that are currently used by PHOENIX/3D, such as the Jacobi and the Gauss-Seidel method. Mathematically speaking, both algorithms belong to the class of iterative *operator splitting methods*.
After introducing linear solvers in general and the class of operator splitting methods more precisely, the chapter ends on a illustration of the solvers' parallel potentials.

Linear solvers can be divided coarsely into *direct* and *iterative* methods. A direct method is an algorithm that yields the exact solution of a problem, except for rounding errors, after a finite number of computations. Iterative methods, on the other hand, successively compute approximate solutions by repeatedly applying the same calculation specification, the so-called *iteration scheme*. The iterations are aborted when the approximate solution sufficiently agrees with the (unknown) exact solution. For a system of equations

$$Mx = b \tag{2.4}$$

with $M \in \mathbb{R}^{n\times n}$ and $b \in \mathbb{R}^n$ an iterative scheme has the form

$$x^{(j+1)} = \phi(x^{(j)}, b) \tag{2.5}$$

where $x^{(0)}$ is a starting value. A special case are linear iteration schemes, where ϕ can be written as

$$\phi(x, b) = Xx + Yb, \tag{2.6}$$

with $X, Y \in \mathbb{R}^{n\times n}$.
For the evaluation of an iteration scheme's quality, the concepts of *consistency* and *convergence* are highly relevant:

Definition 2.2.1. *(See [26]) An iteration scheme ϕ is* consistent *to a matrix M, if for all $b \in \mathbb{R}^n$ $M^{-1}b$ is a fixed-point of ϕ to b, i.e.,*

$$M^{-1}b = \phi(M^{-1}b, b). \tag{2.7}$$

An iteration scheme ϕ is convergent*, if for all $b \in \mathbb{R}^n$ and all $x^{(0)} \in \mathbb{R}^n$ a boundary value*

$$\hat{x} = \lim_{j\to\infty} x^{(j)} = \lim_{j\to\infty} \phi(x^{(j-1)}, b) \tag{2.8}$$

exists independently of the choice of $x^{(0)}$.

As a result, the limit of a convergent and consistent iteration series fulfills the linear equation for every starting vector. However, a reasonable choice of starting vector might lead to faster convergence of the iteration method.
In real-life applications, convergence of the series of iterations is usually examined by a stopping criterion, which checks the change in result compared to the previous

iteration step. If the change is small enough, the iteration scheme is assumed to be converged. A common stopping criterion is the relative error

$$||x^{(j+1)} - x^{(j)}|| \leq \epsilon ||x^{(j+1)}||, \tag{2.9}$$

though other definitions are possible.

With the above definitions in mind, the remainder of this chapter focuses on a subset of iterative methods, namely the operator splitting methods that are currently implemented in PHOENIX/3D. Two examples, the Jacobi and the Gauss-Seidel-method, are presented in more detail, along with an overview on how these methods can be parallelized effectively.

Operator Splitting Methods The fundamental idea of operator splitting methods to solve the system of equations Eq. 2.4, $Mx = b$, is to write the operator M as

$$M = M^* + (M - M^*), \quad M^* \in \mathbb{R}^{n \times n}. \tag{2.10}$$

Inserting this into the system of equations leads to

$$x = (M^*)^{-1}(M^* - M)x + (M^*)^{-1}b. \tag{2.11}$$

M^* has to be invertible to allow this conversion.
The above system then yields the iteration scheme

$$x^{(j+1)} = Xx^{(j)} + Yb \tag{2.12}$$

with

$$X := (M^*)^{-1}(M - M^*) \tag{2.13}$$

and

$$Y := (M^*)^{-1}. \tag{2.14}$$

M^* can be chosen arbitrarily, as long as the resulting iteration method is consistent with the original problem and converges towards its solution. Regarding the consistency of operator splitting methods, there is the following theorem:

Theorem 2.2.2. *(See [26]) Be $M^* \in \mathbb{R}^{n \times n}$ a regular matrix. The linear iteration scheme*

$$x^{(j+1)} = (M^*)^{-1}(M^* - M)x^{(j)} + (M^*)^{-1}b \tag{2.15}$$

then is consistent with the matrix M.

Therefore, every operator splitting method is consistent with its original problem, as long as M^* is regular. As for the convergence properties of operator splitting methods, we introduce the spectral radius:

Definition 2.2.3. *(See [38].) The spectral radius of a matrix $M \in \mathbb{R}^{n \times n}$ is defined by*

$$\rho(M) = \max\left\{|\lambda| : \lambda \in \lambda(M)\right\}, \tag{2.16}$$

where $\lambda(M)$ is the set of eigenvalues of M.

The correlation between spectral radius and convergence of an iteration scheme is given by

Theorem 2.2.4. *(See [26]) A linear iteration method ϕ is convergent if, and only if, the spectral radius of the iteration matrix X fulfills*

$$\rho(X) < 1. \tag{2.17}$$

Furthermore, the spectral radius is an indicator for the rate of convergence: the smaller the radius is, the faster the method will converge. Accordingly, every choice of a regular M^* that fulfills the $\rho(X) < 1$ condition, leads to a convergent and consistent iteration scheme. The reason for splitting M in the first place is to choose M^* so that the resulting iteration method has good convergence properties and/or is less computationally expensive.

To illustrate operator splitting methods, the Jacobi method and the Gauss-Seidel method are presented, both currently implemented in PHOENIX/3D.
When discussing the Jacobi method, one needs to define the diagonal matrix D, which contains the diagonal entries of M and is zero otherwise. The approximate operator M^* is chosen as

$$M^*_{\mathrm{J}} = D, \tag{2.18}$$

which defines the Jacobi iteration through

$$X_{\mathrm{J}} = D^{-1}(D - M) \tag{2.19}$$
$$Y_{\mathrm{J}} = D^{-1}, \tag{2.20}$$

where the index "J" refers to "Jacobi".
This definition leads to the iteration scheme

$$x^{(j+1)} = D^{-1}(D - M)x^{(j)} + D^{-1}b, \tag{2.21}$$

which can also be written as

$$x_i^{(j+1)} = \frac{1}{m_{ii}} \left(b_i - \sum_{k=1, k\neq i}^{n} m_{ik} x_k^{(j)} \right) \quad i = 1, \ldots, n \tag{2.22}$$

in a matrix-entry notation. This notation also leads to an easily confirmable convergence criterion for the Jacobi method:

Definition 2.2.5. *A matrix M is* diagonally dominant row-wise*, when its elements m_{ij} satisfy the following condition:*

$$|m_{ii}| \geq \sum_{j=1, j\neq i}^{n} |m_{ij}|. \tag{2.23}$$

M is diagonally dominant column-wise*, if M^T is diagonally dominant.*
If the matrix satisfies the strict inequality, it is called strictly diagonally dominant*.*

If M is strictly diagonally dominant row- or column-wise, the Jacobi method converges towards the exact solution $M^{-1}b$ for all starting vectors $x^{(0)}$ and all right-hand sides b (see [26]).

Another example for an operator splitting method is the Gauss-Seidel iteration. Here, M is written as

$$M = L + D + R, \tag{2.24}$$

where D again holds M's diagonal entries, while L and R are strict lower resp. upper triangular matrices, holding the corresponding entries of M. Inserting this notation into the system of equations can be written as a system with

$$X_{\mathrm{GS}} = -(D+L)^{-1}RY_{\mathrm{GS}} = (D+L)^{-1}, \tag{2.25}$$

where

$$M^*_{\mathrm{GS}} = D + L \tag{2.26}$$

and the "GS" index implying that the matrices belong to the Gauss-Seidel method. The iteration scheme in operator splitting notation then reads

$$x^{(j+1)} = -(D+L)^{-1}Rx^{(j)} + (D+L)^{-1}b. \tag{2.27}$$

In a notation with matrix components the Gauss-Seidel method can also be written as

$$x_i^{(j+1)} = \frac{1}{m_{ii}}\left(b_i - \sum_{k=1}^{i-1} m_{ik}x_k^{(j+1)} - \sum_{k=i+1}^{n} m_{ik}x_k^{(j)}\right) \quad i = 1, \ldots, n. \tag{2.28}$$

Again, this iteration scheme converges towards the exact solution of $Mx = b$ if M is strictly diagonally dominant (also see [26]).

Parallel Implementation Ensuing the introduction to operator splitting methods and two of their more prominent representatives, this paragraph investigates how the Jacobi and the Gauss-Seidel iteration scheme can be implemented in parallel.

For every new iteration step $x^{(j+1)}$ of the Jacobi method, only values from the previous iteration step $x^{(j)}$ are necessary. Furthermore, there are no data dependencies in (2.22): the computation of one component of $x_i^{(j+1)}$ is completely independent of all other entries $x_{l\neq i}^{(j+1)}$. Therefore, the entries of the current iteration can be computed in parallel. Nevertheless, $x^{(j)}$ needs to be saved until the computation of $x^{(j+1)}$ is complete.
With $i = 1, \ldots n$ components, each iteration can be performed in parallel by at most $p = n$ processing elements. After completing one iteration step though, the resulting entries have to be communicated to all other processing elements, since the computation of each $x_i^{(j+2)}$ requires the full vector $x^{(j+1)}$.
Optionally, the Jacobi method can be interpreted as two matrix-vector multiplications, a matrix-matrix multiplication and a vector-vector addition as shown in

(2.21). Since D is a diagonal matrix, computing D^{-1} is trivial. Parallelizing the matrix multiplications corresponds to computing the inner products in parallel with a maximum grade of parallelism of also $p = n$.

In contrast, for the Gauss-Seidel iteration scheme, each iteration step requires the current iteration's entries up to the current component $x_i^{(j+1)}$ and from then on the components of the former iteration step. This way, $x^{(j)}$ can successively be overwritten by $x^{(j+1)}$. However, this also results in a data dependency inside the iteration step: computing $x_i^{(j+1)}$ uses results from that same iteration step, which leads to a situation in which entries of the current $(j + 1)$ iteration can only be computed in succession. Parallelization is still possible inside the matrix-vector multiplications, yet overall not to the same degree as for the Jacobi method.

For more details on the parallel implementation of the Jacobi method, the Gauss-Seidel method, or iterative solvers in general, see, e.g., [30].

Chapter 3

3D Radiative Transfer in Stellar Atmospheres

Radiative transfer is the process of energy being transported through a medium in form of electromagnetic radiation. It is a critical effect when considering such diverse areas as combustion physics, medicine and climatology (see, e.g., [17], [36]). Furthermore, it is a highly relevant process in atmosphere physics.
Most of what can be observed of a star is defined by the processes in the outer layers of that star, the stellar atmosphere. One dominating mode of energy transport in atmospheres is radiative transfer.

As mentioned in the Introduction, modeling stellar atmospheres leads to a better understanding of stars and the possibility to interpret observational data. Since a star is a three-dimensional object, a one-dimensional approximation of the radiative transfer neglects anisotropic stellar activity, such as spots, convection cells and effects caused by fast rotation. Although solving the three-dimensional problem demands more resources than solving a one-dimensional atmosphere model, the results show that the effort is well worth it (see [21]). When modeling stellar atmospheres, we therefore need fast and reliable algorithms to solve the three-dimensional radiative transfer problem.

This chapter is organized as follows: the first section focuses on the physical side of radiative transfer, including a short summary on (stellar) atmospheres and the radiative transfer equation. Based on that, the mathematical side of radiative transfer is outlined in the second section by explaining analytic properties of the equation as well as introducing numerical approaches to solve the radiative transfer problem. In the last section, the current implementation of a radiative transfer solver in PHOENIX/3D is explained.

3.1 Physics

Atmospheres "A stellar atmosphere comprises those layers of a star from which photons escape freely into space, and can be measured by an outside observer." This definition of a *stellar atmosphere* is given by [22].

Since stars do not have a defined boundary, the area of the star where its outer layers merge into the surrounding medium is defined as the stellar atmosphere. It is optically thin enough so that radiation still carrying information from those layers leaves the star. Optically thin means that $\tau_\nu \leq 1$, where the *optical thickness* τ_ν is defined by

$$d\tau_\nu = \chi_\nu dz, \tag{3.1}$$

where χ_ν is the opacity and z the path length.
A special case of optical thickness is the *optical depth* $\bar{\tau}_\nu$ defined by

$$\bar{\tau}_\nu(z) = \int_z^{z_{\max}} \chi_\nu(z')dz', \tag{3.2}$$

where z is the geometrical depth and $\bar{\tau}(z_{\max}) = 0$ at the outer boundary z_{max} (see [22]). This definition makes $\bar{\tau}$ a quantity to describe how far into a material, i.e. an atmosphere, one can see and at the same time a scale for atmosphere depth, increasing in the inward direction. Since $1/\chi_\nu$ is approximately one photon mean free path at frequency ν, the optical depth can be interpreted as "the number of photon mean free paths at frequency ν along a vertical line of sight from $z_{\max}$ down to z" (from [22]). This emphasizes the fact that in the context of atmospheres, optical depth is a more appropriate variable to measure depth than geometrical depth. The more mean free paths a photon is away from the surface, the higher is the probability that it will not escape from the atmosphere, but be destroyed during an absorption process.

Nearly all information that can be gathered about a star is determined by traits of the stellar atmosphere, e.g., element abundances, pressure and temperature. The inner structure of a star can only be examined indirectly. Due to this, stellar spectra are mostly defined by processes in the stellar atmosphere and its structure.
The temperature gradient is the reason why energy is transported from deeper, hotter layers of a star to the outer, generally cooler layers. Different modes of energy transport in the stellar atmosphere are radiative transfer, convection and thermal conduction.

When comparing the definition of stellar atmospheres to planetary atmospheres, several similarities turn up. A planet's atmosphere is defined as an outer layer of gas, which is held by the planet's gravitational influence. This also applies to stellar atmospheres. Furthermore, the definition of a stellar atmosphere also complies with that of planetary atmospheres.
The above introduced traits of stellar atmospheres therefore extend to (extrasolar) planetary atmospheres. Again, most information about (exo-)planets are gained by atmosphere spectroscopy of the planets transiting their host star. Also, just like in stellar atmospheres, radiative transfer plays a crucial role in planetary atmospheres (see, e.g., [34]).

Radiative Transfer The quantities necessary to describe the radiative transfer through a medium are intensity, opacity, emissivity and optical depth/thickness (see above).
The *Monochromatic Intensity* I_ν is defined as the proportionality factor in

$$dE_\nu = I_\nu dt dA d\Omega d\nu, \tag{3.3}$$

which describes the energy that is radiated off a source per infinitesimal time interval dt, per surface area dA and per solid angle $d\Omega$ in the frequency band $d\nu$. Another important quantity is the *mean intensity* J_ν, which describes monochromatic intensity averaged over all solid angles:

$$J_\nu = \frac{1}{4\pi} \int I_\nu d\Omega. \tag{3.4}$$

While energy in form of photons is transported through the stellar atmosphere, the photons interact with the surrounding material, either by emission, absorption or scattering. This behavior is taken into account by introducing emissivity and opacity coefficients, η and χ. The *monochromatic emissivity coefficient* η_ν affects the intensity via $dI_\nu = \eta_\nu dz$ per path length z. The *monochromatic opacity coefficient* $\chi_{\nu\hat{n}} = \kappa_\nu + \sigma_{\nu\hat{n}}$ takes into account the effects of absorption via the *absorption coefficient* κ_ν as well as *scattering* via $\sigma_{\nu\hat{n}}$, which can depend on both, frequency ν and direction $\hat{n}$. It affects the intensity through the relation $dI_\nu = -\chi_{\nu\hat{n}} I_\nu dz$. Note that I_ν is a proportionality factor. The complications when solving the radiative transfer equation originate here.
I_ν can be interpreted as the energy carried by photons along a ray. Considering the change in I_ν along the ray gives the radiative transfer equation:

$$\frac{dI_\nu}{dz} = \eta_\nu - \chi_{\nu\hat{n}} I_\nu. \tag{3.5}$$

In the more general 3D case, the equation has the following form:

$$\hat{n} \cdot \nabla I(\nu, x, \hat{n}) = \eta(\nu, x) - \chi(\nu, x, \hat{n}) I(\nu, x, \hat{n}). \tag{3.6}$$

When introducing the source function $S_\nu = \frac{\eta_\nu}{\chi_{\nu\hat{n}}}$ and applying the definition of the optical thickness τ (3.1), the radiative transfer equation can also be written as

$$\frac{dI_\nu}{d\tau} = S_\nu - I_\nu. \tag{3.7}$$

In case of a local thermodynamic equilibrium (LTE), velocities can locally be described by Maxwell-Boltzmann distribution, while atomic/molecular excitations are approximated by the Saha-Boltzmann distribution (see, e.g., [31]). In case of LTE conditions without scattering, the source function equals the Planck function $B_\nu(T)$:

$$S_\nu = B_\nu(T) = \frac{2h\nu^3}{c^2} \frac{1}{\exp(h\nu/(kT)) - 1}. \tag{3.8}$$

For a more complex scenario, the source function depends on the mean intensity $S_\nu = S_\nu(J_\nu) = S_\nu(\int I_\nu)$. This behavior complicates finding a solution to the radiative transfer equation.

Radiative transfer is of course not the only physical phenomenon that determines the structure of a stellar atmosphere. A simplified 1D atmosphere model can be described by the following three equations (see [40]): the radiative transfer equation, the hydrodynamic equilibrium and the conservation of energy, i.e.,

$$\frac{dI_\nu}{d\tau} = S_\nu - I_\nu, \tag{3.9}$$

$$\frac{dP(z)}{dz} = -g\rho(z), \tag{3.10}$$

$$F(z) = \sigma T_{\text{eff}}^4 = \text{const.}, \tag{3.11}$$

depending on the geometrical depth z. $F(z)$ is the flux density and σ the Stefan-Boltzmann constant.
For the parameters T_{eff}, g and the element abundances, this system of differential equations determines the pressure distribution $P(z)$, the temperature distribution $T(z)$, the density $\rho(z)$ and the intensity I_ν for all ν. From this information, the spectrum of the model stellar atmosphere with the above parameters can be calculated and, after fitting parameters, compared to observed spectra to gain insights into stellar structures.

3.2 Mathematics

This section will focus on the radiative transfer equation's mathematical traits and on how it can be solved.
The radiative transfer equation (3.7) is an integro-differential equation, because the source function S_ν implicitly depends on the solid-angle integral over I_ν (except for the special case of pure LTE conditions without scattering). In the general case though, radiative transfer is described by a partial integro-differential equation. By only considering transfer along multiple characteristic rays, it simplifies to the ordinary integro-differential equation (3.7) along characteristics.
Since the equation's complexity depends heavily on the assumptions made about the environment in the atmosphere (e.g. (N)LTE, scattering), the radiative transfer equation must be solved numerically.

Method of Characteristics In the last section, the radiative transfer equation was derived heuristically along a ray. When deriving this form mathematically, the *method of characteristics* is applied. It reduces a partial differential equation (PDE), i.e. the general radiative transfer equation, to a system of ordinary differential equations (ODEs), i.e. (3.7), which then has to be solved along several characteristic rays. Here, these rays correspond to the photon propagation directions and are coupled ordinary differential equations themselves.
The general static 3D radiative transfer equation is a partial integro-differential

equation of the form

$$\hat{n} \cdot \nabla I(\nu, x, \hat{n}) = \eta(\nu, x) - \chi(\nu, x) I(\nu, x, \hat{n}). \tag{3.12}$$

As an example, the method of characteristics is applied to the 1D spherical radiative transfer equation:

$$a_r \frac{\partial I_\nu}{\partial r} + a_\mu \frac{\partial I_\nu}{\partial \mu} = f(I, r, \mu), \tag{3.13}$$

where I_ν in spherical coordinates depends on r and $\mu = \cos\theta$, where θ is the angle between the direction normal to the surface and the photon propagation direction. The right hand side f stands for $f = \eta_\nu - \chi_{\nu\hat{n}} I_\nu$, while the coefficients a_r and a_μ describe the connections between the variables (see [19]). Depending on the exact context, additional terms can be necessary to describe the problem in regard.
The function's graph $G(I_\nu(r, \mu))$ along a path $(r(z), \mu(z))$ can be parametrized by a parameter z. Its slope can then be written as

$$\frac{dI_\nu}{dz}(r(z), \mu(z)) = \frac{\partial I_\nu}{\partial r}\frac{dr}{dz} + \frac{\partial I_\nu}{\partial \mu}\frac{d\mu}{dz} \tag{3.14}$$

by applying the chain rule. Comparing the result to the original equation reveals that the ordinary differential equation

$$\frac{dI_\nu}{dz} = f \tag{3.15}$$

holds along the characteristics

$$\frac{dr}{dz} = a_r \text{ and } \frac{d\mu}{dz} = a_\mu. \tag{3.16}$$

Note that the characteristic equations are ODEs themselves.
This way, the mathematical approach also leads to the radiative transfer equation (3.7), which typically is solved numerically along its characteristics.

Numerical Approaches The numerical approach taken by PHOENIX/3D belongs to the class of *Discrete Ordinate Methods*. Methods of this type not only discretize in space, but also in the photon propagation direction.
Any discretization of the radiative transfer equation leads to a system of linear equations with considerable sizes, depending on the problem size and the intended accuracy. Most discretization techniques will give rise to a sparse linear system. There are several approaches to solve this system and the most common ones will be presented in a short overview:
As any system of equations, the linear system could be solved directly, e.g., by matrix inversion or LR-decomposition (see, e.g., Ch.4). Unfortunately, direct methods often produce dense interims systems, even when solving sparse systems. For large sparse systems this results in an unacceptable increase in memory requirement. Also, this behavior might lead to higher computing times. For the desired problem size and resolution, these methods are not generally advisable (see [26]).

An exception of this behavior are narrow-banded systems (see [38]).
As an alternative, the resulting system of equations could also be solved by an iterative method. In this approach, the solution to the system of equations is computed only approximately, but since the system itself originated from the discretization of a continuous problem, the additional error might be negligible. Splitting methods (see Chapter 2) belong to this class of linear solvers.
Another approach in iterative methods are multi-grid methods. With these methods, the approximate solution is alternately iterated on a sequence of coarser and finer grids to speed-up convergence. These methods need a considerable amount of memory for managing values on the different grids and additional computing capacity for transferring the variables between grids.
Also belonging to the class of iterative methods are Krylov subspace solvers. One of their features is that the search for an approximate solution is narrowed down to a subspace of the $\mathbb{C}^n$, which is gradually extended when no sufficiently close approximation is found. More information on this type of iterative methods can be found in Chapter 4.
Monte-Carlo methods are also common in calculating solutions to huge systems of equations. They are iterative methods and use a probabilistic approach. Monte-Carlo methods are highly flexible regarding changes in the original problem, but they induce random noise due to their statistical nature and tend to converge slowly at high optical depths (see [29], [14]).

The numerical method used in PHOENIX/3D belongs to the class of iterative, deterministic methods. In general they are known to converge slowly, but there are several convergence acceleration techniques, e.g., an operator splitting approach often titled as *approximate Lambda iteration* (ALI), which is implemented in PHOENIX/3D. It will be explained in detail in the remainder of this section.

Operator Splitting Methods in the Context of Radiative Transfer The first step of this operator splitting approach is computing a *formal solution*. "Formal" because it neglects the implicit dependency of the source function S_ν on the mean intensity J_ν. This way, the radiative transfer equation (3.7), $\frac{dI_\nu}{d\tau} = S_\nu - I_\nu$, falls into the class of constant coefficient linear ordinary equations (because I is only multiplied by a negative identity matrix, which does not depend on τ) and a solution can be found via Duhamel's principle (see [24]):

$$I_\nu(\tau) = I_\nu(0)\exp(\tau_0 - \tau) + \int_{\tau_0}^{\tau} S_\nu \exp(\tau_0 - \tau')d\tau'. \tag{3.17}$$

The formal solution then is integrated over all solid angles Ω to calculate the mean intensity J_ν via (3.4). The equation can now be written as

$$J_\nu = \Lambda_\nu [S_\nu] \tag{3.18}$$

by introducing the *Lambda-operator* Λ_ν. In this form, the radiative transfer equation will be solved numerically.

Discretization A common concept of discrete ordinate methods is to discretize the equation of interest in space, frequency, and photon propagation direction. The spatial domain is discretized as a 3D grid consisting of *voxels*, volumetric pixels, $\vec{x}_i$ and the frequency dependence of J_ν is resolved by discretizing it to a set of discrete frequency bins ν_j. Also, a discrete set of directions μ_k is chosen to track the photon propagation direction. The rays along which the photons propagate correspond to the characteristic rays from applying the method of characteristics to the radiative transfer problem earlier. Just as for the optical depth, the direction of the ray is from the outer boundary towards the center of the star.
Computing those characteristic rays numerically is done by a so called *full characteristics* (FC) approach. Other common algorithms include *long characteristics* (LC) and *short characteristics* (SC).
While the long characteristics method integrates the formal solution along a whole ray from the outer boundary to the currently considered point in the grid, the short characteristics method extrapolates the upstream intensity at the grid point next to the current point in upstream direction and only integrates over these two values. This leads to the LC method being very accurate but computationally more expensive than the SC method.
The full characteristics method is not applied to every voxel in the spacial grid,

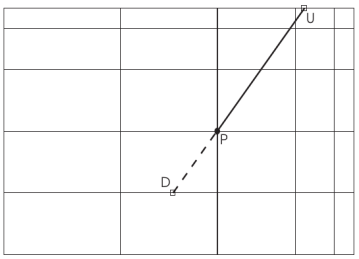


Fig. 1. An illustration of the Long Characteristics (LC) method in Cartesian coordinates. The intensity at point P is computed integrating the transfer equation along the entire ray from the upstream boundary (point U) towards point P.

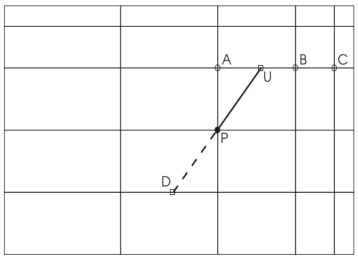


Fig. 2. An illustration of the SC method in Cartesian coordinates. The short characteristic is the line connecting point U to D through P. The value of the intensity at U is determined by quadratic interpolation between the points A, B and C.

Figure 3.1: Comparison of long and short characteristics; from [29]

but instead follows the characteristic rays through the grid from one outer boundary to the other outer boundary. Intersections of characteristic and voxel grid are then the discrete photon propagation. The characteristics are computed so that each voxel is hit at least once by a characteristic. This makes the FC method computationally less expensive than the LC method but still sufficiently adequate. This method is used in PHOENIX/3D to calculate an approximate formal solution.

Through discretizing the problem in the above explained way, the equation in regard, $J_\nu = \Lambda_\nu [S_\nu]$ transforms to its discrete form

$$J_\nu = \Lambda_\nu S_\nu \tag{3.19}$$

with discrete quantities J_ν and S_ν, whereas the operator operation becomes a ma-

trix multiplication with a dense matrix Λ_ν. Solving the system directly can be very computationally expensive compared to the run-times of other parts of the atmosphere model. Therefore an iterative approach is chosen.

The Approximate Lambda Iteration After obtaining an approximation for the formal solution, the solution to the system of equations in regard, $J_\nu = \Lambda_\nu S_\nu$, is solved iteratively. For convenience, the index ν will be neglected for the remainder of this chapter.
The simplest approach would be the iteration

$$J^{(\text{new})} = \Lambda S^{(\text{old})}, \tag{3.20}$$
$$S^{(\text{new})} = S^{(\text{new})}(J^{(\text{new})}), \tag{3.21}$$

but this scheme converges too slowly. To prevent this, the approximate Lambda iteration is applied. Its approach is to split the operator via

$$\Lambda = \Lambda^* + (\Lambda - \Lambda^*), \tag{3.22}$$

where Λ^* is an approximation to the original Lambda operator. Inserting (3.22) into (3.20) leads to

$$J^{(\text{new})} = \Lambda^* S^{(\text{new})} + (\Lambda - \Lambda^*) S^{(\text{old})}, \tag{3.23}$$
$$S^{(\text{new})} = S^{(\text{new})}(J^{(\text{new})}). \tag{3.24}$$

with $\Lambda S^{(\text{old})} = J^{\text{FS}}$, the formal solution obtained through the old source function S^{old}. Interchanging between solving for $J^{(\text{new})}$ and updating $S^{(\text{new})}$ guarantees that the numerical solution to the radiative transfer equation is solved consistently with the rate equations, which influence S. As an additional remark: if $S^{(\text{new})}$ in the above scheme was be replaced by $S^{(\text{old})}$, it would again lead to the simple iteration scheme (3.20).
In case of a two-level atom in a LTE environment with scattering, the source function S would have the following form:

$$S = (1 - \epsilon)J + \epsilon B \tag{3.25}$$

with the *thermal coupling parameter* $\epsilon = \frac{\sigma}{\kappa+\sigma}$ and the Planck function $B(T)$. Inserting this into (3.23), gives the following, well-known iteration scheme

$$[E_n - \Lambda^*(1 - \epsilon)]\, J^{(\text{new})} = J^{\text{FS}} - \Lambda^*(1 - \epsilon)J^{(\text{old})} \tag{3.26}$$
$$S^{(\text{new})} = (1 - \epsilon)J^{(\text{new})} + \epsilon B \tag{3.27}$$

with $J^{\text{FS}} = \Lambda S^{(\text{old})}$ an approximation to the formal solution (3.17) and E_n the identity matrix of size n.
If instead of the above case, multi-level atoms and molecules were regarded, the coefficients for scattering, absorption and emission would change, which in return would change the source function. Anyway, the overall scheme stays the same. For example see [23].

The operator splitting method described above ideally has better convergence properties than the original Lambda iteration. Still, its performance depends greatly on the set-up of the approximate Lambda operator Λ^*. A simple Λ^* decreases computational effort, but at the same time might slow the convergence behavior of the radiative transfer algorithm. The choice on how to set up Λ^* should try to find a balance between these two aims.
There are several approaches to find a suitable approximate Lambda operator. A logical idea is to use a diagonal Λ^* and fill the entries with approximations to the actual Λ operator. The same has been done with tridiagonal (see [27]) and narrow-banded approximate Lambda operators (see [20]). Other common approaches are described in [22].
It can be seen by equation (3.22) that the Approximate Lambda Iteration method belongs to the group of operator splitting methods, which have been described in Chapter 2. Since Λ^* can be chosen freely, one can always design an approximate operator that possesses certain desired traits regarding the ALI's convergence behavior. In [20], e.g., it is shown that an operator splitting scheme using a narrow-banded approximate Lambda operator has a smaller convergence radius than the simple iterative approach in Eq. (3.20).
After setting up a the Λ^* operator, (3.26) still has to be solved in every iteration step. There are many possible methods to do this, among them the already introduced iterative operator-splitting methods from chapter 2. Anyway, it is worth noting that in every iteration step, the matrix stays the same. Also, depending on the choice of Λ^* the matrix might show special traits worth exploiting when deciding on a solver for the linear system.

3.3 Implementation in PHOENIX/3D

PHOENIX/3D is a tool for modeling stellar and planetary atmospheres. This includes finding a numerical solution to the radiative transfer equation via an operator splitting method (see previous section) while simultaneously computing the corresponding level populations of the considered atoms and molecules. PHOENIX/3D is able to not only model the case of LTE (local thermodynamic equilibrium), but also the more complex case of non-LTE, where a solution of the rate equations is necessary to determine level populations. Time-dependent problems can also be solved with PHOENIX/3D, as well as problems including arbitrary velocity fields (e.g. novae). Besides a serial version, the 3D code can also be run in parallel, with MPI, OpenMP and OpenCL versions available. The overall structure of a PHOENIX/3D run is shown in figure 3.2.

In the initial phase, PHOENIX/3D replaces default values for all variables by the desired user input. In the next phase, the equations of state (EOS) are solved, which in the 1D version includes calculating the hydrostatic equilibrium and convection terms, as well as the chemical equilibrium. In 3D, this step is not necessary, because the information is already contained in a pre-calculated hydrodynamics grid, that is passed to the program as input parameter.

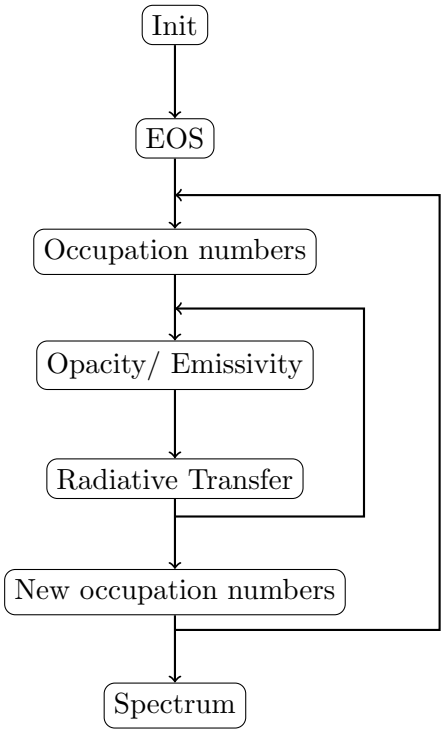


Figure 3.2: Flow chart of a typical PHOENIX/3D run.

Afterwards, a first estimate of occupation numbers for the spectral lines selected in the initial phase is computed and from them the opacity and emissivity coefficients. They are crucial for the next phase, which is the radiative transfer calculation itself. From the computed intensities and the iteratively corrected source function, a new set of corrected occupation numbers is calculated. After rerunning this loop until convergence, a spectrum can be computed from the obtained results for every discrete wavelength bin.

The current implementation for solving the radiative transfer equations iteratively in PHOENIX/3D allows the spatial discretization to be in Cartesian, spherical or cylindrical coordinates. The photon propagation directions are discretized by using a full characteristics method.
The radiative transfer calculations are done in the order shown in figure 3.3.

In PHOENIX/3D, the approximate Lambda operator Λ^* is set up to hold parts of the original Lambda operator Λ. The main diagonals of both operators contain the "self-inflicted" intensity shares of all spatial grid points. The influence of these points on their 26 neighboring voxels is also calculated. Those values enter the side diagonals of Λ^* accordingly, while all other entries are set to zero. That way, Λ^* forms a sparse band-matrix, although the band might be much wider than 26 side-diagonals depending on the ordering of unknowns (see Ch.4 for further information). Λ^* is only computed once for each frequency point at the beginning of the radiative transfer iteration.

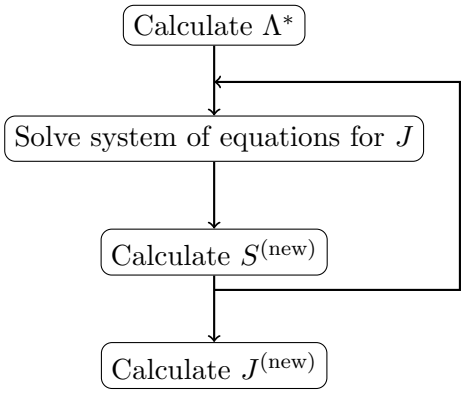


Figure 3.3: Flow chart of Radiative Transfer code structure.

The initial value for S is calculated from the opacity and emissivity estimates (see flow chart 3.2) and the formal solution is obtained via a numerical integration method over Ω. During the iteration steps, the arising systems of equations can be solved either by serial methods, such as a Jacobi solver or a Gauss-Seidel solver, as well as a parallel implementation of the Jacobi solver.
The iteration stops when the relative change, calculated by $\frac{J^{(\mathrm{old})}}{J^{(\mathrm{old})}+J^{(\mathrm{new})}}$, reaches a pre-set accuracy threshold, or when the number of iteration steps exceeds an also pre-set limit. The solution J can then be used to calculate the spectrum of the stellar atmosphere of interest.

Chapter 4

Numerical Methods

In the last chapter, the numerical method used in PHOENIX/3D to solve the radiative transfer problem has been explained. It was mentioned that the operator splitting method results in a linear system of equations, that has to be solved in every iteration step. Furthermore, the matrix of all these systems of equations does not change between iteration steps.
This chapter focuses on how to solve these systems of equations effectively in parallel and the different numerical methods that can be used.

As mentioned in the previous chapters, several classes of solvers are feasible in the radiative transfer context to solve the linear system occurring in each iteration step of the operator splitting method described in Ch.3. They can be divided coarsely into direct and iterative methods. Chapter 2 provided details on iterative operator splitting methods, such as Jacobi and Gauss-Seidel methods. Here however, a parallel direct algorithm, which is a parallel, modified version of the Gaussian elimination, will be presented first, before another class of direct methods, the Krylov subspace methods, will be introduced.

The problem we are left with has the form shown in Eq. (3.26). For the sake of simplicity, in this chapter the systems of equations are denoted as

$$Mx^j = b^j \tag{4.1}$$

where $M \in \mathbb{R}^{n\times n}$ is a sparse narrow-banded matrix. In this context, narrow-banded means that the total bandwidth $2k+1$ fulfills $2k+1 \ll n$, where k is the half-bandwidth. Furthermore, M is assumed to be strictly diagonally dominant column-wise. A sequence of right-hand-sides is given by $b^j \in \mathbb{R}^n$ and the corresponding sequence of solutions is denoted as $x^j \in \mathbb{R}^n$. Upper indices indicate an outer iteration through the operator splitting method, while lower indices describe spatial distributions onto different processing-elements.

Remark on the Assumption of a Diagonally Dominant Matrix: Treating strictly diagonally dominant matrices is a prerequisite for parts of the later presented parallel Gaussian elimination algorithm as well as a property that guar-

antees convergence of both, the Jacobi- and the Gauss-Seidel iterative solvers, which are currently used in PHOENIX/3D (see [20]).
Whether the problem solved in PHOENIX is diagonally dominant depends on the choice of Λ^*. Due to the nature of the operator splitting algorithm, Λ^* can be chosen freely and even without physical meaning (see Ch. 2). Therefore, the problem can be assumed to be diagonally dominant, because it is always possible to choose Λ^* accordingly.
While no general statement can be made on the convergence of non-diagonally dominant matrices, that does not exclude convergence outright but merely implies a conditional behavior. Apart from mathematical theory, even a convergent method might require a huge amount of iteration steps to provide an acceptable solution and will, in any practical approach, be terminated by a stopping criterion, e.g., a small enough change of solution between iterations or a maximum number of iterations, before it is fully converged.

Ordering of Unknowns Since the linear systems $Mx^{(j)} = b^{(j)}$ originate from the discretization of a partial differential equation in three spacial dimensions, the question arises, how the entries of the 3D grid are related to the entries in M and, consecutively, in $x^{(j)}$ and $b^{(j)}$.
Each unknown in the linear system corresponds to one point in the discretized 3D grid. Naively, their order in the linear system is insignificant, since every order will yield the same, but permuted result. However, the ordering of coefficients can have an influence on the properties of the matrix and, therefore, on the convergence behavior of the applied numerical method. Also, the parallel efficiency of an algorithm might depend on the order of unknowns in the linear system.
In the radiative transfer context, this problem becomes even more considerable: a significant property of the matrix $M = E_n - \Lambda^*(1 - \epsilon)$ (see 3.26) is its narrow-bandedness. This is defined as the full bandwidth $2k + 1$ to be significantly less than the number of unknowns n, $2k + 1 \ll n$. The expression "significantly less" is, however, a vague term. Especially for methods specialized on narrow-banded problems, the smaller the bandwidth is the more effective the algorithm works in regard to execution time and memory requirements. Therefore, the ordering of unknowns directly influences the behavior of the numerical method applied to it. To improve the method's performance, one might re-order the unknowns in a way to minimize the matrix' bandwidth. An example for such an ordering algorithm is the Cuthill-McKee method (see [6]).
In PHOENIX/3D, the bandwidth for a medium grid with $nx = ny = nz = 32$ and $n = (2nx + 1)(2ny + 1)(2nz + 1) = 274,625$ total grid points is 4,291. For a more realistic grid of 135,005,697 grid points, corresponding to $nx = ny = nz = 256$, the bandwidth is already greater than 250,000 (see [20]). Those bandwidths are large but still significantly smaller than the matrix dimension n. Even while these matrices qualify as narrow-banded, a different ordering of unknowns and subsequently a smaller bandwidth might encourage shorter execution times.

4.1 A Modified Parallel Gaussian Elimination Algorithm

This section will begin with a short reminder on the serial Gaussian algorithm and its theoretical principles, before presenting a parallel version of the Gaussian elimination, including a version that can be used for problems that do not fulfill the diagonal dominance prerequisite.

4.1.1 The Classic Gaussian Elimination

The classic Gaussian elimination consists of three steps: factorizing the system's matrix, finding the solution to a reduced system, and back-substitute it to find the solution to the original system of equations.
In the factorization step, a LU-decomposition of the form $M = LU$ is carried out, where L is a unit lower triangular matrix and U is an upper triangular matrix. Next, the reduced system $Lz = b$ is solved in a forward elimination step with a interim variable z. The back-substitution step then consists of solving $Ux = z$ to compute the solution x to the original system $Mx = b$.
The LU-factorization is also often referred to as LR-decomposition with $M = LR$.

Another interpretation of the Gaussian elimination is to transform the system of equations into a equivalent triangular form to solve it more easily. This process can be described as a series of matrix multiplications with Gauss transforms G_j which each take care of one column of the original matrix. Each Gauss transform has the form

$$G_j = E_n - \tau_j e_j^T, \tag{4.2}$$

where e_j^T is the j-th unit vector and τ_j is the Gauss vector defined by

$$\tau_j^T = \left(\underbrace{0, \ldots, 0}_{j}, \tau_{j+1}, \ldots, \tau_n \right). \tag{4.3}$$

The factors $\tau_{j+1}, \ldots, \tau_n$ are called multipliers. The Gauss transforms G_j are unit lower triangular matrices.
During the classic Gauss elimination, the Gauss transforms G_j are applied to the original matrix M until it is transformed to an upper triangular matrix. Therefore, R from the LR-decomposition can be defined as

$$G_{n-1} \ldots G_2 G_1 M = R. \tag{4.4}$$

Then

$$L = G_1^{-1} \ldots G_{n-1}^{-1} \tag{4.5}$$

is unit lower triangular and by construction fulfills $M = LR$. Because of the Gauss

transforms' structure (4.2), their inverses can be obtained easily:

$$G_j^{-1} = E_n + \tau_j e_j^T. \tag{4.6}$$

With an understanding of how a LR-decomposition can be found, one can now investigate for which matrices M such a LR-decomposition exists and whether it is unique:

Theorem 4.1.1. *$M \in \mathbb{R}^{n\times n}$ has a LR-factorization if $\det(M(1:j,1:j)) \neq 0$ for $j = 1,\dots,n-1$. If the LR-factorization exists and M is non-singular, then the LR-factorization is unique and $det(M) = u_{11} \cdot u_{22} \dots \cdot u_{nn}$.*

Proof. The interested reader might find the proof in [38], Theorem 3.2.1. □

There is, however, a significant flaw in the Gaussian elimination algorithm: small *pivots*, i.e., elements on the main diagonal of the matrix, in comparison to the non-main-diagonal elements can lead to large numerical errors in the LU decomposition. These errors are augmented during the forward elimination and the back-substitution steps of the algorithm.
Introducing errors to the solution through the LR-decomposition can be avoided by using a pivoting algorithm. Instead of decomposing M as $M = LR$, a decomposition of the form $PMQ = LR$ is performed, where the permutation matrices P and Q ensure a better pivot to non-main-diagonal element ratio by interchanging rows or columns of M, respectively. The case where $Q = E_n$ is called *partial pivoting*, while otherwise it is called *total pivoting*.
For diagonally dominant matrices (see Def. 2.2.5) on the other hand, the need for pivoting is omitted.

Theorem 4.1.2. *If M is strictly diagonally dominant column-wise then M has an LR factorization and the entries of L fulfill $|l_{ij}| \leq 1$. In other words, $P = Q = E_n$.*

Proof. See [38], Theorem 3.4.3 □

As for the stability of the algorithm, the Gaussian elimination is *backwards stable* under certain conditions. This means that the backwards error E is sufficiently small for all approximate solutions $\hat{x}$, where E is defined as the smallest number so that

$$(M + E)\hat{x} = b. \tag{4.7}$$

The bound on E is given by

Theorem 4.1.3. *(See, e.g., [11], [42].) Let M be an $n \times n$ non-singular matrix and $\hat{x}$ be the computed solution of $Mx = b$ by Gaussian elimination with partial pivoting. Then $\hat{x}$ satisfies the equation*

$$(M + E)\hat{x} = b \tag{4.8}$$

with $||E|| \leq f(n)\gamma\beta 10^{-t}$, where t is the precision used in the computation, $f(n) = O(n^3)$, $\beta = \max|m_{ij}|$, $\gamma = \max|m_{ij}^{(l)}/\beta| \leq 2^{n-1}$, and $m_{ij}^{(l)}$ is the ij-th element of M on the l-th step of the elimination.

For banded matrices, the bound on E even improves:

Theorem 4.1.4. *(See, e.g., [11], [7].) Let M be an $n \times n$ non-singular matrix with semi-bandwidth k (with $k > 1$, i.e, a non-diagonal matrix), and let $\hat{x}$ be the computed solution of $Mx = b$ by Gaussian elimination with partial pivoting. Then $\hat{x}$ satisfies the equation*

$$(M + E)\hat{x} = b \tag{4.9}$$

with $||E|| \leq f(n)\gamma\beta 10^{-t}$, where t is the precision used in the computation, $f(n) = O(kn^2)$, $\beta = \max|m_{ij}|$ and $\gamma = \max|m_{ij}^{(l)}/\beta| \leq 2^{2k-3} - (k-2)2^{k-3}$.

In summary, the Gaussian elimination algorithm is appropriate for square, dense, and unstructured matrices. Since the structure of M is known to be sparse and narrow-banded, a classic Gauss elimination is not the method of choice. Furthermore, the Gaussian elimination is usually not used in practical context, because of the amount of time needed for the LR-decomposition of large matrices, especially when using pivoting, and because of potential rounding errors without pivoting. This, however, is not an issue here since M is diagonally dominant. To be useful for a problem of the given size though, some modifications to the classic Gaussian elimination algorithm still have to be made.

4.1.2 The Modified Parallel Gaussian Elimination

The modified parallel Gaussian elimination algorithm that will be presented in this section is an example for a method that establishes parallelism by distributing data to different processing elements while minimizing communication, so that as many operations as possible can be executed independently.

Especially in the case of an optically thick environment or in cases with heavy scattering, the previously presented Operator Splitting method converges slower and, therefore, more iterations need to be carried out. In these cases, it can be beneficial to factorize the matrix M first, which results in solving a reduced system in every subsequent iteration step. The additional effort of the factorization step then is outweighed by the savings in the subsequent iterations and, for the above problem (4.1), should ideally result in a speed-up. Of course, the method is also suited to solve single systems of equations, as the authors of the original study intended, but re-using the matrix factorization for several similar systems of equations should be especially effective.

The version of Gaussian elimination that is presented here, is especially suited for narrow-banded matrices. This kind of problem is not only the result of PHOENIX's operator splitting method, but also often stems from the discretization of ODEs or PDEs from various physical problems. Again, the diagonally dominant property is needed to ensure a stable LR-decomposition. Later in this chapter, a version of the algorithm suited for more general narrow-banded matrices will also be presented.
Particularly narrow-banded problems are in need of suitable algorithms, since applying dense matrix algorithms on sparse systems of equations usually leads to

dense interim problems. This behavior introduces additional and unnecessary effort when solving the sparse problem. In cases where the system is not only sparse but also has a known structure of non-zero entries, such as narrow-banded systems, this disadvantage of dense algorithms is especially bothersome.
Because the special structure of the problem at hand is known in advance, it would still possibly be a waste of resources to use algorithms for general sparse systems. By accommodating the used methods to the special narrow-banded structure of the problem, the computational effort should decrease noticeably.

How the Algorithm Works The system of equations $Mx^j = b^j$ is assumed to have the following structure: the narrow-banded matrix M can be written as a block matrix

$$M = \begin{pmatrix} A_1 & B_1^U & & & & & & & \\ B_1^L & C_1 & D_2^U & & & & & & \\ & D_2^L & A_2 & B_2^U & & & & & \\ & & B_2^L & C_2 & \dots & & & & \\ & & & \ddots & \ddots & \ddots & & & \\ & & & & & \dots & D_{p-1}^U & & \\ & & & & & D_{p-1}^L & A_{p-1} & B_{p-1}^U & \\ & & & & & & B_{p-1}^L & C_{p-1} & D_p^U \\ & & & & & & & D_p^L & A_p \end{pmatrix}, \tag{4.10}$$

while x and b are denoted as

$$x = \begin{pmatrix} x'_1 \\ \zeta_1 \\ x'_2 \\ \vdots \\ \zeta_{p-1} \\ x'_p \end{pmatrix}, b = \begin{pmatrix} b'_1 \\ \beta_1 \\ b'_2 \\ \vdots \\ \beta_{p-1} \\ b'_p \end{pmatrix}. \tag{4.11}$$

A, B^U, B^L, C, D^U and D^L denote different types of sub-matrices, where $A \in \mathbb{R}^{m\times m}$, $C \in \mathbb{R}^{k\times k}$. Along the main diagonal, the sub-matrices A and C are arranged alternately, while the band above and below is covered by the sub-matrix types B^U and D^U, resp. B^L and D^L. m is defined via the relation

$$pm + (p-1)k = n. \tag{4.12}$$

The vectors x and b are arranged as a composite of sub-vectors x', ζ resp. b', β with $x'_i, b'_i \in \mathbb{R}^m$ and $\zeta_i, \beta_i \in \mathbb{R}^k$.
The purpose of denoting the system this way is to enable parallelism by distributing sets of sub-matrices onto p processing elements. The goal is to solve the system of equations by reducing it to a set of problems that can be worked on each set of sub-matrices mostly independently. Afterwards the solution to the original system is assembled from the different processing element's results.

The strategy of the modified parallel Gaussian is to first re-order the narrow-banded system to optimally exploit parallelism. This happens by rearranging x and b to

$$x = \begin{pmatrix} x'_1 \\ x'_2 \\ \vdots \\ x'_p \\ \hline \\ \zeta_1 \\ \zeta_2 \\ \vdots \\ \zeta_{p-1} \end{pmatrix}, b = \begin{pmatrix} b'_1 \\ b'_2 \\ \vdots \\ b'_p \\ \hline \\ \beta_1 \\ \beta_2 \\ \vdots \\ \beta_{p-1} \end{pmatrix}, \tag{4.13}$$

and then re-sorting the equations to fit the new notation. M then reads:

$$M = \left(\begin{array}{ccccc|ccccc} A_1 & & & & & B_1^U & & & & \\ & A_2 & & & & D_2^L & B_2^U & & & \\ & & \ddots & & & & \ddots & \ddots & & \\ & & & A_{p-1} & & & & \ddots & B_{p-1}^U & \\ & & & & A_p & & & & & D_p^L \\ \hline B_1^L & D_2^U & & & & C_1 & & & & \\ & B_2^L & \ddots & & & & C_2 & & & \\ & & \ddots & \ddots & & & & \ddots & & \\ & & & B_{p-1}^L & D_p^U & & & & C_{p-1} & \end{array}\right) \tag{4.14}$$

This system can schematically be written as

$$\begin{bmatrix} A & B^U \\ B^L & C \end{bmatrix} \begin{bmatrix} x' \\ \zeta \end{bmatrix} = \begin{bmatrix} b' \\ \beta \end{bmatrix}. \tag{4.15}$$

The sub-matrices A, B^U, B^L and C are all narrow-banded again.

A notable amount of the computational effort of a classic Gaussian elimination goes into the LU decomposition of the system of equation's matrix. For a large matrix, such as M in this application, this would be a reason not to use the Gaussian elimination. In the modified parallel version however, only the A sub-matrix has to be LU decomposed, which is equivalent to decomposing the individual blocks A_i and can therefore be done in parallel on p processing elements.
Once A is LR-decomposed as $A = LR$, the structure of M enables one to write the LU decomposition of M as

$$M = \begin{bmatrix} A & B^U \\ B^L & C \end{bmatrix} = \underbrace{\begin{bmatrix} L & 0 \\ B^L R^{-1} & E_k \end{bmatrix}}_{L_M} \underbrace{\begin{bmatrix} R & L^{-1}B^U \\ 0 & S \end{bmatrix}}_{R_M} \tag{4.16}$$

where

$$S = C - B^L A^{-1} B^U. \tag{4.17}$$

S is also called the *Schur complement* of the block A in the matrix M.
The next step is also very similar to a classic Gauss algorithm, but not quite the same: inserting the LR-decomposition into the original system of equations leads to

$$\begin{aligned} L_M R_M x &= b && (4.18)\\ \Leftrightarrow \quad R_M x = L_M^{-1} b &=: c && (4.19) \end{aligned}$$

In matrix notation, this modified forward elimination step has the following form:

$$\begin{bmatrix} R & L^{-1}B^U \\ 0 & S \end{bmatrix} \begin{bmatrix} x' \\ \zeta \end{bmatrix} = \begin{bmatrix} L & 0 \\ B^L R^{-1} & I \end{bmatrix}^{-1} \begin{bmatrix} b' \\ \beta \end{bmatrix} =: \begin{bmatrix} c' \\ \gamma \end{bmatrix} \tag{4.20}$$

with

$$c_i' = L_i^{-1} b_i' \tag{4.21}$$

and

$$\gamma_i = \beta_i - B_i^L R_i^{-1} c_i' - D_{i+1}^U R_{i+1}^{-1} c_{i+1}'. \tag{4.22}$$

Any potential confusion that might arise at this point is probably due to the different notations: in (4.20) the sub-matrix notation is used, while in equations (4.21) and (4.22), the calculations on the particular processing element are given.

Now, the reduced system

$$S\zeta = \gamma \tag{4.23}$$

is solved. With a dimension of $(p-1)k \times (p-1)k$ it is smaller than the original system. It has a banded block structure with $k \times k$ blocks. The structure is shown in fig. 4.1. T_i, U_i and V_i are again sub-matrices and can be computed nearly

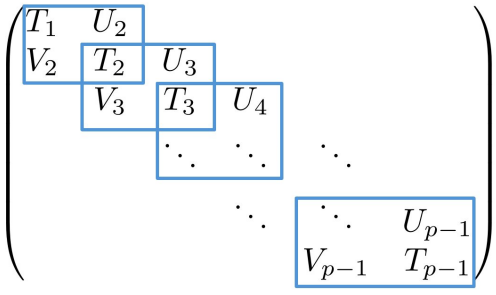


Figure 4.1: Structure of reduced system's matrix S and its decomposition onto processing elements (from [41]).

independently. The overlapping structures seen in Fig. 4.1 mean that some minor inter-processor communication is necessary during the computation of the reduced system.
As shown in Eq. (4.17), S is computed on processing element-level the following

way:

$$T_i = C_i - (B_i^L R_i^{-1})(L_i^{-1} B_i^U) - (D_{i+1}^U R_{i+1}^{-1})(L_{i+1}^{-1} D_{i+1}^L), \quad (4.24)$$
$$U_i = -(D_i^U R_i^{-1})(L_i^{-1} B_i^U), \quad (4.25)$$
$$V_i = -(B_i^L R_i^{-1})(L_i^{-1} D_i^L). \quad (4.26)$$

S itself is again diagonally dominant (see, e.g., [25]).
The reduced system can be solved by any suitable solver. Depending on the system's size, this might be an iterative or direct solver with a good parallel implementation. We will return to this topic in further detail shortly.

When ζ is known, the remainder of the solution x' is calculated via the back-substitution formula that can be deduced from Eq. (4.20). Depending on the processing element i it yields

$$x'_1 = R_1^{-1}(c_1 - L_1^{-1} B_1^U \zeta_1), \quad (4.27)$$
$$x'_i = R_i^{-1}(c_i - L_i^{-1} D_i^L \zeta_{i-1} - L_i^{-1} B_i^U \zeta_i) \quad \forall 1 < i < p, \quad (4.28)$$
$$x'_p = R_p^{-1}(c_p - L_p^{-1} D_p^L \zeta_{p-1}). \quad (4.29)$$

The complete solution x then is composed from the two solution vectors x' and ζ. As a parallel version of the classic Gaussian elimination, the modified algorithm is also backwards stable (see Def. 4.1.4 in this work, [3]).

In our application, where a similar system of equations $Mx^j = b^j$ has to be solved in each iteration step, factorizing the matrix and computing the reduced matrix S only has to be done in the first iteration step. The subsequent iteration steps only include calculating the adapted RHS vector c^j, solving the reduced system $S\zeta^j = \gamma^j$ and doing the back-substitution.
Since S and the sub-matrices have to be saved for subsequent iterations, the memory requirement doubles in comparison to the classical, serial Gauss elimination, though memory demand can be reduced by overwriting sub-matrices that are no longer used, e.g. A can hold L and R, and later L^{-1} and R^{-1}, since the inverse of a (unit) upper resp. lower triangular matrix is (unit) upper/lower triangular (see [38]).

Solving the Reduced System Before presenting several ways to solve the reduced system, it lends itself to briefly discuss the existence of a solution to the reduced system. To transform the original problem to its current form, Eq. (4.20), only equivalent transformations and a partial LR-factorization are used. Therefore, a unique solution to the reduced system exists if the original system has a unique solution.
In comparison to the original $n \times n$ system, the reduced $(p-1)k \times (p-1)k$ system is not a large system anymore. Still, it is large enough to consider solving it in parallel. Depending on its actual size, different kinds of solvers might be of advantage.

An elegant way to solve the reduced system is the cyclic reduction approach. Comparing Eq. (4.10) to Eq. (4.23) yields the insight that M and S have a very similar structure: they are narrow-banded and diagonally dominant. In the cyclic reduction approach, the modified Gauss elimination is applied to the reduced system repeatedly, until the resulting reduced system can be solved comfortably on a single processing element. Afterwards, there are several nested back-substitution steps to compute x. This approach seems to be elegant, but unrewarding, since the degree of parallelism halves with every recursion.
Alternatively, for reduced systems that still are of considerable size, applying an iterative solver again is a promising approach. Chapter 2 gives examples of them. The solvers mentioned there are already implemented in the PHOENIX/3D code. For smaller reduced systems, a direct parallel solver can be effective. Examples for this type of solver are the Krylov-subspace methods, that will be introduced later in this chapter.
Chapter 5 will feature several tests to determine the usefulness of iterative and direct solvers to solve the reduced system depending on the size of S.

Adaption to General Narrow-Banded Systems of Equations Obtaining a stable LU decomposition is a crucial part of the modified parallel Gaussian elimination. In general cases, where a stable LU decomposition does not necessarily exist, pivoting is applied. This way, the existence of the LU decomposition can be assured and possible truncation errors are minimized. This section gives a short overview on a version of the algorithm with partial pivoting that can be applied to general narrow-banded systems. For further details, see [10].

Although the form of M is still assumed to be narrow-banded, the notation differentiates from the diagonally dominant case. The matrix is now made up of the following blocks:

$$M = \begin{pmatrix} A_1 & & & & & D_1 \\ B_1 & C_1 & & & & \\ & D_2 & A_2 & & & \\ & & \ddots & \ddots & & \\ & & & D_p & A_p & \\ & & & & B_p & C_p \end{pmatrix}, \tag{4.30}$$

where the A_i are not square anymore, but $A_i \in \mathbb{R}^{m \times n'}$ with $n' = m - k$ to hold the entries above the main-diagonal. C is still a $k \times k$ matrix. x and b are still denoted as:

$$x = \begin{pmatrix} x'_1 \\ \zeta_1 \\ x'_2 \\ \vdots \\ \zeta_{p-1} \\ x'_p \end{pmatrix}, b = \begin{pmatrix} b'_1 \\ \beta_1 \\ b'_2 \\ \vdots \\ \beta_{p-1} \\ b'_p \end{pmatrix}. \tag{4.31}$$

As explained in the beginning of this chapter, a LU decomposition with partial

pivoting is a factorization of the form $PM = LU$, where P is a row permutation matrix. This decomposition cannot happen immediately, since the form of the problem does not allow for the pivoted factorization to take place on several processing elements independently. Therefore, the system is re-ordered to not hold any blocks in the same column. This leaves the system in the following form:

$$\left(\begin{array}{ccccc|ccccc} A_1 & & & & & & & & & D_1 \\ B_1 & & & & & C_1 & & & & \\ & A_2 & & & & D_2 & & & & \\ & B_2 & & & & & C_2 & & & \\ & & \ddots & & & & \ddots & & & \\ & & \ddots & & & & & \ddots & & \\ & & & A_p & & & & D_p & & \\ & & & B_p & & & & & & C_p \end{array}\right) \left(\begin{array}{c} x_1 \\ \vdots \\ x_p \\ \zeta_1 \\ \vdots \\ \zeta_p \end{array}\right) = \left(\begin{array}{c} b'_1 \\ \beta_1 \\ b'_2 \\ \vdots \\ \beta_{p-1} \\ b'_p \\ \beta_p \end{array}\right) \tag{4.32}$$

Now, a LR-decomposition with partial pivoting can be performed independently on the left-hand sub-matrix block-wise:

$$P_i \left(\begin{array}{c} A_i \\ B_i \end{array}\right) = L_i \left(\begin{array}{c} R_i \\ 0_{k\times n} \end{array}\right) \quad \forall \quad 1 \leq i \leq p. \tag{4.33}$$

Then, $PM = L_M R_M$ is re-written as $L_M^{-1} PM = R_M$. Inserting the conversion into the system of equations leads to

$$L_M^{-1} PMx = R_M x = L_M^{-1} b, \tag{4.34}$$

whereby R_M is found to have the following structure

$$R_M = P \left(\begin{array}{c|c} R & X \\ \hline 0 & S \end{array}\right), \tag{4.35}$$

with X containing products of the sub-matrices $L^{-1}P$ with C and D. To complete the forward elimination step, the reduced system

$$S\zeta = \gamma \tag{4.36}$$

is solved and the full result is then gained by back-substitution. As in the algorithm for diagonally dominant systems, S has the same structure as M.

To summarize, an adaption of the modified parallel Gaussian elimination to systems that are not necessarily diagonally dominant is available. Although this broadens the area of application, it also increases the computational effort. Also, when applied to diagonally dominant matrices, the adapted algorithm does not yield the same reduced system as the first algorithm does. Consequently, when a system is known to be diagonally dominant, this property should be exploited by using the first version of the algorithm.

4.2 Krylov-Subspace Methods

Another class of direct solvers is the class of Krylov-subspace methods. They are solvers for large sparse systems of equations and can also be applied to eigenvalue problems. Among them are solvers like the conjugate gradient (CG) method, as well as the related BiCG and BiCGSTAB methods, generalized minimal residual (GMRES) methods and quasi minimal residual (QMR)-type methods.
This section begins with a short review on the basic concepts of the class of Krylov-subspace methods, before presenting in more detail the GMRES solver. The next section then discusses the use of the GMRES method as internal solver for the modified Gauss method.

4.2.1 The Basic Concepts of Krylov Solvers

Krylov-subspace methods are *projection methods* using *Krylov subspaces* to refine the approximate solution of a problem of the form $Mx = b$.
For the remainder of this chapter, lower indices will denote the related Krylov subspace.

Definition 4.2.1. *A projection method for the solution of $Mx = b$ is a method that computes approximate solutions $x_t \in x_0 + K_t$ with regard to*

$$(b - Mx_t) \perp L_t, \tag{4.37}$$

for any $x_0 \in \mathbb{R}^n$ and K_t and L_t are t-dimensional sub-spaces of $\mathbb{R}^n$. The orthogonality condition is defined via the euclidean scalar product through

$$x \perp y \Leftrightarrow (x, y)_2 = 0. \tag{4.38}$$

(see [26], Definition 4.58)

A Krylov-subspace method can now be defined as follows:

Definition 4.2.2. *A Krylov-subspace method is a projection method designed to solve $Mx = b$, where the Krylov subspace K_t is defined as*

$$K_t = K_t(M, r_0) = \text{span}\{r_0, Mr_0, M^2r_0, \ldots, M^{t-1}r_0\}, \tag{4.39}$$

where $r_0 = b - Mx_0$.
(see [26], Definition 4.60)

Therefore, the approximate solution in stage t of the Krylov solver has the form

$$x_t = x_0 + q_{t-1}(M)r_0 \quad \in K_t(M, r_0), \tag{4.40}$$

where q_{t-1} is a polynomial of degree $t-1$. Accordingly, the residuals can be written as

$$r_t = p_t(M)r_0 \quad \in r_0 + MK_t(A, r_0) \quad \in K_{t+1}(M, r_0) \tag{4.41}$$

with p_t a polynomial of degree t. Implementations of Krylov solvers often calculate the update to the residual indirectly and only once update the solution x in the end or when the residual is smaller than the intended accuracy.
Since Krylov-subspace methods are direct solvers, they should yield the exact solution to the system of equations (4.1) apart from rounding errors. The following theorem elaborates on this:

Theorem 4.2.3. *Be $M \in \mathbb{R}^{n\times n}$ a regular matrix. Then, a Krylov-subspace method will yield the exact solution to the system of equations $Mx = b$ after at most n refinement steps.*

Proof. See [26], Theorem 4.62 □

With enough time and computational effort, Krylov-subspace methods obtain an exact solution. In some applications although, computing all the refinements in ever growing subspaces will exceed appropriate computational costs. Therefore, the actually direct Krylov methods sometimes are used as iterative methods, where the computation is aborted after a number of refinement steps and an approximate solution is accepted.

4.2.2 The GMRES Algorithm

One member of the class of Krylov solvers is the GMRES method, where GMRES stands for *Generalized Minimal Residual*. This name stems from the fact, that, while still a Krylov method, the GMRES approach can also be interpreted as minimizing the residual via the function

$$\begin{aligned} F: \quad \mathbb{R}^n &\to \mathbb{R} \\ x &\to ||b - Mx||_2^2. \end{aligned} \tag{4.42}$$

This is a valid approach, as the following theorem shows:

Theorem 4.2.4. *Be $A \in \mathbb{R}^{n\times n}$ a regular matrix and $b \in \mathbb{R}^n$. Then*

$$\hat{x} = \mathbf{arg}\min_{x\in\mathbb{R}^n} F(x) \tag{4.43}$$

if, and only if,

$$\hat{x} = M^{-1}b \tag{4.44}$$

Proof. See [26], Theorem 4.86 □

Therefore, the GMRES algorithm is suitable for solving general systems of equations $Mx = b$, where M is a regular matrix.
Alternatively, the GMRES algorithm can be seen as Krylov-subspace method where the subspaces L_t are defined as

$$L_t = AK_t. \tag{4.45}$$

Like all methods of the Krylov-subspace class, the GMRES method can be used as a direct method, the classic GMRES, as well as an iterative method, called the restarted GMRES. As a direct method, GMRES gives the exact solution after $j \leq n$ iterations. It is possible though, that a broad number of iterations is necessary, so memory requirements are high to a point of not being applicable. As an iterative method, however, the iterative or "restarted" GMRES algorithm is used repeatedly, so the solution is gradually refined and memory can be cleared after each set of iterations. This method results in an approximate solution.
Implementing the GMRES algorithm includes, among others, the computation of bases of the Krylov subspaces L_t resp. K_t, as well as matrix-vector multiplications. These calculations can be parallelized effectively, especially for sparse matrices M, and therefore provide a parallel solver for linear systems of equations.

4.2.3 Use in PHOENIX/3D

Currently, a Krylov-type method is employed in PHOENIX/3D as part of the modified Gauss MPI implementation , namely a solver for the occurring sparse reduced system. At this point in the modified Gauss algorithm, the user can choose a solver for the reduced system suitable to their specific problem: a direct LR solver is implemented , as well as an iterative Jacobi solver and the aforementioned restarted GMRES method. All three are suitable to solve sparse systems of equations, but depending on the exact parameters and problem size, the efficiency of the inner solvers can differ. Chapter 5 includes tests on their specific behaviors. In this context, the Gaussian elimination could mathematically be interpreted as a pre-conditioner for the Krylov-subspace method.
Alternatively, the GMRES algorithm could be employed to solve the system of equations in the Operator Splitting step of PHOENIX/3D without employing the modified Gauss method first. This is possible since the approximate Lambda approach results in a sparse system of equations, Eq. (3.23), that has to be solved in every iteration step. Still, Krylov-subspace methods might only be suitable up to a specific problem size and are therefore only implemented in the context of the Modified Gauss algorithm, since the reduced system is typically significantly smaller than the original sparse linear system.
As a general remark, Krylov-subspace methods might not suitable as an alternative to the operator splitting step itself, since they are most effective on sparse systems of equations, while Eq. (3.19) might be a dense system.

4.3 Implementation of the Modified Gauss Algorithm

The implementation of the modified parallel Gaussian elimination takes into account that there is a similar system of equations $Mx^j = b^j$ that has to be solved in each iteration step. Therefore, factorizing the matrix M and computing the reduced system S only happens once during the first iteration step, while in all subsequent iterations only the RHS is updated, the reduced system is solved and

the back-substitution is completed.
Since the memory requirements of the parallel algorithm exceed those of the serial version through the need to also save sub-matrices, memory is handled carefully, e.g. by using memory-saving formats for sparse matrices and by overwriting sub-matrices that are no longer needed. Care has to be taken, though, because some of the sub-matrices are needed in every iteration step to modify the current RHS b^j.

Currently, there are two implementations of the modified parallel Gauss solver: one C/OpenCL stand-alone version and a Fortran/MPI version that is already integrated in PHOENIX/3D.
The reason behind having two different implementations of the same solver is connected to the machine architecture one intends to use it on: the OpenCL version runs well on GPUs, CPUs, accelerator cards and combinations thereof, while the MPI version is very suitable for many-core and many-node architectures (see Chapter 2).

General Remarks on Implementation The modified Gauss algorithm can be coarsely divided into four main tasks: the factorization step to compute a reduced system matrix S only has to take place in the first iteration and is then re-used in every succeeding iteration. After the factorization is complete, a typical iteration step only includes the computation of the adjusted RHS, solving the now complete reduced system consisting of S and the new RHS. The last task is a back-substitution step, where the complete result is computed from the result of the reduced system.
However, before the actual algorithm-related tasks can be started, the data has to be brought into another format. PHOENIX/3D saves variables in the so-called *pve grid*. In this context, the approximate Lambda operator `Lstar` is only one entry in the `pve_grid`. Its structure has the following form

```
pve_grid%Lambda(ix, iy, iz)%Lstar(ixn, iyn, izn),
```

where `ix, iy, iz` stand for the grid point and `ixn, iyn, izn` describe a neighbor in the grid. Those neighbors are denoted with $ixn, iyn, izn \in \{-1, 0, 1\}$ and sum up to 27 neighbors for each grid point. In matrix form, this corresponds to `pve_grid%Lambda(ix,iy,in)%Lstar(0, 0, 0)` holding the diagonal entries, while for other values of `ixn, iyn, izn` stand for entries in the same matrix line. To use the modified Gauss algorithm on the systems of equations from PHOENIX/3D, a conversion into matrix format has to take place. Since the scheme uses sub-matrices, it is convenient to save the `Lstar` entries to the appropriate sub-matrices immediately.
It is possible, that the initial combination of values for the variables p, k and m, does not agree with Eq. (4.12). In that case, the size of the sub-matrices is augmented until the smallest consensus between variables is reached. Redundant entries are filled with zeros. This way, the original solution is preserved while the additional computational effort is small.
With the data now in the correct form, the factorization step can start. PHOE-

NIX/3D is equipped with the `new_Lstar` variable, that indicates whether the factorization step is necessary, since there is a newly setup `Lstar` , or if it already was factorized in a previous iteration step.
In case, the factorization step is necessary, it begins with a LR-decomposition of the sub-matrices of type A_i (see Eq. 4.17). Since the original matrix is assumed to be diagonally dominant, no pivoting is necessary during the LU decomposition. The algorithm implemented in both codes stems from [38].
To evenly distribute work between processing elements, the requirement $mp+(p-1)k = n$ holds. Therefore, in some cases, the choice of p is inconsistent with the problem size n. In these cases, the problem size n is adapted, i.e. slightly increased, until m and k can be calculated while still holding the above requirement. The expansion of the problem size is done in a way, that adds lines containing a $E_{n'-n}x = 1$ to the actual problem, where n' is the adapted problem size. To do this, the sub-matrices A and C are initialized as identity matrix $E_{n'}$ and the b, resp. c vectors containing ones. During the algorithm, the part of the original problem's size is overwritten, while any additional lines do not influence the original solution. Although this slightly increases the problem size, it keeps the work balance among processing elements even. Alternatively, [10] proposes an approach with variably-sized sub-matrices and vectors.
How the remainder of tasks is implemented in detail differs depending on the programming language used. Both approaches are described in the following sections.

Details on the OpenCL Implementation The OpenCL implementation of the modified Gauss algorithm is a stand-alone code, but could be incorporated into PHOENIX/3D with some adjustments. PHOENIX/3D provides an OpenCL version, whose infrastructure could be used regarding OpenCL contexts and command queues. To complete the integration into PHOENIX/3D, an additional coordinate transfer kernel would be necessary to transform the `Lstar` format into one accessible to the stand-alone code and vice versa to reconstruct the result after computing an iteration step.
The general structure of the OpenCL implementation follows the separation of tasks as explained above: four main kernels hold the parts of the algorithm that can be executed in parallel. One kernel holds the factorization step, in which the reduced system's matrix S is calculated. From then on, another kernel adjusts the RHS and therefore completes the reduced system. Thereafter, a parallel Jacobi solver calculates the reduced system's result and the fourth main kernel does the back-substitution step. Currently, the implementation only supports solving one system of equations during execution. In future versions, an iteration counter will make sure that the factorization only happens in the first iteration, while the other kernels are invoked in every iteration. There are also some kernels holding debugging routines. The host code sets up the necessary memory objects, loads them to the kernels and enqueues the kernels in the correct order to the command queue. Once the memory objects have been pushed to the device, they stay there and can be manipulated by other kernels in the same command queue, before being read back by the host via queuing a read-command. Since the memory objects are defined on global memory space, communication between different work-items in

a work-group happens through writing into or reading from each other's memory. Afterwards, the memory objects are synchronized through a barrier function to ensure matching variables on all work-items. Auxiliary functions are hold by a C code file, which can be used by the work-items evoked.
One particularity of the OpenCL implementation of the modified Gauss algorithm is the reduced system's matrix S being saved in the *compressed row storage* (CRS) format. Instead of saving a sparse matrix, in the CRS format three vectors hold all information necessary: the non-zero matrix entries are saved in one vector, while two other vectors hold the respective column index and row pointer to reconstruct the entries position in the original matrix. This way of saving S is advisable for the OpenCL implementation, since in OpenCL there is currently no matrix format, and pointers to pointers are not feasible as kernel arguments. All matrix capacities, among them S and the sub-matrices, therefore have to be transformed to vectors before being passed to the kernels. The CRS format might offer an additional saving in terms of memory usage. The sub-matrices are saved as normal vectors, since they are not necessarily sparse.
Because of the necessity to save matrices as vectors, several coordinate transforms take place: from the original matrix to vectors and then to sub-matrices in vector form, before being used to form the reduced system. Furthermore, the transformations to vectors make it necessary to write special routines for matrix-matrix and matrix-vector multiplication. They are callable by work-items with a specific part of the matrix in regard and therefore enable parallel multiplications. The parallel Jacobi solver, which is implemented as internal solver, also is designed to heed the CRS format of S.
Timing information is provided via the C wall clock function for the total execution time. OpenCL timing functions can provide additional information on the execution times of separate kernels through the OpenCL event profiling.

Details on the Fortran/MPI Implementation The Fortran/MPI version of the modified parallel Gauss algorithm is fully integrated into PHOENIX/3D. It is part of the solver routines in `OS_iteration.for` and can be selected through the `solver = 7` option. PHOENIX/3D then calls this solver routine in every iteration step. Therefore, it is important to distinguish between the iteration steps where the factorization has yet to take place and iteration steps where the reduced system's matrix S is already determined. This is done via the `new_Lstar` variable. Once the reduced system matrix is calculated, it is available during all following iteration steps through the `save` attribute, as well as some of the sub-matrices necessary to compute the adjusted RHS in every iteration step.
As mentioned above, it is necessary to reorder the entries of `Lstar` in order to distribute them to their correct sub-matrix. To tackle this problem, a coordinates transform function was implemented to transform the entries of the `pve_grid%Lambda%Lstar` to entries in the respective sub-matrix. This functionality includes the calculation of the correct sub-matrix and its local coordinate in it from a `Lstar` entry. The same function is used to estimate the half-bandwidth k of the imagined full system's matrix. This is necessary since the entries for the 27 neighboring values typically lead to a bandwidth broader than 27 as described in

Ch. 3.3.
The following code excerpt shows the point where the `Lstar` entries are identified regarding their appropriate sub-matrix type and their corresponding processing element by the `typ_matrix` function, and then are distributed there. Boundary conditions are satisfied through adaptive `do`-loops and an accordingly adapted bandwidth parameter k.

Listing 4.1: Extract from Fortran/MPI code showing distribution from PHOENIX Lstar variable to sub-matrices

```
        do izh=-nz,nz
         do iyh=-ny,ny
          do ixh=-nx,nx
          do izL=-nnlz,nnlz
           iz = izh+izL
           if(.not. z_pbc .and. abs(iz) .gt. nz) cycle
           if(z_pbc) then
            if(iz .gt. nz) iz = -nz
            if(iz .lt. -nz) iz = nz
           endif
           do iyL=-nnly,nnly
            iy = iyh+iyL
            if(.not. y_pbc .and. abs(iy) .gt. ny) cycle
            if(y_pbc) then
             if(iy .gt. ny) iy = -ny
             if(iy .lt. -ny) iy = ny
            endif
            do ixL=-nnlx,nnlx
             ix = ixh+ixL
             if(.not. x_pbc .and. abs(ix) .gt. nx) cycle
             if(x_pbc) then
              if(ix .gt. nx) ix = -nx
              if(ix .lt. -nx) ix = nx
             endif
c
c--
c-- loop body for pve_grid%Lambda(ix,iy,iz)%Lstar(-ixL,-iyL,-izL)
c-- Lstar holds (1-eps)Lstar; submatrices will hold En-(1-eps)
    Lstar!!
c--

             call coords_transf_phx(nx, ny, nz, ix, iy, iz, ixh,
                 iyh,
     & izh, i, j)

             call typ_matrix(m, k, i, j, typ, block, local_i,
                 local_j)

             if(local_i.LT.1.OR.local_j.LT.1) CYCLE

             if(typ==0) then
               if(local_i.GT.m.OR.local_j.GT.m) CYCLE
               R(local_i, local_j, block) =
     & - pve_grid%Lambda(ix,iy,iz)%Lstar(-ixL,-iyL,-izL)
             if(ixL .eq. 0 .and. iyL .eq. 0 .and. izL .eq. 0)
```

```
              then
             R(local_i, local_j, block)= R(local_i, local_j,
                 block)+1
            end if

          else if (typ==1) then
            if(local_i.GT.m.OR.local_j.GT.k) CYCLE
            LB(local_i, local_j, block) =
     & - pve_grid%Lambda(ix,iy,iz)%Lstar(-ixL,-iyL,-izL)
            if(ixL .eq. 0 .and. iyL .eq. 0 .and. izL .eq. 0)
                then
             LB(local_i,local_j,block)= LB(local_i,local_j,
                 block)+1
            end if

          else if (typ==2) then
            if(local_i.GT.m.OR.local_j.GT.k) CYCLE
            LD(local_i, local_j, block)=
     & - pve_grid%Lambda(ix,iy,iz)%Lstar(-ixL,-iyL,-izL)
            if(ixL .eq. 0 .and. iyL .eq. 0 .and. izL .eq. 0)
                then
             LD(local_i,local_j,block)= LD(local_i,local_j,
                 block)+1
            end if

          else if (typ==3) then
            if(local_i.GT.k.OR.local_j.GT.k) CYCLE
           Cmat(local_i, local_j, block)=
     & - pve_grid%Lambda(ix,iy,iz)%Lstar(-ixL,-iyL,-izL)
           if(ixL .eq. 0 .and. iyL .eq. 0 .and. izL .eq. 0) then
              Cmat(local_i,local_j,block)=Cmat(local_i,local_j,
                 block)+1
            end if

          else if (typ==4) then
            if(local_i.GT.k.OR.local_j.GT.m) CYCLE
            DR(local_i, local_j, block)=
     & - pve_grid%Lambda(ix,iy,iz)%Lstar(-ixL,-iyL,-izL)
            if(ixL .eq. 0 .and. iyL .eq. 0 .and. izL .eq. 0)
                then
             DR(local_i, local_j, block)= DR(local_i,local_j,
                 block)+1
            end if

          else if (typ==5) then
            if(local_i.GT.k.OR.local_j.GT.m) CYCLE
            BR(local_i, local_j, block)=
     & - pve_grid%Lambda(ix,iy,iz)%Lstar(-ixL,-iyL,-izL)
            if(ixL .eq. 0 .and. iyL .eq. 0 .and. izL .eq. 0)
                then
             BR(local_i,local_j,block)= BR(local_i, local_j,
                 block)+1
            end if

          else
            write(*,*) 'Error OS_step_fact: invalid matrix type
                '
```

```
            end if


           enddo
          enddo
         enddo
        enddo
       enddo
      enddo
```

Later in the code, the same coordinates transform function `coords_transf_phx` is used to write the result back to `pve_grid%J`.
As can be seen in the code segment, the sub-matrices are saved as a vector of two-dimensional arrays, ordered through their corresponding processing element and their local coordinates in the sub-matrix, e.g.,

`R(local_coord 1, local_coord 2, processor no.).`

Accordingly, the RHS is saved as a vector of vectors.
Some of the tasks of the modified Gaussian algorithm can be solved by applying LAPACK (*Linear Algebra Package*) or vendor-optimized LAPACK routines, e.g., Intel MKL routines (see [2], [6], and [1]). They are used to inverse the L and R sub-matrices after the LU decomposition and as well for the matrix multiplications. The LU decomposition itself is not a pre-implemented one, because the before-mentioned packages do not provide a LU decomposition function without pivoting. It is, however, possible to solve the reduced system via a direct LAPACK solver, which internally does a LR-decomposition with partial pivoting. For memory purposes, L is initialized as identity matrix, while R holds the entries of A before the decomposition.
The code is parallelized using MPI. PHOENIX/3D offers initialization and finishing routines with several communicators; the communicator used during the iteration is the `MyMPI_3DRT_COMM_FS`, already in use from the computation of the formal solution. At the point of beginning the Operator Splitting iteration, MPI is already set up and ready to be used. The data contained in the sub-matrices is then distributed to the different ranks. At three points of the algorithm communication is necessary in the form of each rank sending and receiving one sub-matrix each. Those three points are during the set-up of S, during the calculation of the adjusted RHS, and once during the back-substitution step. At two additional points, there is the need for collective communication: since the reduced system is currently solved on one process, information from all processes needs to be gathered before solving the reduced system; afterwards, the root process distributes pieces of the solution to the other processes for the back-substitution step. In the end, the complete solution is again gathers on the root processor to be written back into the PHOENIX/3D format.
The timing information for the routine is provided by PHOENIX/3D routines.

Solving the Reduced System The modified Gauss algorithm contains no guidelines on how to solve the reduced system, once it is calculated. Still, de-

pending on the problem, some approaches might be more promising than others. Here, the internal solvers that were implemented into the two different codes are presented, as well as the reasoning behind the selection.

In the serial code version in PHOENIX/3D, one can choose between three internal solvers for the reduced system: a LR decomposition method with pivoting, a Jacobi solver and an implementation of the GMRES method.
The solvers stand as examples for three popular groups of solvers for linear systems of equations: the LR decomposition method is a direct method and implemented using LAPACK functions, while the Jacobi solver is an iterative solver and belongs to the class of operator splitting methods. The original GMRES algorithm is a Krylov method and also a direct method. However, the algorithm here is implemented as a restarted GMRES and therefore belongs to the group of iterative solvers.
A switch in `OS_iteration.for` lets the user decide which internal solver to use in the modified Gauss algorithm. The decision for an internal solver depends mostly on the size of the reduced system: smaller problems can be expected to be solved faster by a direct solver, while an iterative solver might be advantageous for larger reduced systems. In Chapter 5, tests will be conducted to determine whether those expectations turn out correct and which system sizes exactly count as "smaller" and "larger".

The MPI-parallelized version of the modified Gauss algorithm is also already implemented in PHOENIX/3D. It uses the same internal solvers as the serial version, since the MPI code still solves the reduced system on only one processor. This makes sense, as long as the reduced system is small enough. Once the MPI modified Gauss code is applied to larger sets of problems, it might be worth a thought to implement a parallel internal solver as well. On the other hand, solving the reduced system in parallel results in a need for further communication between processes and therefore might only be effective when solving very large problems.

The stand-alone OpenCL version of the modified Gauss solver uses a parallel Jacobi solver to solve the reduced system. In this implementation, the relevant data is already known to the device used for parallel execution, e.g., a GPU. As a consequence, no additional computational effort is necessary for extra communication and a parallel solver makes sense for problems of every size.

Results from Previous Numerical Experiments The authors of [10], where the parallel Gaussian elimination is described, also conducted some numerical experiments in 1999 on an Intel Paragon. They compared a stand-alone version of the algorithm to one implemented with the ScaLapack package (see [6]). However, their implementations did not allow for solving a series of systems of the form $Mx^j = b^j$. Furthermore, the code was not scalable but relied on hard-coded data structures and was not part of a larger application. Among their findings was the observation that the execution time scales with the number of messages sent between processing elements. The algorithms' behavior also depend strongly on

the problem sizes, e.g. the algorithm using pre-implemented ScaLapack methods behaved better on large systems ($n = 100,000$, half-bandwidth $k \approx 100$) than on medium-sized systems ($n = 100,000, k \approx 20$). Furthermore, they found that the execution times of the partial-pivoting algorithm were not as large as expected compared to the non-pivoting algorithm, but mentioned that their implementation of the pivoting algorithm required twice the memory than the non-pivoting version. In their tests, both implementations of the pivoting and non-pivoting solver showed stable behavior.

The next chapter is dedicated to numerical experiments that have been conducted with the two new OpenCL/ MPI versions of the modified Gaussian elimination. These version are capable of calculating subsequent results from a series of systems of equations $Mx^j = b^j$. Furthermore, they are written in two parallel programming languages resp. language augmentations that are suitable for modern machine architectures and are scalable to different problem sizes.

Chapter 5

Tests and Results

An alternative approach at solving the radiative transfer problem was proposed and implemented in two different parallel programming languages. This chapter features test cases that were designed to quantify the behavior of these implementations.
We begin with a short introduction of the machines used to conduct the tests and then proceed with five groups of tests: "Preliminary Tests", that describe behavior independent of the implementation, "Fortran & MPI Version" dedicated to each of the two implementations, "PHOENIX/3D Tests" to compare the properties of the Fortran/MPI-implementation of the modified Gauss solver to the original PHOENIX/3D radiative transfer solver, and tests regarding the stand-alone "C & OpenCL Version".

Machines and Devices Before presenting the tests and their results, this section gives an overview of the hardware the tests are conducted on. Since general concepts of computer architectures were already presented in Ch. 2, we will focus on the machines' specific set-up, including details such as operating system and processors.
On the Linux machine *Minion*, a AMD Radeon R9 290 GPU with the "Hawaii" graphics processor and 4GB memory allows GPGPU-tests with OpenCL code. See fig. 5.2 for more details. The GPU is listed as PCI 1002:67b0, while the gray block describes the CPU configuration, an Intel Core 2 Extreme X9650 CPU.
Igor, also a Linux machine, contains a Nvidia Quadro P6000 GPU with 24 GB memory. It also serves for OpenCL-code testing.
The machine *Calvin* holds an Intel Xeon Phi 7210 CPU "Knights Landing". It has 64 cores and belongs to the second generation of Intel Xeon Phi processors. In contrast to older models, it is available as a main processor, not only as a co-processor. Calvin also runs on Linux. Figure 5.1 shows the machine's hardware topology. The 64 cores of the Knight's Landing and its memory structure can be recognized easily.
Hummel is a Linux-cluster. It consists of 316 standard-nodes with two CPUs of type Intel Xeon E5-2630v3 each, and 54 additional GPU-nodes with Nvidia K80 GPUs, as well as further special nodes.

Both, Calvin and Hummel are used for testing the Fortran/MPI implementation of the modified Gauss solver.

The machines above are chosen to represent a variety of architectures and devices during the tests.

5.1 Tests 0: Preliminary Tests

The Linear System's Sparsity Pattern During application of the operator splitting method, the following system is solved in every iteration step (see Eq. 3.26):

$$[E_n - \Lambda^*(1-\epsilon)]\, J^{(\text{new})} = J^{\text{FS}} - \Lambda^*(1-\epsilon) J^{(\text{old})}. \tag{5.1}$$

Since solving the system takes up computing time and resources, it is vital to understand its properties. One of them is the matrix' $[E_n - \Lambda^*(1-\epsilon)]$ narrow-bandedness, see, e.g., [20].

Figure 5.3 illustrates this property: the sparsity pattern shows the location of entries in the matrix. The data stems from a $nx = ny = nz = 4$ grid with 729 grid points in total. This pattern is no exception from the rule, as one can see in Fig. 5.4, where the sparsity patterns for several other grid configurations are shown.

Apart from the narrow-bandedness, all sparsity patterns show recurring elements, such as the arrangement in three separate diagonal bands and in these bands the clustering of points. These effects are caused by the projection of a three-dimensional spatial grid onto a two-dimensional matrix. Especially the projection of neighboring values in 3D onto the matrix causes a pattern. Furthermore, the structure is influenced by the boundary conditions imposed on the system.

Another interesting fact is shown in Fig. 5.4a: in the case of a nx=ny=nz=1 grid, which is the smallest possible grid size (greater zero) in PHOENIX/3D, the band-pattern also holds. However, the bandwidth is not any more small compared to the matrix size and $2k + 1 << n$ does not hold. Therefore, in cases with small grids, applying the modified Gauss method is not an effective choice, because the problem does not fulfill the narrow-bandedness condition. In those cases, a full matrix solver might be more effective.

Figure 5.5a additionally demonstrates that boundary conditions influence the sparsity pattern of the matrix. The plot shows a case with periodic boundary conditions in x- and y-direction side by side with a case of open boundary conditions. Both patterns stem from a nx=ny=nz=4 Cartesian grid.
As one can see in plot 5.5a, the periodic boundary conditions lie along an additional outer diagonal. Those entries augment the bandwidth in comparison to the case of open boundary conditions. The bandwidth variable k therefore needs to be adapted, so that the sub-matrices are equipped to hold the full band. This is implemented in the routine calculating k and happens if periodic boundary con-

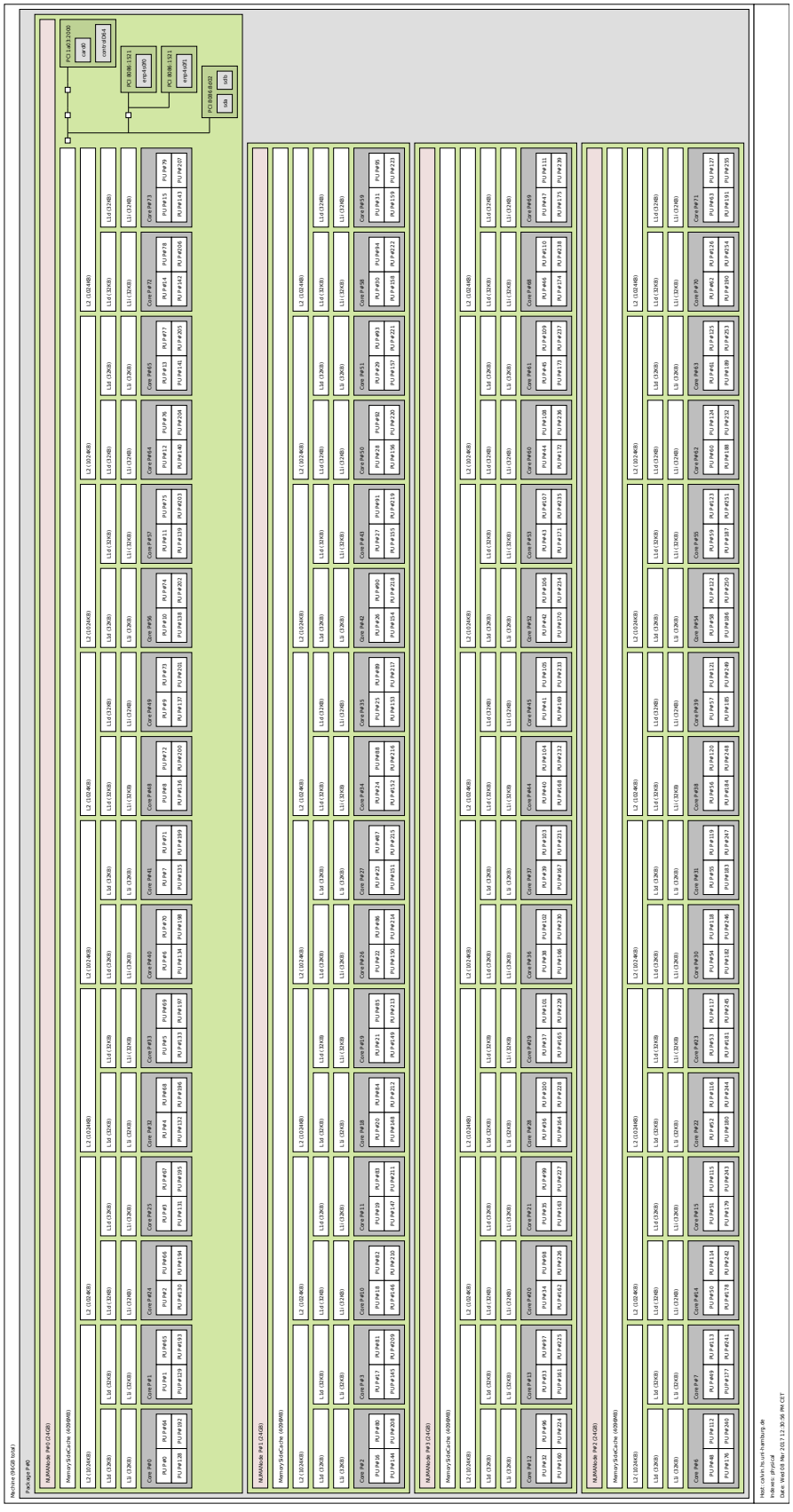


Figure 5.1: Topology map of *calvin*

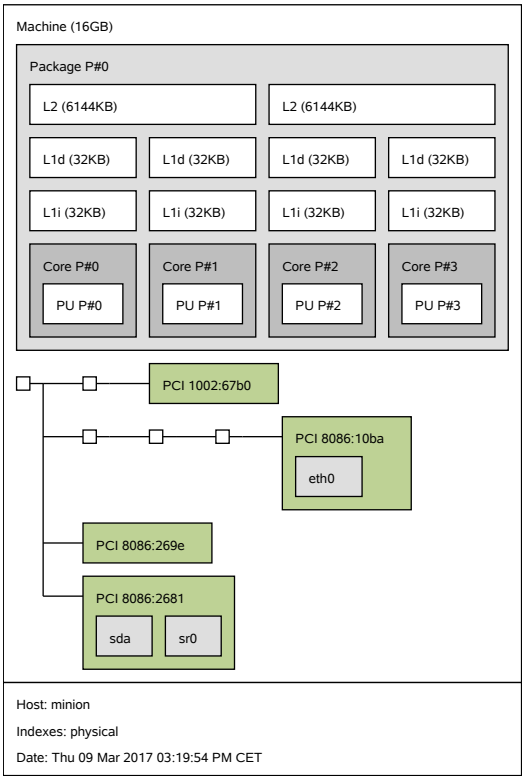


Figure 5.2: Topology map of *minion*

ditions are requested by the user.

5.2 Tests I: Fortran & MPI Version

In this section, the Fortran implementation of the modified Gauss algorithm is tested. The code is parallelized with MPI and integrated into the PHOENIX/3D radiative transfer code.

Compliance of Results The first group of tests in this section is dedicated to show compliance of the modified Gauss solver's results with the original PHOENIX/3D solvers.
There are non-trivial analytic solutions to the 1D radiative transfer problem. For a discussion of those, see, e.g., [22]. A comparison to an analytic solution has, however, not been performed here, since PHOENIX/3D already shows good agreement with 1D radiative transfer problems (see [18]), which, in turn, comply with analytic solutions.

First, the results of the modified PHOENIX/3D version are compared to results obtained with the original PHOENIX/3D code. The data stems from tests on calvin on a $nx = ny = 4, nz = 16$, $(4, 4, 16)$, and a $nx = ny = nz = 4$, $(4, 4, 4)$, grid.

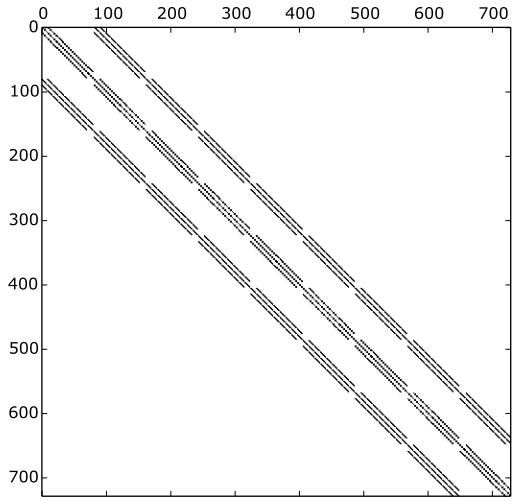
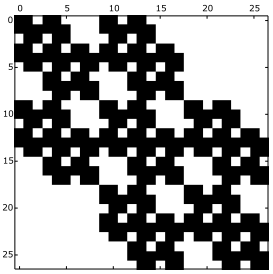


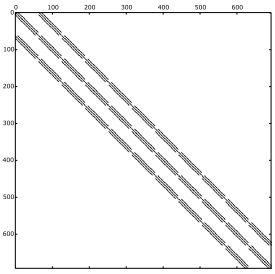
Figure 5.3: Sparsity pattern of the radiative transfer matrix

The stopping criterion was set to $stop = 10^{-6}$. Whether the criterion is fulfilled is computed through the change of results between iteration steps: increasingly small changes in the solution indicate that a converged result is reached.
While the PHOENIX/3D code is run in parallel, at first the serial radiative transfer solvers are compared. Since the modified Gauss solver is not designed to be run serially, the code is modified to execute all computations, that would normally happen in parallel, successively on one processing element only. Later tests will then feature a comparison of results for a parallel Jacobi solver and the parallel modified Gauss method.
Measuring the differences in solution between solvers is done through the two-norm (euclidean norm) of the relative error between the solutions, $\frac{\Delta J}{J}$, and the entry-wise maximum error. The relevant solvers here are the serial Jacobi, Diagonal and Gauss-Seidel solvers, that were incorporated into the original PHOENIX/3D code, and the modified Gauss solver, whose implementation into PHOENIX/3D is part of this work. Since the modified Gauss solver relies on an internal solver for the solution of the reduced system, three versions of the modified Gauss solver were implemented. All solvers are tested against the widely-used Jacobi solver from the original PHOENIX/3D version. The parallel tests compare a MPI implementation of the Jacobi solver with the parallel modified Gauss solver.

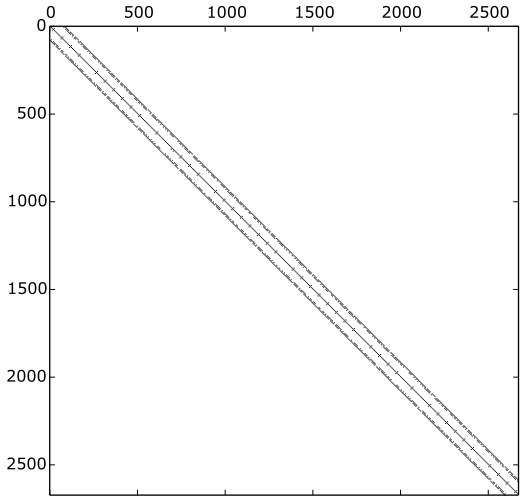
Tables 5.1 and 5.2 hold the results of the serial compliance tests. It shows that the difference between the results obtained with the modified Gauss solver, regardless of the internal solver, and the solvers originally incorporated into PHOENIX/3D lie in comparable orders of magnitude as the differences between the PHOENIX/3D original solvers, the Jacobi, the diagonal and the Gauss-Seidel method. In a numerical sense, the solutions sufficiently agree and the 3D radiative transfer result is indeed independent of the use of solver.



(a) nx=ny=nz=1 grid with 27 grid points

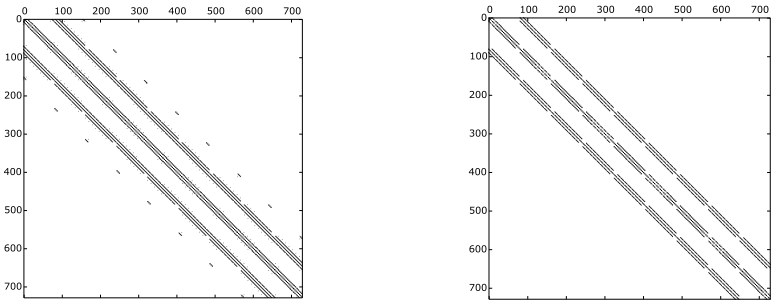


(b) nx= 3, ny= 4, nz=5 grid with 693 grid points in total



(c) nx= ny= 4, nz= 16 grid with 2673 grid points

Figure 5.4: Sparsity patterns for several grid sizes



(a) Periodic boundary conditions (b) Fixed boundary conditions

Figure 5.5: Comparison of sparsity patterns with different boundary conditions

	rel. error	max. error
Jacobi	0.0	0.0
Gauss-Seidel	9.14191e-14	2.37521e-07
Diagonal	6.43609e-14	2.01811e-07
Mod. Gauss (LAPACK)	6.27493e-14	9.25289e-07
Mod. Gauss (Jacobi)	5.61511e-14	9.25289e-07
Mod. Gauss (GMRES)	6.72664e-14	9.25264e-07

Table 5.1: Compliance of **serial** results compared to PHOENIX/3D Jacobi solver on a $nx = ny = nz = 4$ grid

Compliance MPI Version Naturally, the compliance tests are also conducted in parallel with the MPI version of the modified Gauss solver and a parallel implementation of the Jacobi solver, that is the only other parallel solver currently implemented in PHOENIX/3D. Results are given in Table 5.3.

The table shows that the result of the modified Gauss solver agree exactly, regardless of whether it is run serially or in parallel. The difference between the parallel modified Gauss solver and the parallel Jacobi solver is in the magnitude of 10^{-15} and the max. error lies in the 10^{-8} magnitude.
Consistently, on a $nx = ny = 4, nz = 16$ grid, the relative error between the MPI versions of modified Gauss and the Jacobi solver is 5.54116e-14 and the max. error is 6.14228e-07.
Thus, the results of the parallel modified Gauss solver implemented with MPI also comply with the former radiative transfer code's results. The modified parallel Gauss solver therefore runs correctly.

Internal Solvers When implementing the modified Gauss solver, one has to decide how to solve the emerging reduced system. Three different internal solvers

	rel. error	max. error
Jacobi	0.0	0.0
Gauss-Seidel	1.7286e-12	1.72307e-05
Diagonal	9.08072e-13	8.77202e-07
Mod. Gauss (LAPACK)	1.10335e-12	1.08188e-06
Mod. Gauss (Jacobi)	1.12525e-12	1.08188e-06
Mod. Gauss (GMRES)	1.09278e-12	1.08188e-06

Table 5.2: Compliance of **serial** results compared to PHOENIX/3D Jacobi solver on a $nx = ny = 4, nz = 16$ grid

	rel. error	max. error
Mod. Gauss (MPI)	0.0	0.0
Jacobi (MPI)	9.50167e-15	7.26925e-08
Mod. Gauss (serial)	0.0	0.0

Table 5.3: Compliance of **parallel** results on $p = 3$ processes compared to parallel mod. Gauss solver on a $nx = ny = nz = 4$ grid

were implemented: a direct solver from the LAPACK package, a Jacobi solver and a restarted GMRES method. While the LAPACK-algorithm uses a direct method, the Jacobi and restarted GMRES solvers are iterative methods. All three were implemented as serial algorithms. This test examines the internal solvers' behavior depending on the grid size of the overall problem.

The internal solvers' run-times are recorded during 3D radiative transfer runs for multiple grid sizes from $nx = ny = nz = 1$ to $nx = ny = nz = 12$. Grid sizes above that are not tested due to the Gauss method's memory requirements. The 3D radiative transfer MPI runs are conducted on Calvin with $p = 16$, although the Operator Splitting steps are executed serially simulating a $p = 2$ run on one processor. Other relevant variables are $\epsilon = 10^{-1}$ and the stopping criterion $stop = 10^{-6}$.
During a 3D radiative transfer calculation, the internal solvers are called several times, once in every OS-iteration. The times then are averaged over all iterations, since the number of Os-iterations is constant for the three internal solvers. Figure 5.6 presents the results in the upper plot. The lower plot shows the increase in the size of the reduced system. Note, that the $nx = ny = nz = 4$ and the $nx = ny = 4, nz = 16$ grids both result in the same reduced system size. This is due to the fact that the size of the reduced system is $(p-1)k \times (p-1)k$ (see Ch. 4). In some cases this may lead to reduced systems of the same size, while the original problem sizes differ. Figure 5.7 shows how the sizes of the reduced system's matrices develop in comparison to the original problem sizes.

Figure 5.6 clearly shows that for the tested grid sizes, the LAPACK solver always performed the fastest runs. As expected, run-times increase gradually with

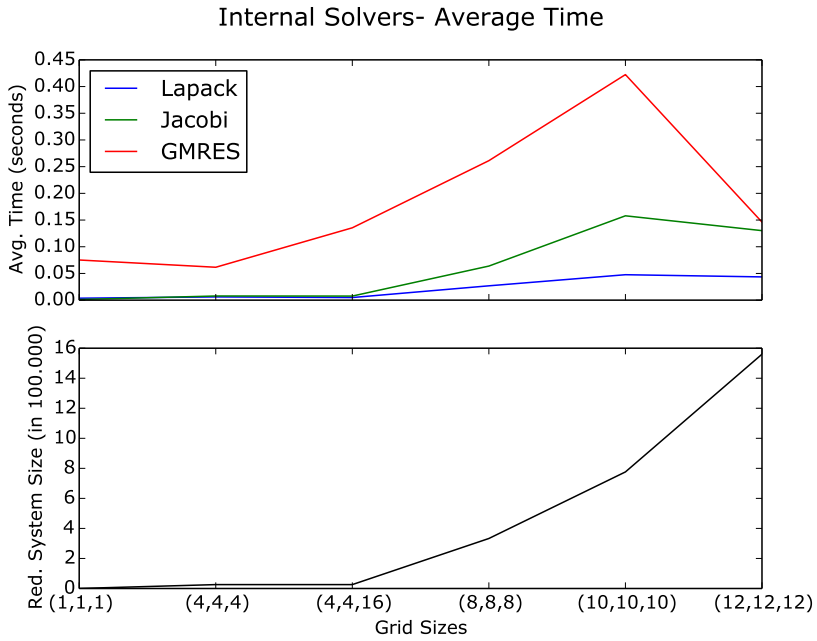


Figure 5.6: Run time of internal solvers on grids of different sizes

the reduced system's size.
The internal Jacobi solver, plotted in green, is comparably fast for smaller grid sizes up to (4,4,16). Afterwards it shows a steeper increase between (4,4,16) and (10,10,10), but then decreases slowly again.
Similarly, the restarted GMRES solver experiences rising run-times up to the (10,10,10) grid, after which the run-time sharply decreases for the (12,12,12) grid. Anyway, its run-times are the slowest throughout the whole test. Additionally, while the run-times of the LAPACK and Jacobi solvers stay similar for problems of the same size, the GMRES solver nearly doubles its run-time between the (4,4,4) and the (4,4,16) grid, although the size of the reduced system is constant here.

A general rule-of-thumb for solvers for linear systems is that direct solvers are expected to perform better for smaller problems, while iterative solvers are better suited for larger set-ups.
The test does not allow for a clear recommendation on which internal solver should be used depending on problem size. In all tested cases, the direct LAPACK solver yielded the fastest results and is advised to use for grid sizes up to at least (12,12,12). Although the behavior of the iterative solvers suggests that for even larger grid sizes, their timing may further decrease and even eventually catch up or even surpass the direct solver, this could not be confirmed due to high memory requirements.

Scaling Tests To better understand the behavior of the modified Gauss solver, a strong scaling test is performed. That means the execution times of a 3D radia-

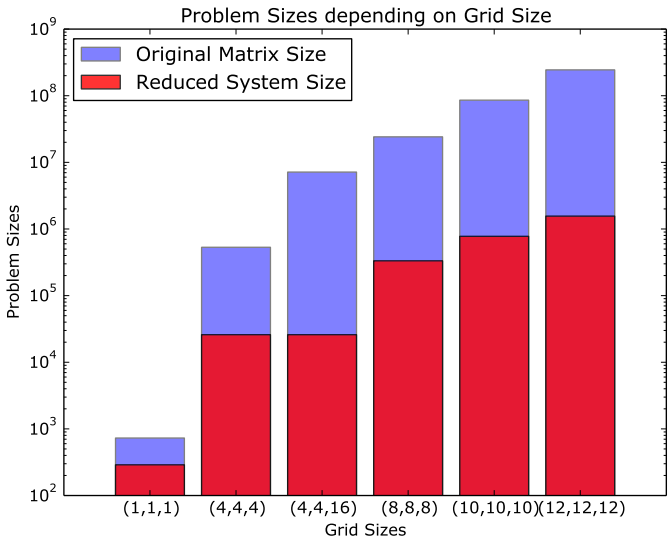


Figure 5.7: Compare original problem size to reduced system size

tive transfer computation is measured for a constant grid size but varying numbers processing elements.

This test is conducted on the Hummel cluster. The radiative transfer problem is solved on a $nx = ny = nz = 16$ grid, with $\epsilon = 10^{-6}$ and the stopping criterion $stop = 10^{-6}$. The tasks-per-node ratio is one, so the number of compute nodes increases with the number of processing elements p. Apart from the serial case with $p = 1$, the number of processors covers the even numbers from two to sixteen. Results are presented in Fig. 5.8.

The serial run gives a reference run-time T_s of 4,222.16 seconds. On $p = 2$ processing elements, the run-time increases up to T_2 =13,012.15 seconds, which is the maximum time measured in this test. For increasing numbers of processors, the run-time drops and reaches the test's minimal run-time of 117.32 seconds for $p = 8$. Further increasing the number of processing elements leads to slightly increased run-times also.
For the fastest radiative transfer computation on $p = 8$ nodes, the computational costs $C = pT_p$ amount to $938.56s$, which is significantly less time than the serial run-time T_s.
The best achieved-speed up $S_8(n) = T_s/T_8$ is 35.99 and the efficiency $E_8(n) = S_8(n)/p$ therefore is 4.5. This corresponds to a super-linear speed-up, since $S_8(n) \geq 8$, and thus the efficiency surpasses the usual limit of $E \leq 1$.

The results of the strong scaling test show that the modified Gauss solver runs significantly more efficient in parallel than in serial, with the exception of $p = 2$. Since it was designed as a parallel algorithm, this does not surprise.
It also indicates, that more processing elements do not always guarantee a faster

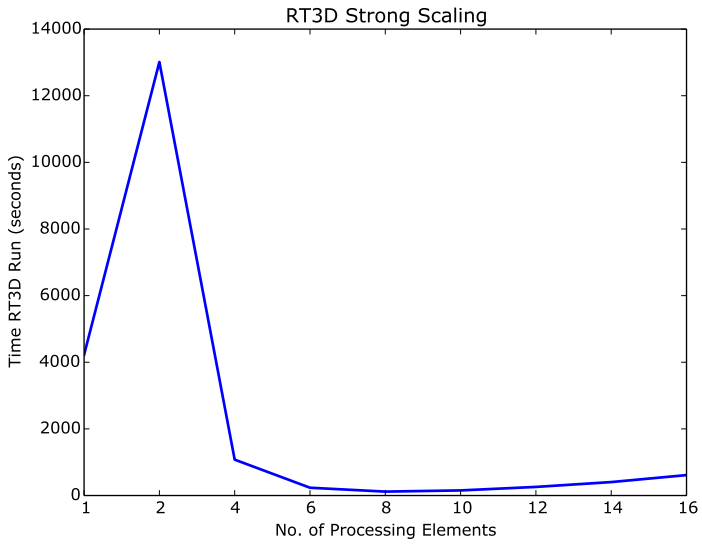


Figure 5.8: Strong Scaling Behavior of the Modified Gauss Solver

execution, and that there is an ideal number of processes depending on the problem size. In this case, the radiative transfer problem is solved fastest with $p = 8$ and grows slower when using more or less processing elements.

The strong scaling test does, however, not give any indications about the performance of the modified Gauss algorithm compared to other solvers. This will be explored in the following tests.

5.3 Tests II: PHOENIX/3D

While the last section already focused on the implementation of the modified Gauss solver, this section takes a closer look at its properties compared to the original PHOENIX/3D radiative transfer solvers.
Apart from the new parallel solver, only one other solver was parallelized, namely an MPI-implementation of the Jacobi solver. All other solvers work serially.

Convergence Behavior The following test addresses the convergence properties of the modified Gauss method compared to the other solvers currently implemented in PHOENIX/3D.
Since for computing a convergence rate an analytic solution of the problem in question is mandatory, this test rather counts the numbers of iterations necessary to reach convergence. Here, this is defined by triggering the user-defined stopping criterion of $stop = 10^{-6}$.

Table 5.4 shows the number of iterations needed to reach convergence for two different grid sizes. The solvers featured are all serial, including the modified Gauss

	(4,4,4) grid	(4,4,16) grid
Jacobi	9	13
Gauss-Seidel	9	10
Diagonal	9	17
Mod. Gauss (LAPACK)	5	6
Mod. Gauss (Jacobi)	5	6
Mod. Gauss (GMRES)	5	6

Table 5.4: Number of Iterations to Reach Convergence

solver, which is run in its pretend-serial mode. The tests are conducted on calvin with $\epsilon = 10^{-1}$.
On both grids, the modified Gauss solver needs significantly less iterations to fulfill the stopping criterion. Also, the number of necessary iterations is independent of the internal solver.

Comparing the parallel PHOENIX/3D solvers, the behavior is consistent with the serial cases: Figures 5.9 and 5.10 show a test that is performed on hummel with a grid of $nx = ny = nz = 16$ and $\epsilon = 10^{-6}$ on $p = 8$ processing elements.
The parallel Jacobi solver needs 17 iterations to reach convergence, while the modified Gauss solver only needs 9. Convergence, again, is defined as a change between iterations of $stop = 10^{-6}$ or less.

Overall, the convergence properties of the modified Gauss solver are promising. In all test cases, it required less iterations to reach the stopping criterion.
With increasing problem size, the number of iterations also increases for both solvers. The slope is comparable, too.

OS-Iterations The behavior that has not yet been studied in detail, is the execution time of the 3D radiative transfer algorithm depending on the used solver. In this test, we time the OS-iterations of a 3D radiative transfer run on Hummel with a $nx = ny = nz = 16$ grid on $p = 8$ processing elements. The other parameters are $\epsilon = 10^{-6}$ and $stop = 10^{-6}$. The modified Gauss method is compared to the parallel Jacobi solver. For the modified Gauss solver, the comparison is conducted with different internal solvers, the LAPACK method as direct method, as well as the Jacobi solver as an iterative scheme.

Figure 5.9 shows the times needed to finish the OS-iterations for the parallel Jacobi solver and the modified Gauss method with internal Jacobi solver.
Overall, the parallel Jacobi solver is much faster. In total, the parallel Jacobi solver needs 0.62s for all OS-steps, while it takes the modified Gauss solver 93.39s. Their behavior also differs: while the iteration times with the Jacobi solver decrease the nearer it comes to the converged solution, the modified Gauss solver needs more time during the first OS-iterations, with the following iterations being nearly constant in time. Out of the overall run-time, 53.6 seconds, meaning 57%

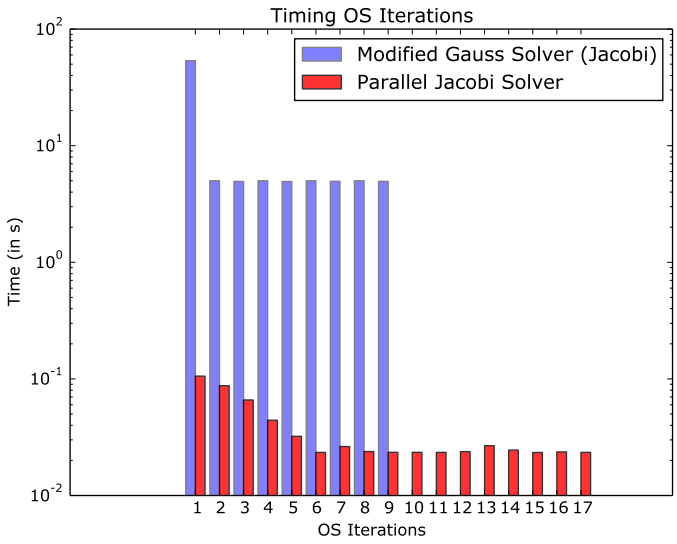


Figure 5.9: Compare time per OS iteration of parallel Jacobi solver and modified Gauss method with internal Jacobi solver

are used for the first iteration. Additionally, the modified Gauss solver uses less iterations to reach the requested accuracy.

Comparing the parallel Jacobi solver to the modified Gauss solver with internal LAPACK solver (see Fig. 5.10), the effect described above even magnifies, since the internal LAPACK routine also uses a factorization approach, which is the same strategy as the modified Gauss solver. Therefore, in the first iteration, the modified Gauss factorization and the internal solver factorization happen, which increases the time needed for the first iteration to $111,36$ seconds, 99%. Following OS-iterations are faster, though, compared to the internal Jacobi solver. Regardless, the overall time of 112.95s is still slower than the Gauss method with internal Jacobi solver and therefore slower than the parallel Jacobi solver.

In summary, the modified Gauss solver is significantly slower than the parallel Jacobi solver. However, Figures 5.9 and 5.10 also show that the modified Gauss solver needs less iterations to reach the desired accuracy. This indicates, that with further optimization, the timing could be improved and may end up to be comparable to the original PHOENIX/3D solver.

5.4 Tests III: OpenCL Version

The results in this section originate from testing the OpenCL stand-alone implementation of the modified parallel Gauss solver. For all following OpenCL tests, only one system of equations is solved per run.

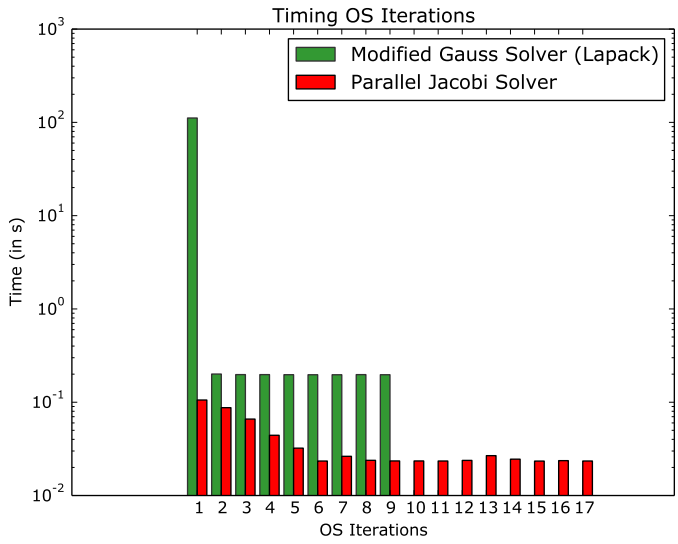


Figure 5.10: Compare time per iteration of parallel Jacobi solver and modified Gauss method with internal LAPACK solver

Compliance of OpenCL Results In this test, the compliance of the modified Gauss algorithm's OpenCL implementation with the exact solution of a linear system of equations is deduced.
Two problems are tested: both have a problem size of $n = 1,000$ and are conducted on the AMD Radeon R9 290. In the first test, a matrix of total bandwidth three is used, while in the second test, the matrix' bandwidth is five. This corresponds to a half-bandwidth of $k = 1$ or $k = 2$, respectively. The side diagonals' entries are set to 1.0, whereas the main diagonal's entries are set to $2k + 1.0$ to account for a diagonally dominant problem. The RHS-vector consists of 1.0 entries. The internal Jacobi solver's stopping criterion is set to $stop = 10^{-15}$ and the test was executed on several numbers of processing elements p.
Tables 5.11a and 5.11b present the relative error between the exact RHS b and the numerical approximation $A\hat{x}$ for the matrix with bandwidth three, respectively, five, on different numbers of processors p.
In both cases, the OpenCL implementation of the modified parallel Gauss solver provides results with a good compliance to the exact solution.

Performance Since OpenCL is designed to be vendor-independent and to be run on heterogeneous machine architectures, the performance of the modified Gauss method's OpenCL implementation is tested on different devices: a CPU and two GPUs. The test system is set up according to the before mentioned test-case with a matrix size of $n = 1000$, a total bandwidth of three, and was solved on $p = 25$ processing elements.
The AMD Radeon R9 290 is the GPU build into minion, while the Intel Core 2 Extreme X9650 is its CPU. The Nvidia Quadro P6000 is the most recent device of this test and build into igor. Figure 5.12 shows the performance of the parallel

p	relative error
5	1.180433e-16
10	1.652355e-16
25	1.669666e-16
100	1.003571e-14

(a) Total bandwidth of three

p	relative error
5	1.431290e-16
10	2.018517e-16
25	2.011466e-16
100	8.636190e-11

(b) Total bandwidth of five

Figure 5.11: Relative errors for a OpenCL test case with different bandwidths on p processing elements

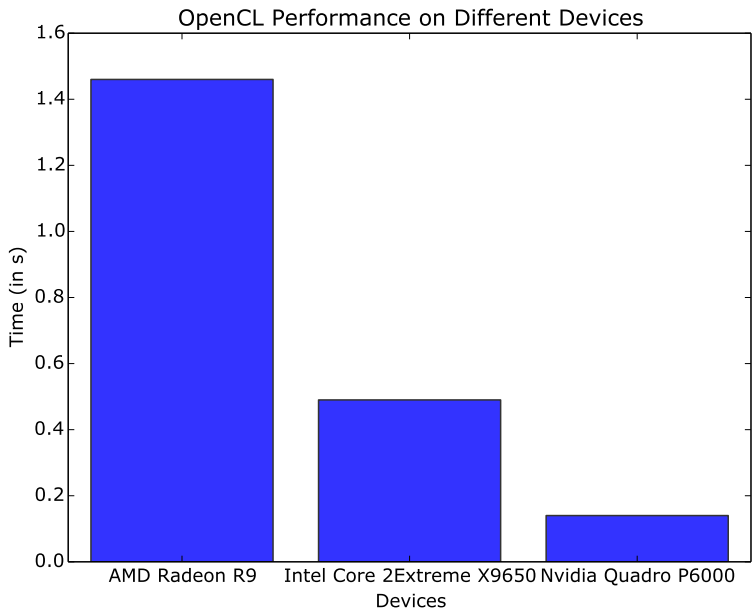


Figure 5.12: Performance of the OpenCL code on different devices

Gauss method's OpenCL implementation on these devices.
As expected, the code executes fastest on the recent Nvidia GPU. Surprisingly, the Intel CPU performs better than the AMD GPU, although the GPU theoretically has a higher throughput. An explanation for this discrepancy could be a bottleneck in memory access. Executing the OpenCL code directly on the CPU, which is also the host device, might save memory operations, compared to the GPU, which needs to load data over a PCIe bus and possibly even swap data to the disk when the device's memory of 4GB is exhausted. Those circumstances may have led to the difference in performance observed here. The Nvidia Quadro P6000, on the other hand, does not show this behavior, possibly because of its high throughput and 24 GB of device memory that prevents the need to swap memory.

In summary, this test confirms that the OpenCL implementation of the modified Gauss solver is indeed portable among different types of devices and vendors.

5.5 Summary

Before we proceed to the discussion, a short summary of test results is given below to emphasize the most important results.

The radiative transfer problem's matrix shows an interesting sparsity pattern: with increased problem size, the matrix becomes more narrow-banded relative to the matrix dimension n. The modified Gauss method, therefore, has potential to be more effectively applied to larger problem sizes.
It also turns out that boundary conditions influence the sparsity pattern. In particular, periodic boundary conditions broaden the band and have to be taken into account in the implementation.

One important result is that both implementations, C/OpenCL and Fortran/MPI, produce results agreeing to the exact solution or the original radiative transfer code, respectively. This holds for the serial, as well as the parallel execution mode and serves as a proof of concept.

The result of the radiative transfer computations is also shown to be independent of the internal solvers, that are incorporated into the Fortran/MPI version. Comparing execution times with different solvers leads to preferring the LAPACK solver, although trends indicate that the iterative solvers, Jacobi and GMRES, might be at advantage for large problems. This could not be tested, though, since memory requirements were too high for larger grid sizes.

A scaling test studies the connection between number of processes and execution time for a fixed grid size. The shortest execution times are reached for $p = 8$, which lies in the middle of the tested range of processor numbers. Compared to the serial code version, this configuration results in a good speed-up.

The modified Gauss method is also compared to the original PHOENIX/3D code in terms of convergence and execution time. Regarding convergence, the Gauss solver needs less OS-iterations to reach a pre-set stopping criterion. Since the number of iterations increases with growing grid-sizes, this behavior should be even more prevalent in larger problems.
With regard to the execution times for OS-iterations, the modified Gauss method is presently significantly slower than its counterpart, the parallel Jacobi solver. This test also calls attention to the iteration behavior of the modified Gauss solver: during the first iteration, the original matrix is factorized, which is reflected by a longer execution time. Successive iterations take shorter time and the time stays constant after the factorization.

There are also properties that were tested indirectly, such as portability and memory management. The portability of the modified Gauss method is granted

for devices that support MPI or OpenCL, respectively, as the tests on different machines show. However, memory requirements limit the problem size that is supported on the devices.

Chapter 6

Discussion and Outlook

After having presented the modified Gauss algorithm and two implementations thereof in the previous chapters, this section discusses the test results of the previous chapter with regard to whether the algorithm meets the expectations: is the method capable of effectively solving the 3D radiative transfer problem in general and, more specifically, is it a reasonable expansion of the PHOENIX/3D atmosphere modeling code as an alternative parallel solver?

6.1 Discussion

The most fundamental property of the modified Gauss solver's implementations is that they reproduce results accurately and reliably. Therefore, the implementations work correctly and the modified Gauss solver is generally suitable to solve the radiative transfer problem.

When applying the modified Gauss method to the radiative transfer problem, it becomes crucial to determine the problem sizes the method is capable of solving efficiently. With increasing spatial grid sizes, the associated matrix becomes more narrow-banded relative to the overall matrix size, which leads to relatively smaller sub-matrices in the modified Gauss algorithm and, therefore, relatively less requirements on memory.
At the same time, increasing grid sizes correlate with more Operator-Splitting-iterations until the algorithm converges. Since the modified Gauss solver generally needs less iterations to reach the stopping criterion than a comparable PHOENIX/3D solver (see Table 5.4), this is favorable especially for larger problems. Additionally, the more iterations are computed, the fewer time does the factorization in the beginning take compared to the total time of all iterations. The average over all iterations then is smaller than for a problem with fewer iterations.
For these reasons, the modified Gauss solver is, in theory, more effective the larger the problem size is.

To be of use in complex scientific applications involving the radiative transfer

problem, an efficient solver's memory requirements should not exceed the memory that current hardware is able to provide. Unfortunately, tests show that the memory requirements of the modified Gauss solver's implementations surpass that for some grid sizes and, therefore, limit the problem size that can be realistically covered.
In the beginning of the modified parallel Gauss algorithm, the off-diagonal submatrices are chosen large enough to contain a number of zero-entries. This is necessary since multiplying sparse sub-matrices later in the algorithm can lead to fill-in, which needs to be accounted for memory-wise from the beginning. Furthermore, the memory management of the MPI-implementation has not yet been optimized.
All in all, the implementations of the modified Gauss solver are not yet memory-aware enough to be used in practical applications.

Another critical property to determine whether an algorithm is suitable to solve the radiative transfer problem in a scientific application is the implementation's execution-time.
The tests conducted on the MPI-implementation of the modified Gauss method show that its execution time is considerably longer than the parallel Jacobi solver's (see, e.g., Fig. 5.9, 5.10). On the other hand, the modified Gauss solver needs significantly less iterations until convergence. With some optimization effort, one can, therefore, anticipate the overall run-time to decrease.
Furthermore, the parallel Jacobi solver in PHOENIX/3D has been highly optimized to the radiative transfer problem and PHOENIX' data structure. The execution-times of the two solvers are, at this point, not entirely comparable.
The current implementations of the modified parallel Gauss solver are not yet fast enough to be used in practical applications. However, tests imply that optimization efforts can decrease the execution time.

Comparing the test results to the work by Arbenz, Cleary and Dongarra (see [10]), it becomes apparent that their test results are more promising than the ones presented here. In Chapter 4, their test setup is described. It was already mentioned, that their setup only includes a stand-alone solver for one linear system of fixed size. Being part of a more complex, scientific application places some limitations on the degree of optimization that is possible for an implementation. Thus, one cannot expect the same efficiency as described in the above work, but following their results, one can still expect optimization efforts to be successful in decreasing run-time and memory requirements.

Since the modified parallel Gauss method involves solving a reduced system, one has to select a method to do this.
In this work, three different serial algorithms were implemented into the MPI-version of the Gauss solver and the work by Arbenz, Cleary and Dongarra (see [10]) additionally proposes a cyclic reduction. This approach consists of applying the modified Gauss method to the reduced system recursively, since the reduced system still fulfills the requirements of narrow-bandedness and is still diagonally dom-

inant. Although this is theoretically possible and an elegant solution, it also leads to computing increasingly smaller systems on less and less processing elements, while more and more of them are idle. This might be inefficient for an application intended to be run highly parallel.
On the other hand, serial solvers are also not a very efficient choice, but are already sufficient to show a trend in behavior. Fig. 5.6 shows the behavior of the internal solvers depending of the problem's size. Differences in behavior are probably due to the algorithms being direct, as the LAPACK solver, or iterative, such as the Jacobi and the restarted GMRES methods.
In summary, selecting an internal solver should consider the problem size because the choice of internal solver influences the overall execution-time of the modified parallel Gauss method.

During tests of the modified Gauss solver's implementations, several different machines were used. Since the radiative transfer solver is intended to run on current and emergent hardware configurations, two different implementations were developed: one in OpenCL that is mostly intended to be run in GPGPU contexts, and a MPI-version for the emerging many-core processors and accelerators, which tend to distributed-memory setups. Both implementations are vendor-independent and therefore portable. Depending on the exact radiative transfer application, one or the other hardware might be favorable. Nevertheless, the portability of implementations is highly recommended for it to be of any practical value.

Even if the hardware setup is fixed, the degree of parallelism that is recommended can vary. Figure 5.8 presents the results of the MPI-implementation's strong scaling test on a Linux-cluster. It becomes clear that there is a range of ideal number of processing elements depending on the problem size. The results indicate that too few processors, as well as too many increase the execution-time compared to the ideal number of processors. Too few processing elements do not have enough computing power to deliver the solution as fast as possible, while too many seem to introduce a communication overhead that is large compared to the work that is actually done on each processing element.
To receive optimal execution-times with a parallel algorithm, the number of processing elements therefore has to be chosen carefully, and tailored to the problem size and hardware configuration.
The scaling test's results for the parallel Gauss algorithm's speed-up and efficiency clearly exceed the expectations. They were calculated for the best performance at $p = 8$. At this number of processing elements, the algorithm shows super-linear speed-up and thus an efficiency surpassing the usual limit of $E \leq 1$. This can be at least partly explained by the definition of speed-up as the ratio of the fastest serial algorithm's execution time to the parallel execution time. Since the fastest serial algorithm for the problem cannot be determined, the comparison was performed between the serial and parallel version of the modified Gauss algorithm. As this method is explicitly designed for parallel execution, the performance of the serial version might distort the resulting speed-up somewhat.

The modified parallel Gauss method has two theoretical restrictions: it is applicable to narrow-banded, diagonally dominant linear systems of equations only. Therefore, it is interesting to consider its behavior when those requirements are not met.
In case of the narrow-bandedness, a matrix that does not have this property could still be theoretically treated with the modified Gauss solver. However, this would increase the bandwidth up to being the matrix size and therefore critically raise both, the memory requirements and execution-time. The solution would still be correct, though.
Applying the modified Gauss solver to a linear system, where the matrix is not diagonally dominant, has similar consequences. The diagonal dominance guarantees that the LR-decomposition can be computed correctly without pivoting. Using this algorithm on a matrix that is too far from the required property will introduce errors into the LR-decomposition and slow down the radiative transfer solver's convergence or possibly prevent convergence at all. In those cases, it is advised to use the modified Gauss solver with pivoting, as it is described in Ch. 4.

6.2 Conclusions

The modified parallel Gauss method has the potential to solve the radiative transfer problem efficiently on large grids. Furthermore, it could possibly enhance the 3D radiative transfer algorithm of the PHOENIX/3D atmosphere modeling code, which currently only has one other parallel solver for OS iterations.
The current implementations, both in Fortran/MPI and C/OpenCL, do not yet provide the memory-management or execution-times to be applicable to practical radiative transfer computations. Nevertheless, the tests indicate that optimization efforts can be expected to reduce run-times and memory requirements: theory and former works suggest that an implementation with appropriate memory requirements is possible. Additionally, the convergence behavior implies that optimization might very well reduce run-times to a competitive level. This is further supported through tests regarding the internal solvers, in that their choice influences the runtime, although to a limited extent.
Considering the modified Gauss algorithms theoretical restrictions to narrow-banded and diagonally dominant systems, the recommendation stands, although dropping the narrow-bandedness criterion only negatively influences performance, but not the result itself. Not adhering to the diagonal dominance requirement, on the other hand, will hurt the method's convergence behavior.
Further tests enforce the notion that portability between hardware-setups is to be considered for an implementation to be of practical use. The machine configuration also influences the degree of parallelism that can be applied effectively.

6.3 Outlook

Analyzing the test's results delivers two main topics designated for further research and optimization: the parallel modified Gauss method's memory management and

its execution-times.
Regarding the memory requirements, it is suggested to further incorporate specific sparse matrix storage formats, such as the Compressed Sparse Row format, that has already been implemented in the OpenCL code for testing purposes. Additionally, the bandwidth of the radiative transfer problem after the Operator Splitting method is determined by the conversion from the 3D spatial grid to the matrix. There are many possibilities to do that, as well as algorithms to determine an order that will minimize the bandwidth, e.g., a version of the Cuthill-McKee algorithm (see [12]) for non-symmetric linear systems. This is an opportunity to greatly reduce the memory usage of the parallel modified Gauss method.

Likewise, there are several strategies to further speed up the modified Gauss implementations: together with sparse matrix storage formats for memory reduction, sparse matrix linear algebra routines optimize matrix operations and reduce the computational effort together with the run-time. Although sparse linear systems occur in a variety of scientific applications, there are still few software packages that implement a sufficient range of sparse matrix routines. As a consequence, it might be necessary to implement those routines from scratch.
Since the internal solvers are still implemented serially, except for the OpenCL version's, there is potential in using parallel versions of the respective algorithms. For all mentioned internal solvers, parallel approaches exist and could be implemented. Furthermore, there is the opportunity to decrease the modified Gauss method's execution-time through the application of machine-optimized software packages, although this might influence the implementation's portability across platforms. Alternatively, a hybrid approach at parallelization through combining code for shared and distributed memory models, see e.g. [35], preserves portability while still enhancing the implementation's performance.

The OpenCL version of the modified Gauss method is still a stand-alone program, but could be integrated into PHOENIX/3D with little effort: a method to distribute the PHOENIX/3D `Lstar` entries to sub-matrices and back is necessary to incorporate the modified parallel Gauss OpenCL implementation into PHOENIX/3D. Furthermore, the current implementation does not yet support solving several similar systems of equations in one execution and needed to be adapted accordingly to reach its full potential. The above recommendations for enhancing runtimes and memory management also apply here.

In summary, although the implementations of the parallel modified Gauss solver are not yet suitable for large applications, this work indicates that further optimization with regard to memory requirements and execution-times will make it possible to solve the radiative transfer problem with it across a broad range of architectures and problem sizes.

Bibliography

[1] *Intel Math Kernel Library. Developer Reference.* Intel Corporation, 2018.

[2] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide.* Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[3] P. Arbenz and M. Hegland. Scalable Stable Solvers for Non-Symmetric Narrow-Banded Linear Systems. *Seventh International Parallel Computing Workshop*, 1997.

[4] M. Arkenberg, V. Wichert, and P. H. Hauschildt. Proceeding On : Parallelisation Of Critical Code Passages In PHOENIX/3D. In *19th Cambridge Workshop on Cool Stars, Stellar Systems, and the Sun (CS19)*, page 31, October 2016.

[5] M. Benzi. Key moments in the history of numerical analysis. 10 2018.

[6] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide.* Society for Industrial and Applied Mathematics, Philadelphia, PA, 1997.

[7] Z. Bothe. Bounds for Rounding Errors in the Gaussian Elimination for Band Systems. 16, 1975.

[8] A. Böttcher. *Rechneraufbau und Rechnerarchitektur.* 2006.

[9] A. Cleary and J. Dongarra. Implementation in ScaLAPACK of Divide-and-Conquer Algorithms for Banded and Tridiagonal Linear Systems. *Technical Report*, 1997.

[10] P. Arbenz, A. Cleary and J. Dongarra. A Comparison of Paralell Solvers for Diagonally Dominant and General Narrow-banded Linear Systems. *EuroPar '99 Parallel Processing*, 1999.

[11] J.M. Conroy. Parallel Algorithms for the Solution of Narrow-Banded Systems. *Applied Numerical Mathematics*, 1989.

[12] E. Cuthill and J. McKee. Reducing the Bandwidth of Sparse Symmetric Matrices. *ACM '69 Proceedings of the 1969 24th National Conference*, 1969.

[13] D.F. Griffiths, J.W. Dold and D.J. Silvester. *Essential Partial Differential Equations*. 2015.

[14] C.P. Dullemond and R. Turolla. An Efficient Algorithm for TwoDimensional Radiative Transfer in Axisymmetric Circumstellar Envelopes and Disks. *A&A*, 2013.

[15] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard, Version 3.1*. High-Performance Computing Center Stuttgart, HLRS, 2015.

[16] A. Munshi, B.R. Gaster, T.G. Mattson, J. Fung and D. Ginsburg. *OpenCL Programming Guide*. 2012.

[17] P. González-Rodríguez and A.D. Kim. Diffuse Optical Tomography Using the One-Way Radiative Transfer Equation. *Biomedical Optics Express*, 2015.

[18] P. H. Hauschildt and E. Baron. A 3D Radiative Transfer Framework. VI. PHOENIX/3D Example Applications. *A&A*, 509, 2010.

[19] P.H. Hauschildt. A Fast Operator Perturbation Method for the Solution of the Special Relativistic Equation of Radiative Transfer in Spherical Symmetry. *J. Quant. Spectrosc. Radiat. Transfer*, 47, 1992.

[20] P.H. Hauschildt and E. Baron. A 3D Radiative Transfer Framework: I. Non-Local Operator Splitting and Continuum Scattering Problems. *A&A*, 451, 2006.

[21] P.H. Hauschildt and E. Baron. A 3D Radiative Transfer Framework: VIII. OpenCL Implementation. *A&A*, 533, 2011.

[22] I. Hubeny and D. Mihalas. *Theory of Stellar Atmospheres*. 2015.

[23] W. Kalkofen. *Numerical Radiative Transfer*. 1987.

[24] R.J. LeVeque. *Finite Difference Methods for Ordinary and Partial Differential Equations*. 2007.

[25] D. Carlson , T. Markham. Schur Complements of Diagonally-Dominant Matrices. *Czech. Math. J.*, 104.

[26] A. Meister. *Numerik linearer Gleichungssysteme*. 2015.

[27] G.L. Olson and P.B. Kunasz. Short Characteristic Solution of the Non-LTE Line Transfer Problem by Operator Perturbation - I. the One-Dimensional Planar Slab. 1987.

[28] O. Pola. Leistungsverbesserung der Simulation von 3D Strahlung auf extrasolare Planeten. Bachelor's thesis, Hamburg University, 2018.

[29] E. Meinköhn, G. Kanschat, R. Rannacher and R. Wehrse. Numerical Methods for Multidimensional Radiative Transfer. *Reactive Flows, Diffusion and Transport*, 2006.

[30] T. Rauber and G. Rünger. *Parallele Programmierung*. 2012.

[31] R.J. Rutten. *Radiative Transfer In Stellar Atmospheres*. 2003.

[32] H. Ernst, J. Schmidt and G. Beneken. *Grundkurs Informatik*. 2015.

[33] H. Schwandt. *Parallele Numerik: eine Einführung*. 2003.

[34] S. Seager and D. Deming. Exoplanet Atmospheres. 2010.

[35] J. Squar. MPI-3 Algorithms for 3D Radiative Transfer on Intel Xeon Phi Coprocessors. Master's thesis, Hamburg University, 2017.

[36] J. Zhonghai, T.P. Charlock, K. Rutledge, K. Stamnes and Y. Wang. Analytical Solution of Radiative Transfer in the Coupled Atmosphere-Ocean System with a Rough Surface. *Applied Optics*, 2006.

[37] R. Trobec, M. Vajteršic and P. Zinterhof, editors. *Parallel Computing: Numerics, Applications and Trends*. 2009.

[38] G.H. Golub, C.F. van Loan. *Matrix Computations*. 1989.

[39] T. Weber. Parallelisierung astrophysikalischer Berechnungen mittels OpenACC. Bachelor's thesis, Hamburg University, 2017.

[40] H.J. Wendker, A. Weigert and L. Wisotzki. *Astronomie und Astrophysik*. 2012.

[41] V. Wichert, M. Arkenberg, and P. H. Hauschildt. A Parallel Numerical Algorithm To Solve Linear Systems Of Equations Emerging From 3D Radiative Transfer. In *19th Cambridge Workshop on Cool Stars, Stellar Systems, and the Sun (CS19)*, page 32, October 2016.

[42] J.H. Wilkinson. Error Analysis of Direct Methods of Matrix Inversion. 1961.

Declaration on Oath

I hereby declare, on oath, that I have written the present dissertation by my own and have not used other than the acknowledged resources and aids.

Eidestattliche Versicherung

Hiermit erkläre ich an Eides statt, dass ich die vorleigende Dissertation selbst verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Hamburg, May 15, 2019