TECHNISCHE UNIVERSITÄT MÜNCHEN

Lehrstuhl für Informatik mit Schwerpunkt Wissenschaftliches Rechnen

Parallelization Strategies for Density Functional Software

Martin Wolfgang Roderus

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des Akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:		Univ.–Prof. Dr. Hans Michael Gerndt
Prüfer der Dissertation:	1.	Univ.–Prof. Dr. Hans–Joachim Bungartz
	2.	Univ.–Prof. Dr. Dr. h.c. Notker Rösch

Die Dissertation wurde am 05.10.2012 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 25.10.2012 angenommen.

Abstract

Today, computational quantum chemistry is one of the most active fields of computational science. It allows to gain insight into the electronic structure of matter from first principles, making it a key technology in a wide variety of chemical applications. However, the numerical treatment of large chemical systems is computationally expensive and demands for the power of parallel supercomputers. But the design of efficient parallel codes proves to be difficult: the diversity of involved data structures and algorithms, as well as the frequently occurring inherent sequential control flow make an efficient use of large processor numbers a challenging problem.

This thesis describes contributions to a collaborative work, which aims to improve the parallel performance of the density functional quantum chemistry code ParaGauss on today's massively parallel supercomputer architectures. Therefore, a static malleable scheduler for parallel eigenvalue computations was developed, which minimizes the execution time of this central step and avoids, thus, severe bottlenecks in simulations of large atomic systems. Furthermore, a high-level Fortran interface to parallel matrix arithmetics was designed and implemented, which facilitates a clear and comprehensible expression of relativistic transformations, while expensive operations are executed by parallel, performance-optimized routines.

In summary, the presented results show better load balancing as well as improved speedup and parallel efficiency for higher processor numbers. Thus, this work states an important step towards the massively parallel computation of complex chemical problems.

Contents

Table of Abbreviations

API	Application Programming Interface
BLAS	Basic Linear Algebra Subprograms
CGTO	Contracted Gaussian Type Orbitals
CPU	Central Processing Unit
DFT	Density Functional Theory
DKH	Douglas–Kroll–Hess
DKS	Dirac–Kohn–Sham
FLOP	Floating Point Operation
GGA	Generalized Gradient Approximation
GPU	Graphics Processing Unit
GTO	Gaussian Type Orbitals
HF	Hartree–Fock
HPC	High Performance Computing
IRREP	Irreducible Representation
KS	Kohn–Sham
LCAO	Linear Combination of Atomic Orbitals
LCGTO	Linear Combination of Gaussian Type Orbitals
LDA	Local Density Approximation
LPT	Longest Processing Time
LRZ	Leibniz Rechenzentrum
MPI	Message Passing Interface
MPTS	Malleable Parallel Task Scheduling
NPTS	Nonmalleable Parallel Task Scheduling
PGC	Point Group Class
PW	Plane Waves
SCF	Self Consistent Field
SIMD	Single Instruction Multiple Data
SPMD	Single Program Multiple Data
STO	Slater Type Orbitals
XC	Exchange Correlation

1. Introduction

Computational science has become indispensable to numerous other sub-fields of science and engineering. It is used to gain insight into real-world problems by means of numerical simulation, which facilitates "virtual experiments" executed on a computer. Thus, computational science was acknowledged the "third pillar of scientific enterprise" [1], next to the two classical approaches – theoretical analysis and physical experiment. Theoretical analysis is vital for the construction of mathematical models, which is the foundation of almost all approaches to understand physical systems and predict observed processes and phenomena. Their examples are countless, ranging from the Newton's laws of motion, established in the seventeenth century, to the fundamental physical laws at the "quantum realm" of atomic and subatomic length scales, discovered in the early twentieth century and formulated in the famous Schrödinger equation. Physical experiments are commonly used to establish or validate existing models, but also to optimize processes, which is for instance a common industrial application. A prominent example for the experimental validation of a theoretical model is the recent discovery of the Higgs–Boson [2] at the European Organization for Nuclear Research, CERN[1]. This discovery proved, at least to a very high probability, that some predictions made by the Standard Model of particle physics, formulated in the 1970s, hold. Yet, the experimental approach often entails severe shortcomings: a physical experiment might be expensive, such as crash tests in the automotive industry, or simply impossible – for example, how could one make two galaxies collide? Furthermore, analytical solutions to complex problems often exist only for very simple cases. Here, computational science offers an opportunity to fill these gaps and facilitates, thus, applications and developments in a large variety of scientific fields. This includes physics, chemistry, biology, human- and earth sciences as well as engineering, however this list is by far not complete.

This thesis discusses advances in computational methods, which are applied in *computational chemistry* – a discipline that employs numerical simulation to solve problems arising in chemistry. The importance of chemical simulations has grown over recent years, which is also reflected by the usage statistics of the national supercomputer HLRB II, installed in the Leibniz Rechenzentrum, Germany[2]: in 2011, more than 16 % of the total CPU-hours consumed was spent in chemical applications, which is the second largest of all fields of application, see Table 1.1. In chemistry, there is a wide range of applications where computational methods play a key role: studies on nano structures, catalysts, chemical reactions, materials or drug design are just a few of the most important exam-

[1]www.cern.ch
[2]www.lrz.de

Field of Application	CPU-h share in %
Computational Fluid Dynamics	31.3
Chemistry	**16.2**
Astrophysics/Cosmology	12.8
Physics - High Energy Physics	11.1
Biophysics/Biology/Bioinformatics	8.8
Physics - Solid State	8.5
Meteorology/Climatology/Oceanography	3.7
Physics - others	2.7
Geophysics	2.4
Others	2.5

Table 1.1.: Usage statistics of the national supercomputer HLRB II, Leibniz Rechenzentrum, Munich, Germany, in 2011, listed by the field of application. Data source: [3].

ples. Here, numerical software is commonly employed to determine important properties of molecules, clusters and solids, such as energy levels, reaction barriers or geometric structures.

A big majority of the methods used in computational chemistry are based on the quantum mechanical model of atoms and molecules. This research field is commonly termed *computational quantum chemistry*. The quantum mechanical model, discovered in the early part of the twentieth century, is the prevalent model to describe matter at the atomic and subatomic scale, where the laws of classical mechanics—the Newton's laws of motion—fail. A central part in the formalism of quantum mechanics plays the so-called *wave function*, which is an abstract description of the *quantum state* of an atomic system. It reveals all information of interest about the electronic properties of the enclosed subatomic particles, especially of the electrons, and their dynamics. In this context, the term *electronic structure* is often used. In quantum chemistry, the wave function is also of central importance. It exposes all chemical properties of interest of a chemical system, provided the appropriate operator is applied to it. Thus, the core problem of any quantum chemical study is the (approximate) determination of the electronic structure of the observed chemical system, carried out in *electronic structure calculations*.

The theoretical foundation for the determination of the wave function is the *Schrödinger equation*, published 1926 by Erwin Schrödinger [4], and reason for his award of the Nobel Price in Physics in 1933. Chapter 2 introduces to its basic concepts. However, its direct solution is only possible for very small systems with a single electron, such as the Hydrogen atom H or the ionized Helium atom He^+. For larger systems, approximate *ab* *initio* methods, implemented in computer codes, are employed. The development of efficient methods looks back at a long history, where the *Hartree-Fock approximation (HF)*, developed in the 1930s, provided the first practically usable method. HF is still in use today, and also laid the foundation for the development of more elaborate methods later on. The probably most important milestone in the history of this development states the invention of the *density functional theory (DFT)*, mainly shaped by Walter Kohn, Pierre Hohenberg and Lu Jeu Sham in the 1960s. Density functional methods provide a good ratio between accuracy and computational costs, which makes them the first choice in the majority of quantum chemical applications today. The award of the Nobel Price in Chemistry to Walter Kohn in 1998, together with John Pople for the development of computational methods in DFT implementations, emphasizes the importance of this contribution to modern chemistry.

Today, there is a variety of computer codes which implement the density functional theory. An overview is given in Chapter 3. As the chapter also shows, most of the codes are based on algorithms requiring $\mathcal{O}(N^3)$ operations, where N is the number of electrons comprised by the chemical system. An application which involves more than 100 atoms can usually not be computed anymore by a sequential computer in reasonable time, and requires parallel computing. Applications treating more than $200 - 300$ atoms demand for parallel supercomputers, which provide a large amount of computing power. Today's supercomputers accommodate an enormous amount of computing resources. Some of them are capable to process several, up to 20, petaFLOPS (10^{15} floating-point operations per second), and it is likely that the first exaFLOPS-machine will be built within this decade [5]. These peak-performance rates are reached by employing massively parallel computer architectures, see also Figure 1.1. However, the parallelization of density functional codes proves to be difficult. For example, the iterative *self-consistent field (SCF)* procedure, requiring between 30 and 100 iterations in a typical electronic structure calculation, is the computationally prevalent part and has an inherently sequential structure. Furthermore, the efficient parallel computation of some of the most expensive numerical problems arising in an SCF cycle, such as the eigenvalue problem, are commonly acknowledged as one of the major challenges in computational science today [6]. And finally, many chemical applications involve *geometry optimizations*, which imply 50 to several hundred electronic structure determinations. Here, the possibility for an independent execution of the single determinations is typically limited and sometimes not given at all. In total, this adds up to $10^3 - 10^5$ SCF iterations in a typical application, with very few potential for their concurrent execution. Thus, massive parallelism is extremely difficult to achieve. This lack of parallelism severely restricts the opportunities to model large chemical systems, such as catalysts, nano-structured materials, as well as large complexes in solution.

This thesis describes contributions to a collaborative work, which aims to improve the parallel performance of the quantum chemistry code *ParaGauss* [8] on today's massively parallel supercomputer architectures. ParaGauss implements the density functional theory using localized Gaussian-type basis functions. Chapter 3 introduces to

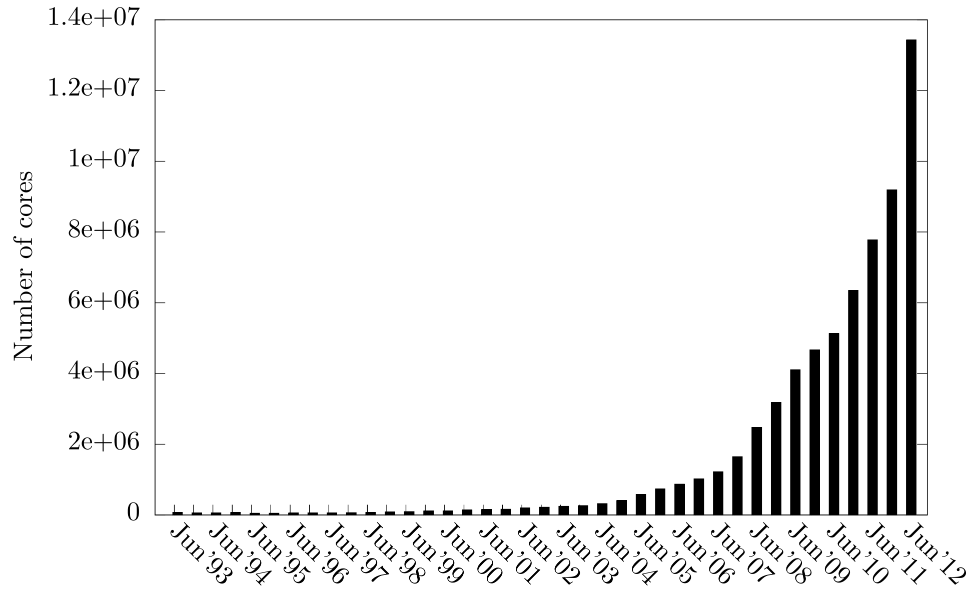


Figure 1.1.: The number of processor cores, comprised by all computer systems in the *TOP500 list*. The TOP500 project maintains a list of the 500 most powerful machines worldwide, according to their score in the LINPACK benchmark [7]. An updated list is released every six months, usually in June and November. Data source: www.top500.org.

these concepts. Its development started in 1994, and was designed from the beginning on for distributed memory architectures [9]. The programming language employed is mostly *Fortran 90/95*, however in some parts *FORTRAN 77* and *C* is used. Furthermore, for message-passing, it initially relied on the software tool PVM [10], which was later replaced by the today prevalently used *message passing interface (MPI)* [11]. The general parallelization paradigm in the ParaGauss project is to directly address the computationally expensive sub-problems arising in an electronic structure calculation (see Chapter 3), and establish new concepts, algorithms, and finally their implementations, which facilitate an efficient parallel execution. This strategy is also followed by the novel contributions to ParaGauss, which are presented in this work.

Thesis Structure

We start by familiarizing the reader with the theory behind quantum chemistry and electronic-structure calculations in Chapter 2. This includes the important Schrödinger equation, the density functional theory, and the Kohn–Sham formalism. Following this, Chapter 3 discusses computational methods, which are commonly used to cast the Kohn–Sham equations into a practical computer code. Here, the emphasis is on methods using localized Gaussian-type functions for the representation of atomic orbitals, as implemented in ParaGauss. After this introduction to the most basic concepts, we present our own contributions in the following two chapters: Chapter 4 describes a novel Fortran interface and its library implementation. This interface allows to express matrix algebra directly in a high-performance Fortran code using a clear and concise pseudo-mathematical syntax. Thus, the existing sequential implementation of relativistic expressions in ParaGauss could be parallelized efficiently, requiring only minor modifications to the original program semantics. Furthermore, Chapter 5 discusses the parallel computation of the generalized eigenvalue problem – an inherent step in each SCF cycle. ParaGauss exploits the symmetry of a molecule to reduce the general problem size. However, this problem reduction results in a symmetric block-diagonal matrix structure, difficult to handle by existing parallel eigenvalue solvers. We present a novel technique, which uses a sophisticated scheduling algorithm, to provide an efficient solution. And finally, Chapter 6 summarizes the achievements of the presented contributions.

2. Quantum Chemistry and Density Functional Theory

This chapter introduces to the most fundamental problem in quantum chemistry—the solution of the Schrödinger equation—and how this problem can principally be solved by applying the variational principle. As we will see, this approach to an exact solution is of rather theoretical significance, but based on this theory there exist approximate methods, which find common application in computational chemistry. The method which is by far most used today is the Kohn–Sham formulation of density functional theory (DFT), whose basic statements are also presented here. These methods facilitate the development of computational schemes and finally practical computer codes, which will be discussed in the next chapter.

2.1. The Schrödinger Equation

As already mentioned, the core problem of most quantum chemical approaches is the solution of the time-independent, non-relativistic Schrödinger equation:

$$\hat{\mathrm{H}}\,\Psi_i(\vec{X}) = E_i \Psi_i(\vec{X})\,. \tag{2.1}$$

The equation states an eigenvalue problem with the Hamilton operator $\hat{\mathrm{H}}$, its eigenvalues E_i, and eigenfunctions Ψ_i. $\hat{\mathrm{H}}$ is a differential operator representing the total energy of a molecular system of interest, which consists of M nuclei and N electrons, and will be further described below.

$\vec{X}$ is a multivariate vector and is defined as

$$\vec{X} := (\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_N, \vec{R}_1, \vec{R}_2, \ldots, \vec{R}_M)\,. \tag{2.2}$$

The electron variable $\vec{x}$ is further defined as

$$\vec{x} := \begin{pmatrix} \vec{r} \\ \sigma \end{pmatrix}, \tag{2.3}$$

and the vectors $\vec{R}$ and $\vec{r}$ represent the spatial coordinates of a corresponding nucleus and electron, respectively:

$$\vec{R},\, \vec{r} := \begin{pmatrix} x \\ y \\ z \end{pmatrix} . \tag{2.4}$$

The *electron spin* σ has only two discrete states: *spin up* and *spin down*,

$$\sigma \in \{\uparrow, \downarrow\} . \tag{2.5}$$

The *Hamilton operator* $\hat{\mathrm{H}}$ consists of two main components:

$$\hat{\mathrm{H}} = \hat{\mathrm{H}}_{\mathrm{kin}} + \mathrm{v} . \tag{2.6}$$

$\hat{\mathrm{H}}_{\mathrm{kin}}$ is an operator representing the classical kinetic energy and is defined as

$$\hat{\mathrm{H}}_{\mathrm{kin}} := -\frac{1}{2}\left(\sum_i^N \nabla_i^2 + \sum_A^M \frac{1}{M_A}\nabla_A^2\right) , \tag{2.7}$$

where ∇^2 is the *Laplace operator* dependent on the three spatial coordinates. The *Coulomb potential* v of the molecular system splits up into three individual terms:

$$\begin{aligned} \mathrm{v} &= \mathrm{v}_{\mathrm{nn}} + \mathrm{v}_{\mathrm{ne}} + \mathrm{v}_{\mathrm{ee}} \\ &:= \sum_A^M \sum_{B>A}^M \frac{Z_A Z_B}{|\vec{R}_A - \vec{R}_B|} - \sum_i^N \sum_A^M \frac{Z_A}{|\vec{r}_i - \vec{R}_A|} + \sum_i^N \sum_{j>i}^N \frac{1}{|\vec{r}_i - \vec{r}_j|} . \end{aligned} \tag{2.8}$$

Here, v_{nn} represents the Coulomb interaction between the nuclei, v_{ne} the nuclei-electron interaction and v_{ee} the interaction between the electrons. Z_A represents the charge of nucleus A.

We note here that for the presentation of all equations, the *system of atomic units (a.u.)* is employed. In this system, all physical quantities, such as mass, length or action, are normalized to basic physical constants, such as the mass of an electron m_e, the Bohr radius a_0 or Planck's constant divided by 2π, $\hbar$, respectively. This allows to write the equations in a very compact form without units. A complete list of all relevant physical constants is given in Appendix A.

The Schrödinger equation (2.1) yields the desired many-body wave function Ψ_i of the i'th *quantum state* of the system as eigenfunction, as well as the corresponding energy E_i as eigenvalue. Ψ_i is a direct representation of its electronic structure and holds all desired chemical information about the system described by the Hamiltonian $\hat{\mathrm{H}}$. In practical applications, the *ground state* Ψ_0 is often of special interest. It is the energetically most stable state, in which molecules often appear in nature, and is indicated by the lowest eigenvalue E_0. Higher eigenvalues indicate excited states, often found in chemical reactions. See Figure 2.1 for an example of the hydrogen atom in its ground state.

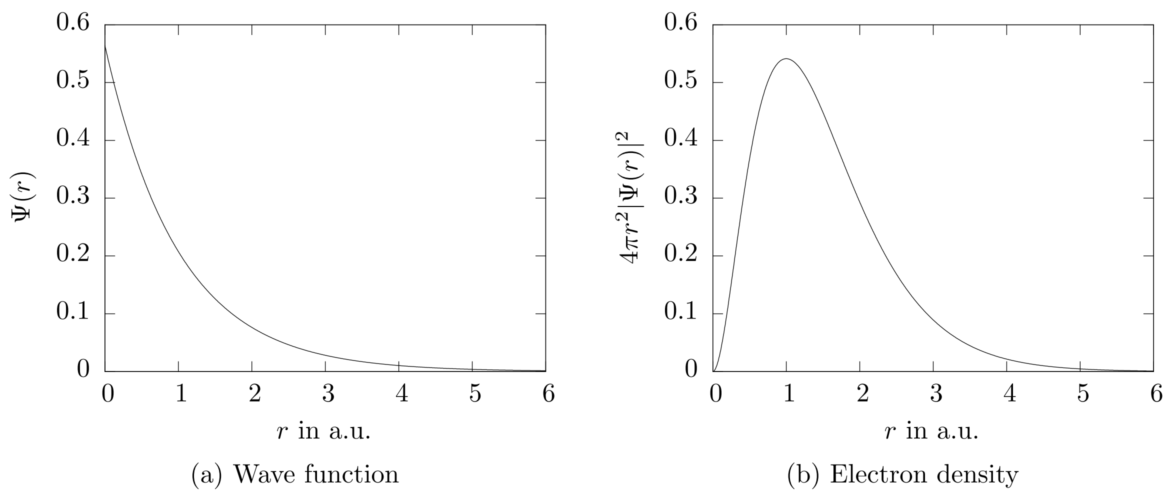


Figure 2.1.: One-dimensional plots showing the radial wave function of the hydrogen (H) atom in its ground state (a), next to an alternative interpretation, which shows the probability to find the electron on a spherical shell around the nucleus (b). The radius r represents the distance from the nucleus, and is measured in atomic units, see Appendix A. Note that the probability in Figure (b) has its peak at exactly 1 a.u., which is the radius of the closest s-orbit in the (simpler) Bohr model of the H-atom.

A direct analytic solution of Equation (2.1) is generally not possible, except for a few cases[1]. The Schrödinger equation is thus more of theoretical relevance. However, there exists a viable strategy to still achieve the ground state: one can obtain an expectation value of the energy which corresponds to a valid function Ψ_{trial} with the energy functional

$$E = \mathrm{E}[\Psi] := \frac{\int \Psi^* \, \hat{\mathrm{H}} \, \Psi d\vec{X}}{\int \Psi^* \Psi d\vec{X}} \,. \tag{2.9}$$

Furthermore, the *variational principle* states that this expectation value can never be lower than the energy value of the ground state:

$$E_{\text{trial}} = \mathrm{E}[\Psi_{\text{trial}}] \geq E_0 = \mathrm{E}[\Psi_0] \,. \tag{2.10}$$

With Equation (2.10) it is now principally possible to search for the ground state Ψ_0 within the functional space of valid wave functions. Valid means that these functions must meet certain requirements in order to make physically sense. For example, Ψ must be continuous everywhere, two times differentiable ($\Psi \in C^2$) and square integrable. Thus, this search can also be stated as a minimization problem:

$$E_0 = \min_{\Psi \in C^2} \mathrm{E}[\Psi] \,. \tag{2.11}$$

[1]To be precise, direct solutions exist for atoms with only one electron, such as hydrogen H or ionized helium He^+.

However, as can be seen in Equations (2.2) to (2.5), $\Psi(\vec{X})$ is a multivariate function. More accurately, the number of variables grows linearly with the number of involved nuclei M and electrons N. This important attribute makes Equation (2.11) a high-dimensional minimization problem, even for practical applications which involve only few electrons.

Obviously, the above stated problem is very difficult to tackle. There exists a number of approximate methods, which yield a wave function with energy close to E_0. However, there also exist simplifications to the model, which reduces the number of dimensions and consequently the problem size. The most ubiquitous one, which is applied in the majority of quantum chemical applications, is the *Born–Oppenheimer approximation.* It is based on the fact that the mass of a nucleus is by several orders of magnitude higher than that of an electron, and moves consequently much slower. For example, only a single proton weights roughly 1800 times more than an electron. This allows—without much loss of accuracy—to consider the nuclei to be fixed in space with zero kinetic energy. As a result, this simplifies the first term in Equation (2.8), v_{nn}, to a mere constant E_{nuc}, which can later be added to the total energy,

$$E = E_{\mathrm{nuc}} + E_{\mathrm{elec}}\,, \tag{2.12}$$

with

$$E_{\mathrm{nuc}} = \sum_{A}^{M} \sum_{B>A}^{M} \frac{Z_A Z_B}{|\vec{R}_A - \vec{R}_B|}\,. \tag{2.13}$$

The simplified *electronic Hamiltonian* has now the form:

$$\begin{aligned} \hat{\mathrm{H}}_{\mathrm{elec}} &= \hat{\mathrm{H}}_{\mathrm{kin}} + \mathrm{v}_{\mathrm{ne}} + \mathrm{v}_{\mathrm{ee}} \\ &:= -\frac{1}{2}\sum_{i}^{N} \nabla_i^2 - \sum_{i}^{N}\sum_{A}^{M} \frac{Z_A}{|\vec{r}_i - \vec{R}_A|} + \sum_{i}^{N}\sum_{j>i}^{N} \frac{1}{|\vec{r}_i - \vec{r}_j|}\,. \end{aligned} \tag{2.14}$$

Another implication is that the spatial variables, assigned to the nuclei in the variable vector $\vec{X}$, are eliminated, which reduces the dimensionality of Ψ to $3N$ (spin variables are neglected):

$$\vec{X}_{\mathrm{elec}} := \{\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_j, \ldots, \vec{x}_N\}\,. \tag{2.15}$$

Other than for the energies E_i, there is no direct physical interpretation for the wave function Ψ_i. However, the square of the function,

$$|\Psi(\vec{X}_{\mathrm{elec}})|^2 d\vec{X}_{\mathrm{elec}}\,, \tag{2.16}$$

yields the probability to find all N electrons simultaneously in volume element $d\vec{X}_{\text{elec}}$. Compare also to Figure 2.1b. An implication of the probability interpretation in Equation (2.16) is that Ψ_{elec} must be *normalized*:

$$\int \dots \int |\Psi(\vec{X}_{\text{elec}})|^2 d\vec{X}_{\text{elec}} = 1\,. \tag{2.17}$$

At this point we introduce another important property of the electronic wave function: it is *antisymmetric* with respect to an interchange of the spatial and spin coordinates of any two electrons[2]:

$$\Psi(\vec{x}_1, \vec{x}_2, \dots, \vec{x}_i, \vec{x}_j, \dots, \vec{x}_N) = -\Psi(\vec{x}_1, \vec{x}_2, \dots, \vec{x}_j, \vec{x}_i, \dots, \vec{x}_N)\,. \tag{2.18}$$

This *antisymmetry principle* applies to all fermions. In contrast, the wave function of bosons is symmetric.

The Born–Oppenheimer approximation states a significant simplification to the search for the ground state, formally expressed in Equation (2.11). Based on this simplification, there exists a variety of methods for the approximate determination of Ψ_0. Among the most used methods are the *Hartree–Fock approximation* and especially the *density functional theory*, both of which are introduced in the next section.

2.2. Determining the Ground State: Hartree–Fock and Kohn–Sham DFT

This section gives an introduction to the basic concepts of the two most common methods, used to achieve the (approximate) ground-state wave function Ψ_0. The Kohn–Sham approach to density functional theory (DFT) is given formally, however it is not intended to be a comprehensive description. For further information we refer to [12, 13] or any other standard textbook on density functional theory.

2.2.1. The Hartree–Fock Approximation

As already mentioned above, the Hartree–Fock approximation, short Hartree–Fock or simply HF, developed in the 1930s, was the first practically useful and reasonably accurate approach to determine the desired ground state Ψ_0. Based on this approach, researchers are able to design computational schemes, which eventually result in software tools for practical applications.

The basic concept of HF is to replace the complicated N-electron wave function by the so-called *Slater determinant*, which is an *antisymmetrized product* of a set of N one-electron wave functions: the *spin orbitals*. These orbital functions are minimized in the *Hartree–Fock equations* according to the total energy represented by the Slater determinant. This scheme introduces the one-electron *Fock operator*, which is a replacement for the N-electron Hamilton operator. Its eigenvalues have the physical interpretation of orbital energies, and its eigenfunctions correspond to the above described spin orbitals.

The HF approach states a practically useful scheme, which can be cast into an algebraic form for its efficient treatment on computers. Typical implementations scale formally as $\mathcal{O}(N^4)$, N being the number of involved electrons. However, the Slater determinant is a fairly inaccurate approximation of the real ground-state wave function – often too imprecise for practical applications. The energy difference between the two functions is called *correlation energy*. Consequently, a number of methods have emerged, which augment HF by computing approximately the missing correlation energy and achieve, thus, higher accuracy. These methods are also called *Post–Hartree–Fock* methods. As few things in life are for free, they also imply higher computational costs: for example, the expansions *MP2* and *MP4* scale as $\mathcal{O}(N^5)$ and $\mathcal{O}(N^7)$, respectively. Other common (expensive) expansions are *configuration interaction (CI)* and *coupled cluster (CC)*, the latter being the most accurate method to date.

(Post–)Hartree–Fock had been the quasi-standard method for many years, until in the 1990s density-functional methods became accurate enough for practical applications. The accuracy of today's DFT-based methods is comparable to expensive wave function methods, such as *MP2*, but are computationally much cheaper – typical implementations scale as $\mathcal{O}(N^3)$ and offer, thus, a superior ratio between accuracy and computational effort. Some chemistry groups also use coupled cluster approaches—e.g. the *NWChem* framework contains a popular implementation [14]—for applications which require very high accuracy, but the vast majority of today's software—such as ParaGauss—is based on density functional methods. The rest of this work copes exclusively with DFT, starting here with a brief introduction.

2.2.2. Electron Density and the Kohn–Sham Approach

Here we introduce the reader to the basic concepts of density functional theory (DFT), and, based on it, the Kohn–Sham approach, which is a practical method for the determination of the ground-state density. As the topic is very comprehensive, we will only cover the fundamental statements necessary for the understanding of the following computational schemes.

In DFT, the complicated high-dimensional wave function $\Psi(\vec{X})$ as central variable is replaced by the *electron density* or *charge density* $\rho(\vec{r})$, which depends solely on the spatial vector $\vec{r} \in \mathbb{R}^3$. It is defined as the multiple integral

$$\rho(\vec{r}) = N \int \dots \int |\Psi(\vec{X})|^2 d\sigma_1 d\vec{x}_2 \dots d\vec{x}_N \,. \tag{2.19}$$

Here, Ψ refers to the normalized electronic wave function (see Equation (2.17)). The electron density ρ represents a probability distribution, in which

$$\rho(\vec{r})d\vec{r} \tag{2.20}$$

indicates the number of electrons[3] in the infinitesimal volume $d\vec{r}$. Furthermore, as a consequence of Equation (2.19), the integral

$$\int \rho(\vec{r})d\vec{r} = N \tag{2.21}$$

yields the total number of electrons comprised by the Hamiltonian $\hat{\mathrm{H}}$.

In 1964, Walter Kohn and Pierre Hohenberg established the theoretical foundation for the density functional theory in their two famous *Hohenberg–Kohn theorems* [15]. Specifically, it was shown that

1. the electron wave function Ψ is a unique functional of the electron density ρ (to within a constant);

2. the energy density functional $F^{HK}[\rho]$ yields its minimal value for the *ground state density* ρ_0: $F^{HK}[\rho_{\text{trial}}] \geq F^{HK}[\rho_0] = E_0$. Hence, the variational principle applies.

The explicit form of F^{HK} is to date unknown – parts of the functional have to be modeled approximately. See also the discussion later in this section.

In the following year, Walter Kohn and Lu Jeu Sham proposed a workable computational approach to determine the ground state density [16]: the *Kohn–Sham (KS) approach.* Most of the DFT-implementations existing today are based on KS. In analogy to Hartree–Fock, it introduces a reference system of N *non-interacting Kohn–Sham orbitals* (short KS orbitals) $\varphi_i(\vec{x})$. Recall the electron variable $\vec{x}$ from Section 2.1, which embraces a spatial vector as well as a spin coordinate:

$$\vec{x} := \begin{pmatrix} \vec{r} \\ \sigma \end{pmatrix} . \tag{2.22}$$

The ground state density is constructed from the KS orbitals as

$$\rho_0(\vec{r}) = \sum_i^N \sum_\sigma |\varphi_i(\vec{r}, \sigma)|^2 . \tag{2.23}$$

According to Kohn–Sham, the functional, which yields the energy comprised by a given electron density ρ, is given as:

[3] potentially all electrons existent in the molecular system

$$
\begin{aligned}
E[\rho] &= E_{\mathrm{kin}}[\rho] + E_{\mathrm{ne}}[\rho] + E_{\mathrm{coul}}[\rho] + E_{\mathrm{XC}}[\rho] && (2.24)\\
&= E_{\mathrm{kin}}[\rho] && (2.25)\\
&\qquad + \int V_{\mathrm{ne}}\rho(\vec{r})dr && (2.26)\\
&\qquad\qquad + \frac{1}{2}\iint \frac{\rho(\vec{r})\rho(\vec{r}')}{|\vec{r}-\vec{r}'|}d\vec{r}d\vec{r}' && (2.27)\\
&\qquad\qquad\qquad + \int f_{xc}(\rho(\vec{r}))d\vec{r} && (2.28)\\
&= -\frac{1}{2}\sum_i^N \int \varphi_i \nabla^2 \varphi_i d\vec{r} && (2.29)\\
&\qquad - \sum_i^N \int \sum_A^M \frac{Z_A}{|\vec{r}-\vec{R}|}|\varphi_i(\vec{r})|^2 d\vec{r} && (2.30)\\
&\qquad\qquad + \frac{1}{2}\sum_i^N\sum_j^N \iint |\varphi_i(\vec{r})|^2 \frac{1}{|\vec{r}-\vec{r}'|}|\varphi_j(\vec{r}')|^2 d\vec{r}d\vec{r}' && (2.31)\\
&\qquad\qquad\qquad + E_{\mathrm{XC}}[\rho]\,. && (2.32)
\end{aligned}
$$

Note that this functional is given in its spin-independent form. Furthermore, it is subject to the Born–Oppenheimer approximation (see Section 2.1), consequently the nucleus-nucleus interaction is neglected and added later as a constant.

Equations (2.25) to (2.28) give the functionals depending on the density ρ, whereas Equations (2.29) to (2.32) present the same terms, in a form dependent on the orbital functions φ_i.

The kinetic energy functional E_{kin} in Equation (2.29) is exclusively given in its orbital-dependent form. This functional yields the exact kinetic energy of a reference system of *non-interacting* electrons. As electrons actually do interact by Coulomb correlation, there is a difference between the energy of this reference system and the real kinetic energy. This residual is treated in the XC-functional, explained later.

The second term, E_{ne} in Equations (2.26) and (2.30), is the classical electrostatic nucleus-electron attraction and is an exact representation of the *ne*-term in the original Hamiltonian in Equation (2.14).

E_{coul} in Equations. (2.27) and (2.31) represents the classical electron-electron repulsion due to the electrostatic force. It introduces an error, because it also implies that an electron interacts with itself. This self-interaction is corrected later by the XC-term.

The *exchange-correlation (XC)* term in Equation (2.28) contains all parts which are not known in their explicit form, such as the non-classical effects of exchange and correlation, as well as the above mentioned residual of the kinetic energy and a correction term for self-interaction, introduced in the Coulomb functional. To date, this term is not known in its exact form, so quantum chemical applications have to resort to approximate representations. It is the critical modeling part in density-functional methods, and the right choice of the XC-potential is of crucial importance for the accuracy of the method.

The choice of a specific XC functional typically depends on the type of application, and there exists a great amount of literature providing functionals for all kinds of common applications. However, the development of accurate functionals is still one of the most active fields of theoretical chemistry, and consequently the approximations are becoming gradually better.

The energy functional E_{XC} is formed from the *energy density* function f_{xc}. In Equation (2.28), f_{xc} is given in the so-called *local density approximation (LDA)* form, which depends only on the total density[4]. Another class of representations is the *generalized gradient approximation (GGA)*, which depends on different variables, such as the spin-dependent density and its gradients:

$$E_{\mathrm{XC}}[\rho] = \int f_{xc}(\rho_{\langle\uparrow\rangle}, \rho_{\langle\downarrow\rangle}, \nabla\rho_{\langle\uparrow\rangle}, \nabla\rho_{\langle\downarrow\rangle}, \ \ldots)d\vec{r}. \tag{2.33}$$

The latter is more complicated to evaluate and introduces additional computational effort (a constant factor) when implemented, but gives more accurate results in many applications.

Given a feasible XC-functional, one can now apply the variational principle and minimize the total energy E by variation of the spin orbitals φ_i. The orbitals which yield the ground state energy E_0 can be achieved by the *Kohn–Sham equations* (see [17] for a detailed derivation)

$$\begin{aligned}(H_{\mathrm{kin}} + V_{\mathrm{ne}} + V_{\mathrm{coul}} + V_{\mathrm{XC}})\,\varphi_i \\ := \left(-\frac{1}{2}\nabla^2 - \sum_{A}^{M}\frac{Z_A}{|\vec{r}-\vec{R}|} + \int\frac{\rho(\vec{r'})}{|\vec{r}-\vec{r'}|}d\vec{r'} + V_{\mathrm{XC}}(\vec{r})\right)\varphi_i \\ = \varepsilon_i\varphi_i\,,\end{aligned} \tag{2.34}$$

or in its short form

$$\hat{\mathrm{f}}^{\mathrm{KS}}\,\varphi_i = \varepsilon_i\varphi_i\,, \tag{2.35}$$

[4]The corresponding spin-dependent representation is called *local-spin-density approximation (LSDA).*

where the *Kohn–Sham operator* $\hat{\mathrm{f}}^{\mathrm{KS}}$ comprises the terms in the parenthesis in Equation (2.34). The exchange-correlation potential V_{XC} is defined as the functional derivative of E_{XC} with respect to the density:

$$V_{\mathrm{XC}} := \frac{\delta E_{\mathrm{XC}}}{\delta \rho} \,. \tag{2.36}$$

One important implication of the Kohn–Sham approach is that there exists a mutual dependency: the KS equations are constructed from the electron density ρ, which is constructed from the spin orbitals φ_i. However, the spin orbitals are in turn the outcome of the KS equations:

$$\rho \Rightarrow \hat{\mathrm{f}}^{\mathrm{KS}} \Rightarrow \varphi_i \Rightarrow \rho \,. \tag{2.37}$$

Practically, this dependency is solved in the *self-consistent field (SCF)* procedure, introduced in Section 3.1.2.

The Kohn–Sham approach provides a feasible strategy for the determination of the ground state of a molecular system. The next chapter presents a scheme for its practical implementation in a computer program.

3. Kohn–Sham Implementation: A Parallel Computational Scheme

This chapter presents how the Kohn–Sham strategy for the determination of the ground state, presented in the previous chapter, can be finally mapped onto a computational scheme, feasible for the practical implementation in a density functional code to carry out electronic structure calculations. Today there exist two main approaches, together with several other techniques of minor importance, which all result in very different numerical problems. This chapter focuses on one main branch, the *LCGTO ansatz*, as implemented in ParaGauss and a great variety of other state-of-the-art chemistry codes. Next to the basic idea of LCGTO, we give a discussion on the employed basis set, highlight the most important computational problems, and show the involved costs along with some parallelization strategies applied.

Throughout the chapter, we will use the $\mathcal{O}$-notation to express computational costs. This makes sense because in practical applications, high asymptotic costs are also reflected by real execution times, even for "small" problem sizes. In other words, those tasks whose asymptotic cost function exhibits the highest polynomial degree (here $\mathcal{O}(N^3)$ with N being the number of involved electrons) do usually also state the expensive parts in practical applications. In this chapter, we will concentrate on these tasks.

As we will see, in a typical, non-relativistic, DFT-based electronic-structure calculation, there exist three main computational steps: the generation of the Coulomb contribution (Section 3.1.4), the generation of the exchange-correlation contribution (Section 3.1.5), and the solution of a generalized matrix eigenvalue problem (Chapter 5). These steps are expensive in terms of asymptotic costs of $\mathcal{O}(N^3)$[1], as well as practical costs reflected by the execution times of electronic structure calculations. What makes the efficient (parallel) computation of those parts even more important is that they are part of the iterative SCF routine, described in Section 3.1.2, meaning that these tasks need to be solved several times (typically 20 up to 100) in an electronic structure calculation. If geometry optimization is required by the application, the typical number of required solutions adds up to 10^3 to 10^5. Furthermore, in case relativistic effects are considered for higher accuracy, additional transformations of the Hamiltonian are necessary. This requires linear algebra operations with $\mathcal{O}(N^3)$ costs. Chapter 4 discusses this issue in detail.

[1]The costs of a straightforward implementation of the Coulomb contribution actually scale as $\mathcal{O}(N^4)$, however we will discuss a technique in Section 3.1.4 which reduces this to $\mathcal{O}(N^3)$.

The algorithms presented in this chapter are commonly used in a large variety of DFT codes. However, besides this widely established class of $\mathcal{O}(N^3)$-algorithms, there is a branch of theoretical chemistry, which strives to develop methods based on algorithms with $\mathcal{O}(N)$ runtime. In chemistry, this field of research is commonly referred to as *linear scaling*. These methods work well for atomic systems, which are expanded in space. However, in this work we do not cover linear scaling methods, for more information we refer to the review article [18].

As already mentioned in Chapter 1, the general parallelization paradigm in the ParaGauss project is to directly address the computationally expensive sub-problems arising in an electronic structure calculation. Practically, this concerns mostly the $\mathcal{O}(N^3)$-steps, presented in this chapter. Luckily, the two most expensive ones—the Coulomb contribution and the exchange-correlation contribution—are relatively easy to parallelize[2]. However, the relativistic transformations and especially the generalized eigenvalue problem require more sophisticated techniques to achieve good scalability. The main contribution of this work focuses on those two problems, which will be presented in Chapters 4 and 5, respectively.

3.1. Orbital Representation: The LCGTO Ansatz

This section discusses how the theoretical formulations of the Kohn–Sham approach, introduced in Chapter 2, can be finally cast into a computational algorithm for its practical solution. The first essential step is to define a set of functions, which reproduce the KS orbitals φ_i, introduced in Section 2.2.2. This set is commonly referred to as the *basis set*, and the functions therein *basis functions*[3]. There are two predominant types of approaches, implemented in a variety of codes. The first approach uses a linear combination of Slater- or Gaussian-type functions, localized at the spatial coordinate of the involved nuclei. This scheme is usually termed *linear combination of atomic orbitals (LCAO)*, or, if Gaussian-type functions are used, *linear combination of Gaussian-type orbitals (LCGTO)*. LCGTO bases are implemented in ParaGauss, as well as in the codes *Gaussian* [19], *TURBOMOLE* [20] or the DFT-module of *NWChem* [14] (among many others). Another popular approach is to use a finite number of *plane waves (PW)*, implemented e.g. in the chemistry codes *VASP* [21], *NWChem* [14] or *Quantum ESPRESSO* [22].

These approaches result in numerical problems of very different characteristics. While LCAO/LCGTO implementations typically yield compact, dense matrices, the matrix representations of PW are rather large and sparse. On the application side, PW is

[2]This applies only if the chargefit technique is applied to the Coulomb contribution, see Section 3.1.4.

[3]The term "basis function" is not strictly correct in a mathematical sense, because the functions do not span a vector space. Here, we refer to the functions comprised by a basis set.

especially favorable to model periodic systems, such as solids, whereas LCAO/LCGTO is more suitable for finite systems, such as single molecules. For more information, please refer to Chapter 4 of [23].

3.1.1. From the Kohn–Sham Equations to a Discrete Generalized Matrix Eigenvalue Problem

The LCAO scheme introduces a set of L basis functions, η_μ, which, multiplied by coefficients $c_{\mu i}$ and linearly combined, model the original Kohn–Sham orbitals φ_i from Equation (2.35):

$$\varphi_i = \sum_{\mu=1}^{L} c_{\mu i}\eta_\mu\,. \tag{3.1}$$

Practically, Equation (3.1) is hard to comply with, as it requires either the basis functions η to have the *exact* appearance of the orbitals, or L to be infinite. The first requirement does obviously make no sense, since the orbital functions is what we are actually looking for. And furthermore, L needs to be finite, unless we have an infinitely fast computer available. A wise choice of η is crucially important, as it significantly impacts both the accuracy and the costs of electronic structure computations. However, we postpone this discussion to Section 3.1.3, for now let η_μ be an arbitrary real function.

In the LCAO scheme, one proceeds by inserting Equation (3.1) into Equation (2.35):

$$\hat{\mathrm{f}}^{\mathrm{KS}} \sum_{v=1}^{L} c_{vi}\eta_v = \varepsilon_i \sum_{v=1}^{L} c_{vi}\eta_v\,. \tag{3.2}$$

We now multiply from the left with a basis function η_μ and integrate over space. The resulting equation can be written as a sum of L equations:

$$\sum_{v=1}^{L} c_{vi} \int \eta_\mu(\vec{r})\,\hat{\mathrm{f}}^{\mathrm{KS}}(\vec{r})\eta_v(\vec{r})\; d\vec{r} = \varepsilon_i \sum_{v=1}^{L} c_{vi} \int \eta_\mu(\vec{r})\eta_v(\vec{r})\; d\vec{r}\,. \tag{3.3}$$

The algebraic equations in (3.3) can also be cast into a matrix form. Therefore, we introduce the real symmetric *Hamilton matrix* $H \in \mathbb{R}^{L\times L}, H_{\mu v} = H_{v\mu}$, as well as the *overlap matrix* $S \in \mathbb{R}^{L\times L}, S_{\mu v} = S_{v\mu}$, with matrix elements

$$H_{\mu v} = \int \eta_\mu(\vec{r})\,\hat{\mathrm{f}}^{\mathrm{KS}}(\vec{r})\eta_v(\vec{r})\; d\vec{r}, \tag{3.4}$$

$$S_{\mu v} = \int \eta_\mu(\vec{r})\eta_v(\vec{r})\; d\vec{r}\,. \tag{3.5}$$

Furthermore, we introduce the *coefficient matrix* $C \in \mathbb{R}^{L\times L}$, containing the coefficient vectors to establish the orbital functions φ_i according to Equation (3.1) as columns

$$C = \begin{pmatrix} c_{11} & \dots & c_{1L} \\ \vdots & \ddots & \vdots \\ c_{L1} & \dots & c_{LL} \end{pmatrix}, \tag{3.6}$$

as well as the diagonal matrix $E \in \mathbb{R}^{L\times L}$, which has the orbital energies ε_i as its diagonal and zeroes as all other elements:

$$E = \operatorname{diag}(\varepsilon_1 \dots \varepsilon_L)\,. \tag{3.7}$$

Using the here introduced matrices, we can rewrite Equation (3.3) in matrix notation:

$$HC = SCE\,. \tag{3.8}$$

Equation (3.8) represents a *generalized matrix eigenvalue problem*, a well-known problem from linear algebra. Chapter 5 elaborates on this topic in detail.

One usually proceeds by splitting up the entries of $H_{\mu v}$ into several contributions, as the different components of the Kohn–Sham operator $\hat{\mathrm{f}}^{\mathrm{KS}}$ typically result in computational problems of diverse characteristics. We therefore give Equation (3.4) in its explicit form:

$$\begin{aligned} H_{\mu v} &= \int \eta_\mu(\vec{r})\, \hat{\mathrm{f}}^{\mathrm{KS}}(\vec{r})\eta_v(\vec{r})\; d\vec{r} && (3.9)\\ &= \int \eta_\mu(\vec{r}) \left(-\frac{1}{2}\nabla^2 - \sum_A^M \frac{Z_A}{|\vec{r}-\vec{R}|} + \int \frac{\rho(\vec{r'})}{|\vec{r}-\vec{r'}|} d\vec{r'} + V_{\mathrm{XC}}(\vec{r}) \right) \eta_v(\vec{r})\; d\vec{r} && (3.10)\\ &= -\frac{1}{2}\int \eta_\mu(\vec{r})\nabla^2\eta_v(\vec{r})\; d\vec{r} && (3.11)\\ &\quad - \int \eta_\mu(\vec{r}) \frac{Z_A}{|\vec{r}-\vec{R}|}\eta_v(\vec{r})\; d\vec{r} && (3.12)\\ &\quad + \iint \eta_\mu(\vec{r}) \frac{\rho(\vec{r'})}{|\vec{r}-\vec{r'}|}\eta_v(\vec{r})\; d\vec{r}d\vec{r'} && (3.13)\\ &\quad + \int \eta_\mu(\vec{r})V_{\mathrm{XC}}(\vec{r})\eta_v(\vec{r})\; d\vec{r}\,. && (3.14) \end{aligned}$$

The first two terms, Equations (3.11) and (3.12), representing the kinetic energy and the electron-nuclear interaction, are comprised into a contribution which we call W:

$$W_{\mu v} = \int \eta_\mu(\vec{r}) \left(-\frac{1}{2}\nabla^2 + \frac{Z_A}{|\vec{r}-\vec{R}|} \right) \eta_v(\vec{r})\; d\vec{r}\,. \tag{3.15}$$

Its computational steps—mostly multiplications and finding derivatives—can be done efficiently and state a minor problem in the overall process (provided η is chosen wisely).

The second contribution to the Hamilton matrix is the Coulomb potential $J_{\mu v}$, given in Equation (3.13). Applying the LCAO scheme, the equation evolves into

$$J_{\mu v} = \sum_{\lambda}^{L} \sum_{\sigma}^{L} P_{\lambda\sigma} \iint \eta_\mu(\vec{r})\eta_v(\vec{r}) \frac{1}{\vec{r} - \vec{r}'} \eta_\lambda(\vec{r}')\eta_\sigma(\vec{r}')\; d\vec{r}\; d\vec{r}' \,. \tag{3.16}$$

Here we introduce the *density matrix* P, constructed from the eigenvector matrix C:

$$P_{\lambda\sigma} = \sum_{i}^{N} C_{\lambda i} C_{\sigma i} \,. \tag{3.17}$$

The Coulomb contribution in Equation (3.16) is expressed as a *four-center-two-electron integral*. Formally, the total number of two-electron integrals to be computed is about $L^4/8 = \mathcal{O}(N^4)$, which makes the computation of J the asymptotically most expensive step in the overall process of generating H. Fortunately, there exist simplifications, which allow for a computationally less demanding complexity. More details will be presented in Section 3.1.4.

The last part refers to the exchange-correlation potential V_{XC}. Its matrix contribution is given by

$$V^{\text{XC}}_{\mu v} = \int \eta_\mu(\vec{r}) V_{\text{XC}}(\vec{r}) \eta_v(\vec{r})\; d\vec{r} \,. \tag{3.18}$$

The definition from Equation (3.14) is left unchanged, since the explicit form of V_{XC} is not generally known. Here, practical applications employ approximate potentials, which usually change from application to application. Their appearance can be complicated, and does normally not allow for analytic integration. Thus, codes need to apply numerical integration schemes to solve Equation (3.18), which will be further discussed Section 3.1.5.

3.1.2. The Iterative SCF Algorithm

Here, the *self-consistent field (SCF)* algorithm is introduced, to overcome one inherent problem of the Kohn–Sham approach, already mentioned in Section 2.2.2: the mutual dependency between the density, the KS-operator and the orbital functions. This mutual dependency, formally expressed in Equation (2.37), is also reflected in the LCGTO scheme, where the matrix-equivalent is

$$P \Rightarrow H \Rightarrow C \Rightarrow P \,. \tag{3.19}$$

In the SCF approach, one starts by constructing an initial Hamiltonian from density-independent terms. A straightforward solution is to simply use the matrix contributions, comprised in matrix W (see Equation (3.15)),

$$H^{\langle 0 \rangle} = W \,, \tag{3.20}$$

which however often results in poor SCF-convergence. To improve this situation, practical applications add a first heuristic "guess" of the Coulomb- and the exchange-correlation terms.

From $H^{\langle 0\rangle}$, the first approximate coefficient matrix $C^{\langle 1\rangle}$ can be achieved by solving the generalized eigenvalue problem in Equation (3.8). From $C^{\langle 1\rangle}$ one constructs the density matrix $P^{\langle 1\rangle}$, and subsequently the contributions $J^{\langle 1\rangle}$ and $V_{\mathrm{XC}}^{\langle 1\rangle}$. These newly established contributions can be comprised in a new Hamiltonian

$$H^{\langle 1\rangle} = W + J^{\langle 1\rangle} + V_{\mathrm{XC}}^{\langle 1\rangle}\,, \tag{3.21}$$

from which $C^{\langle 2\rangle}$ can be achieved in a new iteration. This procedure is repeated until convergence is reached. One usually checks for convergence by means of difference between the density matrices

$$\gamma^{\langle k\rangle} = \max_{\lambda\sigma} |P_{\lambda\sigma}^{\langle k\rangle} - P_{\lambda\sigma}^{\langle k-1\rangle}|\;, \quad k \geq 1\,, \tag{3.22}$$

where one iterates until $\gamma^{\langle k\rangle}$ falls below some threshold γ^*. A typical value for γ^* is 10^{-8}. Figure 3.1 shows a flowchart of the overall SCF procedure. To further improve the SCF-convergence, there exist more advanced techniques, such as to mix Hamiltonians from different iterations, or the DIIS technique, see [24].

3.1.3. Basis Sets

The objective is to find a set of basis functions, whose linear combination provides a good representation of the actual orbital functions. The functions should be chosen such to allow the set to be as small as possible, while still providing an accuracy high enough for practical applications. Large bases generally have a negative impact on the execution time of the electronic structure calculation, and may result in numerical instabilities.

A good approximation provide *Slater type orbitals (STO)*, which are similar to the hydrogen wave function, known in its explicit form (see Figure 2.1). STOs have the general form

$$\eta_{nlm}^{\mathrm{STO}} = N Y_{lm}(\vec{r}) r^{n-1} e^{-\alpha r}\,, \tag{3.23}$$

where N is a normalization constant which ensures that $\int \eta_\mu^* \eta_\mu \, d\vec{r} = 1$. The radius $r = |\vec{r} - \vec{R}|$ is anchored at the atomic center $\vec{R}$, and n, l and m are the basic quantum numbers:

- n is the *principal quantum number* (electron shell or energy level; $n = 1, 2, 3, \ldots$),
- l is the *angular momentum number* and describes the subshell: $0 \leq l \leq n-1$; 0:=s-orbital, 1:=p-orbital, 2:=d-orbital, 3:=f-orbital, etc. Finally,
- m is the *magnetic quantum number* $(-l \leq m \leq l)$.

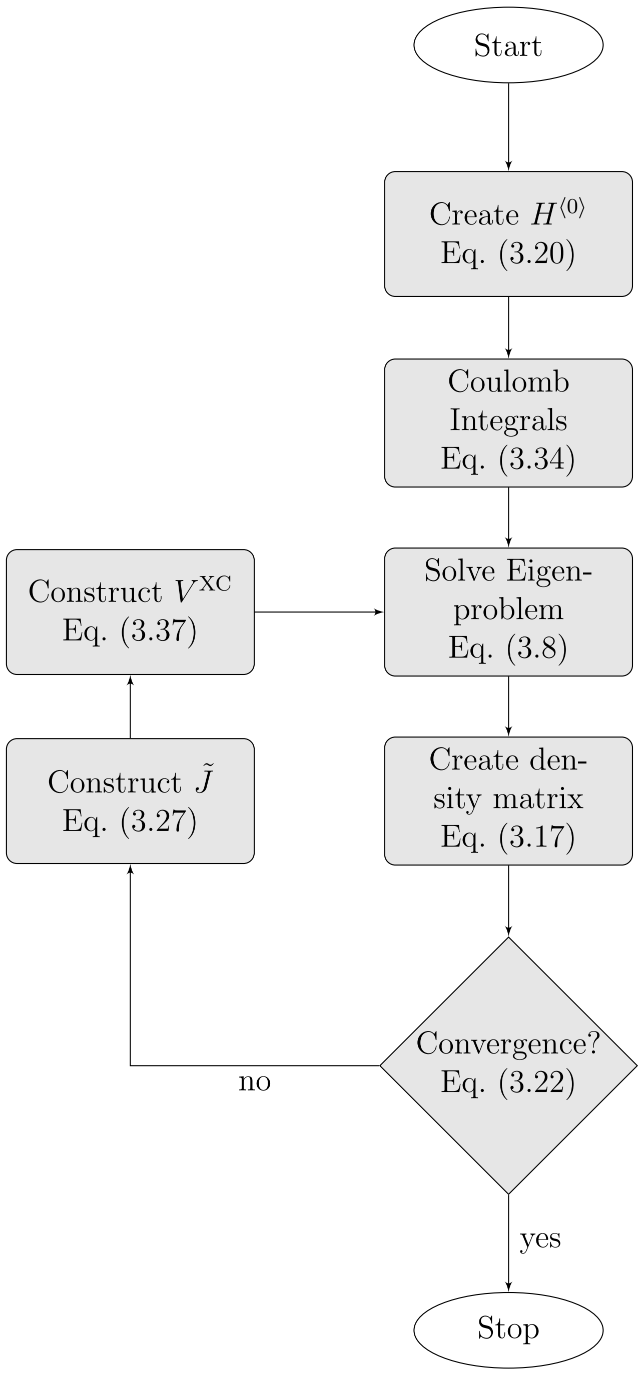


Figure 3.1.: Flowchart of the overall SCF algorithm in its simplest form. Techniques which improve the convergence are not considered here.

The fourth quantum number, the *spin projection quantum number* m_s, does not have any impact on the form of the orbitals. $Y_{lm}(\vec{r})$ is a real solid spherical harmonic, whose form is determined by l and m. The exponential expression $e^{-\alpha r}$ is the actual Slater function, whose shape is completely determined by α: small values ($\alpha \ll 1$) result in diffusive functions, whereas big values ($\alpha \gg 1$) result in a more compact shape. STOs are implemented e.g. in the DFT codes *Siesta* [25] or *Amsterdam Density Functional (ADF)* [26], which is however rather an exception because they imply a major computational disadvantage: the numerous integrals, which have to be computed to establish the Hamilton matrix H (especially for the Coulomb contribution J), can not be solved analytically and require, thus, expensive numerical treatment.

A much more common approach is to employ Gauss-function based orbitals: *Gaussian type orbitals (GTO)* of the common form

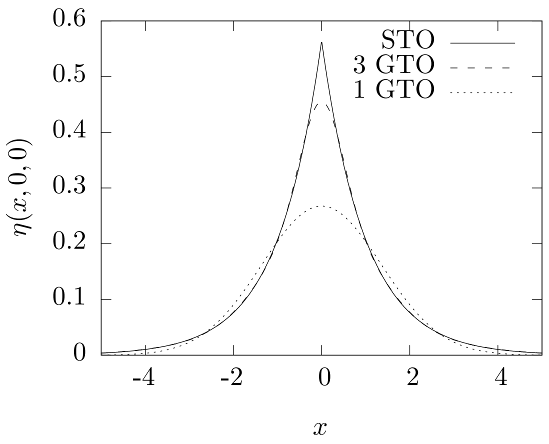
$$\eta_{nlm}^{\mathrm{GTO}} = N x^l y^m z^n e^{-\alpha r^2} . \tag{3.24}$$

GTOs have the major advantage that integral expressions as in Equation (3.16) can be evaluated either by use of analytic formulas or by relatively simple algorithms, depending on the form of the subshell. This can speed up the integral evaluations, necessary for the Coulomb contribution J, by several orders of magnitude, compared to the numerical evaluation necessary for STOs, while avoiding undesired numerical noise. However, compared to STOs, single GTOs provide a rather poor approximation to the actual orbitals φ_i: close to the nucleus ($r \to 0$) a GTO does not have the characteristic singular cusp, and its carriers ($r \to \infty$) fall of too quickly. To achieve better, Gauss-based, basis functions, the single Gaussian in Equation (3.24) is replaced by a superposition of several Gaussians:

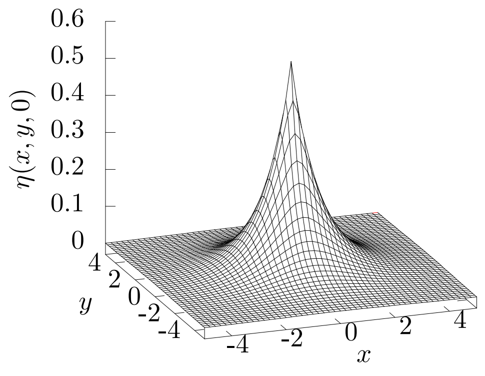
$$\eta_{nlm}^{\mathrm{CGTO}} = \sum_{i}^{P} a_i \eta_{nlm}^{\mathrm{GTO}} . \tag{3.25}$$

The coefficients $\{a_i, \alpha_i\}$ are usually "fitted" to a Slater function representing the according orbital, e.g. by the *least squares* method, explained in detail in Chapter 15 of [27]. For a comparison of Slater- and Gauss-orbitals, see Figure 3.2. This kind of basis function is commonly referred to as *contracted Gaussian type orbital (CGTO)*. In practice, typically between three and six *primitive* Gaussians are used for a feasible approximation. Compared to Slater type orbitals, this of course increases the total number of (primitive) basis functions, however the computational advantages outbalance these additional costs such that (C)GTO is by far the predominant basis function type implemented in existing LCAO-based chemistry codes. In this case, LCAO should correctly be termed *linear combination of Gaussian-type orbitals (LCGTO)*.

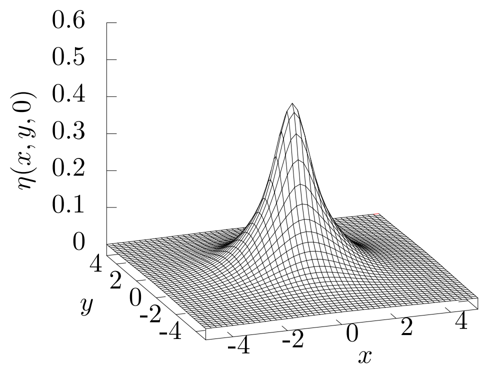
A typical basis set for practical applications contains at least one basis function per occupied shell. Often, several more bases for unoccupied valence shells are employed, for more variational flexibility. For example, the DFT-based study reported in [28] uses a total of 13 905 contracted Gaussian-type basis functions for the palladium metal cluster



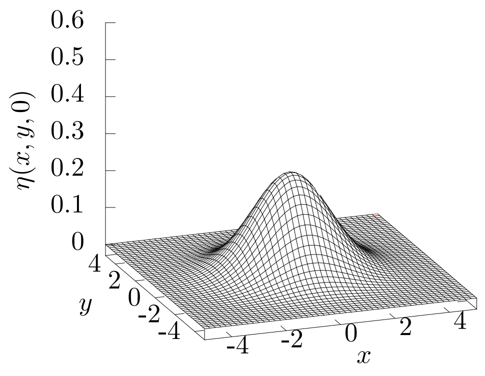
(a) 1D manifold of a STO, a primitive GTO and a superposition of three GTOs on the x-axis (y=z=0)



(b) 2D manifold of a STO on the x-y-plane (z=0)



(c) 2D manifold of a contraction of three GTOs on the x-y-plane (z=0)



(d) 2D manifold of a primitive GTO on the x-y-plane (z=0)

Figure 3.2.: One- and two-dimensional graphs of Slater- and (contracted) Gauss type orbital functions, centered at the Cartesian origin. The functions model an s-orbital in the first shell ($n = 1$). As the graphs demonstrate, a single (primitive) Gauss function (1 GTO) gives a rather poor approximation of a Slater function, especially close to the nucleus ($x, y \to 0$). A contraction of three Gauss functions (3 GTO) improves this approximation.

Pd_{309}. The 309 atoms comprise a total of 14 214 electrons in 7 107 occupied orbitals, hence, the calculation also included additional 6 798 unoccupied orbitals. The many-atomic systems, presented in this publication, had a *closed-shell* electronic structure, which means that the two possible electrons on each orbital (spin-up and spin-down) are represented by a single basis function. *Open-shell* calculations, which consider these electrons independently, result in a basis set of double size, which in turn results in two equally sized instances of the Hamilton- and overlap matrix. However, we can generally assume the number of required basis functions, L, to be proportional to the number of involved electrons, $L \propto N$.

3.1.4. The Coulomb Contribution J

Density Fit: Reducing the Computational Effort

As Section 3.1.1 shows, if J is to be computed exactly, the implied computational effort scales as $\mathcal{O}(L^4)$, which is the asymptotically most expensive step in the LCGTO ansatz. A common strategy to reduce this cost is to employ a density in the Hartree potential with a smaller (GTO) basis set, which is "fitted" to the density generated by the LCGTO orbitals. This technique is commonly referred to as *charge fit* or *density fit.* This section presents its basic machinery with a focus on its implementation. Fitting techniques are crucial for the computational efficiency of the LCGTO approach, as discussed in [29, 30]. Therein, Section 2 of [29] and Section 3 of [30] review the theory behind the charge fit technique. Furthermore, the latter reference introduces an error term which allows to estimate the accuracy of the established fitted potential.

To create a fitted density, an additional set of Gaussian-type basis functions ω_κ is introduced. Their linear combination provides an approximation to the LCGTO electron density ρ:

$$\tilde{\rho}(\vec{r}) = \sum_{\kappa}^{K} a_\kappa \omega_\kappa(\vec{r}) \approx \rho(\vec{r})\,. \tag{3.26}$$

(Please do not confuse ω_κ with the basis functions η_μ which model the orbital functions φ_i.) Substituted into Equation (3.13), the Coulomb contribution can be rewritten as

$$J_{\mu v} = \iint \frac{\eta_\mu(\vec{r})\eta_v(\vec{r})\rho(\vec{r}')}{|\vec{r}-\vec{r}'|} d\vec{r}d\vec{r}' \approx \sum_{\kappa}^{K} a_\kappa \iint \frac{\eta_\mu(\vec{r})\eta_v(\vec{r})\omega_\kappa(\vec{r}')}{|\vec{r}-\vec{r}'|} d\vec{r}d\vec{r}' = \tilde{J}_{\mu v}\,. \tag{3.27}$$

The integral expression in Equation (3.27) is also commonly referred to as *three-center integrals.* This method reduces total number of integral evaluations to $\mathcal{O}(KL^2)$, compared to $\mathcal{O}(L^4)$ if the complete set of four-center integrals is computed. Practically, the number of basis functions K has to be $2-3$ times larger than the number of basis functions of the LCGTO basis set L to give a good approximation [31]. Given the empiric assumption that the relation between K and L is a mere constant ($K \propto L$), the total number of integrals to be evaluated can also be stated as $\mathcal{O}(L^3)$.

As in the LCGTO approach, given a fixed set of basis functions, the variational freedom now lies in the expansion coefficients a_κ. The coefficients can be achieved by solving the system of linear equations

$$G\vec{a} = \vec{b}\,, \tag{3.28}$$

where the determinant matrix G is symmetric and positive definite. Its elements are defined as

$$G_{\kappa i} := \iint \frac{\omega_\kappa(\vec{r})\omega_i(\vec{r}')}{|\vec{r}-\vec{r}'|} d\vec{r}d\vec{r}'\,, \tag{3.29}$$

and the right hand side vector $\vec{b}$ as

$$b_\kappa := \iint \frac{\omega_\kappa(\vec{r})\rho(\vec{r}')}{|\vec{r}-\vec{r}'|} d\vec{r}d\vec{r}' = \sum_\lambda \sum_\sigma P_{\lambda\sigma} \iint \frac{\omega_\kappa(\vec{r})\eta_\lambda(\vec{r}')\eta_\sigma(\vec{r}')}{|\vec{r}-\vec{r}'|} d\vec{r}d\vec{r}'\,. \tag{3.30}$$

Finally, the solution vector $\vec{a}$ contains the expansion coefficients a_κ.

Cholesky Decomposition

This charge fit procedure is part of the iterative SCF procedure, consequently Equation (3.28) needs to be evaluated several times with right-hand sides $\vec{b}^{\langle k\rangle}$ changing over the iterations. Thus, it is computationally beneficial to reformulate the equation by factorizing G:

$$G = LL^T\,, \tag{3.31}$$

where L is a lower triangular matrix with strictly positive diagonal entries. This *Cholesky decomposition* is a common problem in linear algebra and is efficiently implemented in the numerical libraries *LAPACK* [32], *ScaLAPACK* [33], *PETSc* [34] and *PLAPACK* [35] (among others). Except for LAPACK, all packages provide parallel implementations. For numerical stability, it is common practice to compute an additional *permutation matrix*, which is however neglected here for brevity. In an SCF cycle, the vector $\vec{a}^{\langle k\rangle}$ is achieved by solving the two systems of linear equations

$$L\vec{y}^{\langle k\rangle} = \vec{b}^{\langle k\rangle}\,, \tag{3.32}$$

$$U\vec{a}^{\langle k\rangle} = \vec{y}^{\langle k\rangle}\,, \tag{3.33}$$

which can be done by cheap forward- and backward substitution, respectively. In this process, the decomposition in Equation (3.31) states the expensive step – common algorithms require $\mathcal{O}(K^3)$ operations. However, it needs to be executed only a single time, e.g. before the SCF procedure. The substitutions in Equations (3.32) and (3.33), to be solved in each SCF cycle, have an execution time of $\mathcal{O}(K^2)$.

Integral Computation

If the here presented simplifying scheme is applied, the construction of $\tilde{J}$ still formally involves the evaluation of $\mathcal{O}(L^3)$ integrals, which is typically also a major portion of the time, spent in an electronic structure calculation. To further reduce the execution time of this step, an additional accelerating technique is applied here:

The computation of the Coulomb contribution in an SCF cycle is given in Equation (3.27). The equation reveals that the variational freedom between the SCF cycles lies in the coefficient vector $\vec{a}^{\langle k \rangle}$, the basis functions and consequently the integral expressions

$$\iint \frac{\eta_\mu(\vec{r})\eta_v(\vec{r})\omega_\kappa(\vec{r}')}{|\vec{r}-\vec{r}'|} d\vec{r}d\vec{r}' \tag{3.34}$$

remain unchanged. Computationally, the integration is the most expensive step – once the integral is computed, Equation (3.27) reduces to a simple scalar multiplication. Thus, it is common practice to compute the integrals only a single time—usually before the SCF—and store the results in the memory. During the SCF cycle, the routine streams through the precomputed results and has to perform simple floating point multiplications. This approach can speed up the computation significantly, but requires additional memory of size $\mathcal{O}(L^3)$, which states the biggest data structure in the overall process of an LCGTO electronic structure computation.

Computational Costs and Parallelization

Here we summarize again the major steps and their computational costs, involved in the construction of the Coulomb contribution. Note that the costs refer to the complete matrix $\tilde{J}$, not just to a single element. Furthermore, we partition the steps into those which are done a single time before the SCF (*pre-SCF*), and those which have to be done more than once within the iterative scheme (*SCF*). Please recall that L represents the size of the basis set for the orbital functions, K the size of the basis set for the density fit, and N the number of involved electrons.

Pre-SCF:

- evaluate integrals, Eq. (3.34): $\mathcal{O}(KL^2)$ plus $\mathcal{O}(KL^2)$ memory requirement;
- construct G, Eq. (3.29): $\mathcal{O}(K^2)$;
- LU-factorization of G, Eq. (3.31): $\mathcal{O}(K^3)$.

SCF:

- construct $\vec{b}$, Eq. (3.30): $\mathcal{O}(KL^2)$;
- achieve $\vec{a}$, Eqs. (3.32) and (3.33): $\mathcal{O}(K^2)$;
- create $\tilde{J}$, Eq. (3.27): $\mathcal{O}(KL^2)$

One can generally assume the relation

$$K \propto L \propto N\,, \tag{3.35}$$

which allows to state the total computational costs of creating the Coulomb contribution as $\mathcal{O}(N^3)$, as well as the memory requirements as $\mathcal{O}(N^3)$.

The data structure, which holds the precomputed three-center integrals, states the only potential memory bottleneck in practical applications. Thus, for big problems, it is necessary to parallelize the data and distribute it over several compute nodes. ParaGauss partitions the integrals and, accordingly, the workload via the κ-index. Thus, Equation (3.27) becomes

$$\tilde{J}_{\mu v} = \sum_{p}^{P} \sum_{\kappa_p}^{K_p} a_{\kappa_p} \iint \frac{\eta_\mu(\vec{r})\eta_v(\vec{r})\omega_{\kappa_p}(\vec{r}')}{|\vec{r}-\vec{r}'|} d\vec{r}d\vec{r}'\,, \tag{3.36}$$

where P represents the total number of employed processes, p the process index, and K_p and κ_p the process-specific equivalents to K and κ, according to a chosen distribution scheme. Generally, any kind of partitioning can be applied, because there are no dependencies between the divided data.

3.1.5. The Exchange–Correlation Contribution V^{XC}

The importance of the exchange-correlation potential in density-functional methods has already been pointed out in Section 2.2.2. Here we present some important aspects of its computational treatment in practical LCAO codes. Its contribution to the Hamilton matrix is formally given by Equation (3.18). The equation contains the exchange-correlation potential V_{XC}, whose explicit form can vary significantly, depending on the type of application and modeled atomic system, and can usually not be integrated analytically. The integral expression requires, thus, a numerical quadrature scheme, which will be elaborated on in this section.

As a first step, Equation (3.18) is mapped to the general numerical quadrature scheme

$$V^{\text{XC}}_{\mu v} = \int \eta_\mu(\vec{r})V_{\text{XC}}(\vec{r})\eta_v(\vec{r})\; d\vec{r} \approx \sum_{k}^{N_g} w_k\; \eta_\mu(\vec{r_k})V_{\text{XC}}(\vec{r_k})\eta_v(\vec{r_k})\,, \tag{3.37}$$

where w_k are scalar quadrature weights, $\vec{r_k}$ are points in space where the integrand $V_{\text{XC}}(\vec{r})$ is evaluated, and N_g is the total number of evaluations. One can now employ a quadrature method, which distributes the vectors $\vec{r_k}$ over the observed domain and calculates the corresponding weights w_k. Generally, the goal of any quadrature scheme is to make an intelligent choice of the vectors, such to keep N_g is as small as possible while a certain accuracy criterion is still met.

Numerical integration is a well-studied field, and a great variety of efficient methods for all kinds of different function types can be found in the literature. Chapter 4 of the textbook [27] gives an excellent introduction. However, the shape of a typical XC-potential is somewhat "special": the shape of the potential $V_{\mathrm{XC}}(\vec{r})$ is mainly determined by the density $\rho(\vec{r})$, with characteristic cusps at the positions of the fixed nuclei (compare to the Slater type function in Figure 3.2b), and high variation in the area close to the nucleus. Thus, an efficient integration scheme would typically set the grid points in those areas to high resolution, while in areas between the nuclei, where V_{XC} is typically rather "plain" and does not exhibit a lot of variation, a sparser mesh is usually sufficient. In [36], A. D. Becke proposes such a scheme, which has proven successful over the last 20 years and is still the most commonly used integration method for XC-potentials in LCAO codes to date (with some consecutive modifications).

Its basic approach is to decompose the space into overlapping "cells", centered at the atomic nuclei, and to subsequently integrate each cell autonomously using polar coordinates with the nucleus serving as origin. However, as we will later see, the mesh-generation algorithm itself does also imply notable computational effort, as the profiling work in the course of this work showed, and can easily be parallelized. We will therefore give a brief overview of the employed integration scheme.

Following the notation of Becke, Equation (3.18) is replaced by the simple general form

$$I = \int F(\vec{r}) d\vec{r} \,. \tag{3.38}$$

Each of the M nuclei is assigned a relative weight function, $v_i(\vec{r})$, which defines the cell of nucleus i. Within the defined cell of this nucleus, the function yields (almost) *unity* ($v_i \to 1$), and $v_i \to 0$ inside the cell of other nuclei. All functions are normalized such that their sum is unity,

$$\sum_i^M v_i(\vec{r}) = 1 \,. \tag{3.39}$$

We will see later how these "cell functions" can be constructed. The functions are now used to decompose the integrand $F(\vec{r})$ into the *single-center components* $F_i(\vec{r})$:

$$F(\vec{r}) = \sum_i^M F_i(\vec{r}) = \sum_i^M v_i(\vec{r}) F(\vec{r}) \,. \tag{3.40}$$

This decomposition principally allows to integrate the single components independently, using a nucleus-centered integration scheme:

$$I_i = \int F_i(\vec{r}) d\vec{r} \,. \tag{3.41}$$

The total integral I is a sum of all sub-integrals,

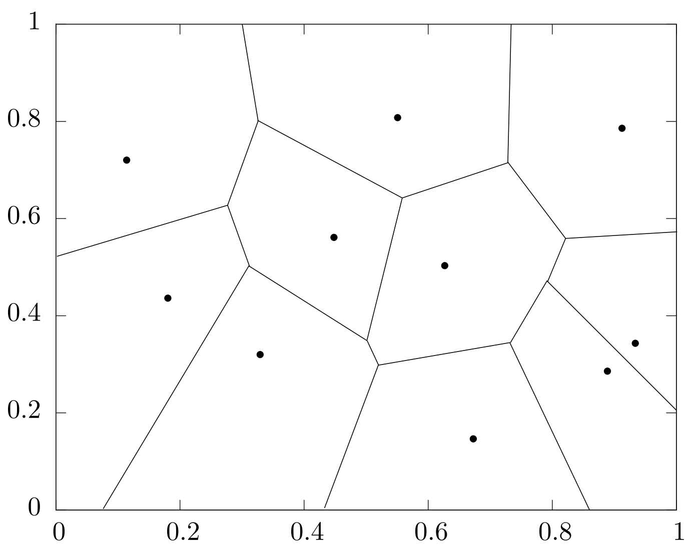


Figure 3.3.: A 2D Voronoi diagram of 10 sites, randomly sampled in the unit square.

$$I = \sum_{i}^{M} I_i \,. \tag{3.42}$$

However, one should note that for numerical reasons, the weight functions $w_i(\vec{r})$ need to be continuous and well-behaved. We will see later how this requirement can be met.

An important aspect of the Becke scheme is the decomposition of the three-dimensional molecular space into the above described cells. This decomposition step is realized by means of so-called *Voronoi polyhedra* or *Voronoi cells*. To give a brief explanation: given a set of objects in space (the "sites"), the Voronoi cell assigned to a site defines the a discrete sub-space, whose spatial points are closer to this site than to any of the others. Figure 3.3 shows a simplified two-dimensional plot of a Voronoi diagram. A special role play the points which lay exactly between two nuclei. Those points form a bisecting hyperplane, here referred to as the "face" between two nuclei. The cell of a nucleus is the space which is bordered by all facing planes around the nucleus. In two-dimensional space, these planes form a polygon, in three-dimensional space a polyhedron. The construction of a Voronoi diagram is a common problem in computer graphics, see [37] for more information.

In the Becke scheme, Voronoi cells are realized by employing weight functions $v(\vec{r})$. Therefore, consider the two-center coordinate (the two centers are two nuclei at $\vec{R}_i$ and $\vec{R}_j$)

$$\mu_{ij}(\vec{r}) = \frac{r_i - r_j}{R_{ij}}\,, \tag{3.43}$$

where r_i and r_j represent the distance of the point $\vec{r}$ from $\vec{R}_i$ and $\vec{R}_j$, respectively ($r_i = |\vec{r} - \vec{R}_i|$), and R_{ij} the distance between those two nuclei ($R_{ij} = |\vec{R}_i - \vec{R}_j|$). $\mu(\vec{r})$ is a hyperbolic function and can have values in the range $-1 \le \mu \le 1$: -1 at coordinate $\vec{R}_i$, and 1 at coordinate $\vec{R}_j$. Note also that the points in space where $\mu = 0$ indicate the face between the two nuclei. Furthermore, s defines a step function, which takes μ as input argument:

$$s(\mu_{ij}) = \begin{cases} 1, & -1 \le \mu_{ij} \le 0 \\ 0, & 0 < \mu_{ij} \le 1 \end{cases}\,. \tag{3.44}$$

The Voronoi polyhedron P_i for nucleus i is then constructed by the product of M step functions:

$$P_i(\vec{r}) = \prod_{i \neq j}^{M} s(\mu_{ij}(\vec{r}))\,. \tag{3.45}$$

So far, the functions P_i can principally be used as weight functions w_i, which decompose the space into discrete cells. Now, the “sharp” faces, imposed by the step function in Equation (3.44), are replaced by continuous and well-behaved weight functions. Thus, s is redefined as the alternative function

$$s(\mu) = \frac{1}{2}(1 - f(\mu))\,, \tag{3.46}$$

where f is a function with the properties

$$\begin{aligned} f(-1) &= -1, \\ f(1) &= 1, \\ \frac{df}{d\mu}(-1) = \frac{df}{d\mu}(1) &= 0\,. \end{aligned} \tag{3.47}$$

The simplest function satisfying these boundary conditions is the polynomial

$$p(\mu) = \frac{3}{2}\mu - \frac{1}{2}\mu^3\,, \tag{3.48}$$

which makes however a too smooth intersection. A steeper cutoff function can be obtained by iterations like

$$\begin{aligned} f_1(\mu) &= p(\mu), \\ f_2(\mu) &= p(p(\mu)), \\ f_3(\mu) &= p(p(p(\mu))), \\ &\dots\,, \end{aligned} \tag{3.49}$$

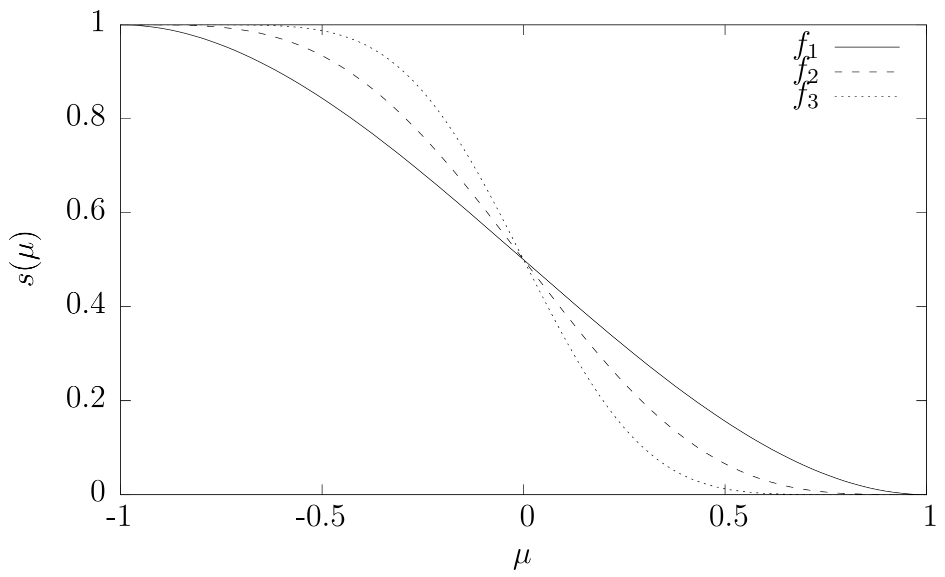


Figure 3.4.: One-dimensional plots of "smooth" cutoff functions between the Voronoi cells. With increasing polynomial degree (see Equation (3.49)), the functions become steeper and provide thus a "sharper" cutoff.

making f "sharper" with increasing index, see also Figure 3.4. Becke suggests three iterations for practical applications [36], which results in a polynomial of degree $3^3 = 27$.

As a last step, one has to ensure that the normalization constraint in Equation (3.39) is satisfied, which can be achieved by the following definition:

$$v_i(\vec{r}) = P_i(\vec{r}) / \sum_j^M P_j(\vec{r})\,. \tag{3.50}$$

So far, the space has been decomposed into independent, nucleus-centered cells by employing continuous, well-behaved weight functions. The single integral contributions I_i can now be numerically computed using standard integration schemes based on polar coordinates,

$$I_i = \int_0^\infty \int_0^\pi \int_0^{2\pi} F_i(r,\theta,\phi)\; r^2 \sin\theta \; dr \; d\theta \; d\phi\,. \tag{3.51}$$

The paper proposes furthermore, to proceed by directly applying Gauss-type quadrature formulas on spheres with different radii around the nucleus. For the 1-dimensional grid along the radius r, a *Gauss-Chebyshev* quadrature is suggested, however *Gaussian* for example employs a *Euler-McLaurin* scheme instead. For the angular part, θ and ϕ, there exist efficient grid distribution schemes from V. I. Lebedev, see [38, 39]. For more discussion on the numeric integration of the XC potential, we refer to [36] and references therein, as well as Section 7.4 of [12].

The integration scheme described above generates a set of grid points with corresponding weights, $\{(\vec{r}_k, w_r)\}$, which are used for the approximate evaluation of Equation (3.37). The weights w_r are a product of a cell weight $v_i(\vec{r}_k)$ and a weight determined by the quadrature scheme working on the corresponding cell.

Once the grid points and weights are generated, the quadrature scheme can be applied to the evaluation of the XC integral. Recall the definition of the exchange-correlation potential V_{XC}:

$$V_{\mathrm{XC}} = \frac{\delta E_{\mathrm{XC}}}{\delta \rho} = \frac{\partial f_{xc}}{\partial \rho} - \left(\nabla \cdot \frac{\partial f_{xc}}{\partial \nabla \rho} \right) . \tag{3.52}$$

Thus, the approximate XC–contribution from Equation (3.37) can be determined as

$$V_{\mu v}^{\mathrm{XC}} \approx \sum_{k}^{N_g} w_k \left[\eta_\mu(\vec{r}_k)\eta_v(\vec{r}_k)\frac{\partial f_{xc}}{\partial \rho}(\vec{r}_k) + \left(\frac{\partial f_{xc}}{\partial \nabla \rho}(\vec{r}_k) \cdot \nabla \eta_\mu(\vec{r}_k)\eta_v(\vec{r}_k) \right) \right] . \tag{3.53}$$

The total contribution can also be expressed in a practical matrix notation

$$V^{\mathrm{XC}} \approx \frac{1}{2} B a B^T + (\nabla_i B \cdot \vec{b}_i) B^T , \tag{3.54}$$

with the matrix

$$B_{\mu k} = \eta_\mu(\vec{r}_k) , \tag{3.55}$$

and three matrices for each spatial derivative

$$\nabla_i B_{\mu k} = \nabla_i \eta_\mu(\vec{r}_k) . \tag{3.56}$$

All matrices are usually dense and have the dimension $L \times N_g$. Finally, vector $\vec{a}$ and the three vectors $\vec{b}_i$ are defined as

$$a_k = w_k(\vec{r}_k) \cdot \frac{\partial f_{xc}}{\partial \rho}(\vec{r}_k) , \tag{3.57}$$

$$b_{ik} = w_k(\vec{r}_k) \cdot \frac{\partial f_{xc}}{\partial \nabla_i \rho}(\vec{r}_k) . \tag{3.58}$$

Computational Costs

There are two tasks which cause the main portion of the computational effort:

- each of the N_g Voronoi weights v_i requires M^2 steps: the construction of the polyhedron P carries out M function multiplications, and the weight v requires M polyhedrons, see Equations (3.45) and (3.50), respectively. Hence, the total execution time scales as $\mathcal{O}(N_g M^2)$;

- Equation (3.54) contains matrix-matrix multiplications, which can be done by invoking efficient BLAS level 3 routines. The asymptotic costs of a multiplication depend on the implemented multiplication algorithm. Most implementations use a straightforward algorithm with costs $\mathcal{O}(n^3)$, where n refers to the matrix dimension. There also exist more efficient algorithms, such as Strassen's Algorithm with execution time $\mathcal{O}(n^{\log_2 7})$, however those are rarely implemented. Therefore, we declare the costs of this step as $\mathcal{O}(N_g L^2)$.

One can generally assume that

$$N_g \propto L \propto N \tag{3.59}$$

and

$$N \geq M\,, \tag{3.60}$$

which allows the total computational costs to be stated as $\mathcal{O}(N^3)$. However, since N_g can be much larger than L in practical applications, the real time spent in the creation of the XC–contribution is usually the greatest fraction of the total time of an electronic structure calculation.

3.2. Symmetry Treatment

Molecules often show certain symmetry properties, which can be exploited to reduce the computational effort of electronic structure calculations significantly. A detailed discussion is given in [40, 41, 42], in this section we briefly reiterate their most important statements which lead to the computational problem discussed in Chapter 5.

In this technique, one proceeds by assigning the molecule of interest a *symmetry group*, described in the mathematical field of *group theory* (see the textbook [43] for an introduction). This subdivides the basis functions η_μ into groups of symmetry equivalent orbitals: the *irreducible representations (IRREPs)* Γ. Furthermore, the existing bases are transformed into delocalized, *symmetry-adapted linear combinations (SALCs)*:

$$\eta_\mu^\Gamma(\vec{r}) = \sum_i s_i^\Gamma \eta_i(\vec{r} - \vec{r}_i')\,. \tag{3.61}$$

These transformed functions have the property to be orthogonal between groups,

$$\int \eta^\Gamma(\vec{r})\eta^{\Gamma'}(\vec{r})d\vec{r} \propto \delta_{\Gamma\Gamma'}\,, \tag{3.62}$$

which divides the L-dimensional space, spanned by the original basis functions, into several sub-spaces. This irreducible orthogonality of course also affects the interactions comprised in the Hamilton- and overlap matrix,

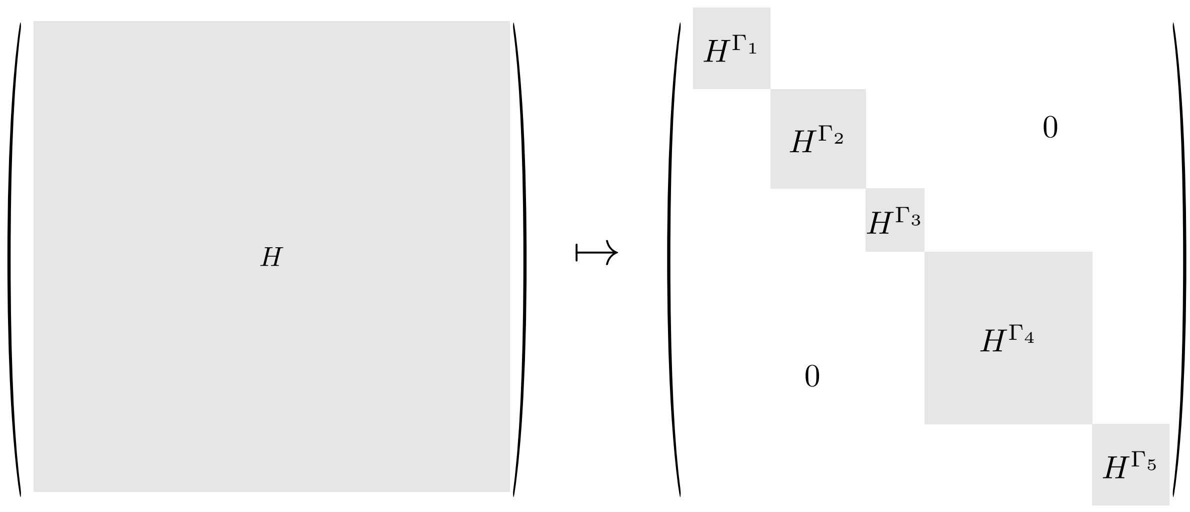


Figure 3.5.: Transformation of the dense Hamilton matrix into a block-diagonal form. Gray blocks are dense (sub-)matrices, white parts consist of zero-elements.

$$\left.\begin{aligned} H_{\mu v} &= \int \eta_\mu^\Gamma(\vec{r})\,\hat{\mathrm{f}}^{\mathrm{KS}}\,\eta_v^{\Gamma'}(\vec{r})\;d\vec{r} \\ S_{\mu v} &= \int \eta_\mu^\Gamma(\vec{r})\eta_v^{\Gamma'}(\vec{r})\;d\vec{r} \end{aligned}\right\} = 0 \quad \text{for} \quad \Gamma \neq \Gamma' \,, \tag{3.63}$$

which transforms the Hamiltonian into a block-diagonal matrix form:

$$\begin{aligned} H &= H^{\Gamma_1} \oplus H^{\Gamma_2} \oplus \cdots \oplus H^{\Gamma_n} \,, \\ S &= S^{\Gamma_1} \oplus S^{\Gamma_2} \oplus \cdots \oplus S^{\Gamma_n} \,, \end{aligned} \tag{3.64}$$

demonstrated in Figure 3.5. This transformation effectively divides the problem into n smaller, independent, sub-problems, which can speed up the creation of the Hamiltonian, as well as the computation of its eigenvectors, by factors of several magnitudes. However, this blocked form complicates the efficient *parallel* computation of the eigenvectors significantly. Chapter 5 discusses this issue in detail.

4. Parallel Linear Algebra Operations in Relativistic Transformations

The physical model, on which most ab-initio electronic structure calculations are based—the Schrödinger equation—, accounts for the energy of a quantum mechanical system due to particle movement and their electrostatic interactions. However, some chemical applications also require the consideration of relativistic effects, briefly introduced in Section 4.1, especially if heavy elements are involved. Their treatment in DFT codes, often implemented in *relativistic transformations*, implies computationally expensive linear algebra operations, resulting from complicated algebraic expressions, see Section 4.2. These operations include dense matrix multiplications and solutions of the generalized eigenvalue problem, which require $\mathcal{O}(N^3)$ time. Hence, relativistic transformations belong to the class of the computationally most expensive numerical problems typically appearing in electronic structure codes (see Chapter 3). There exists a sequential implementation in ParaGauss. However, its time-consuming operations in large-scale applications require parallel routines to avoid severe performance bottlenecks.

This chapter introduces a novel high-level Fortran interface to parallel matrix algebra. Thus, complicated algebraic expressions from relativistic transformations can be expressed directly in a parallel Fortran code, using a concise and comprehensible pseudo-mathematical syntax, while expensive linear algebra operations are executed in the back-end by performance-optimized parallel routines. Furthermore, the interface facilitates easy parallelization of the relativistic transformations, already implemented in ParaGauss. In particular, the parallelization of the existing code requires only few additional lines of code, while the actual mathematical semantics remain untouched. The interface specification is introduced in Section 4.3. This specification is accompanied by a library implementation, presented in Section 4.4. Finally, the approach is evaluated in Section 4.5 by means of the ease of parallelizing the existing relativistic transformations, resulting code quality, and parallel performance.

4.1. Relativistic Effects and their Treatment in Quantum Chemical Applications

Relativistic effects occur when particles, such as electrons, move at velocities close to the speed of light (see Appendix A). In the majority of chemical applications, these effects account only for a very minor contribution to the relevant physics. This gives rise to the fact that relativism does not appear in the standard underlying physical model – the Schrödinger equation in its original, non-relativistic, version, as presented in Chapter 2.1 . However, the situation changes in heavy elements with high atomic numbers, occurring in the later part of the periodic table: electrons close to the nucleus are exposed a strong electrostatic field, which causes them to move at very high velocities. For example, in the gold atom Au, the average speed of the electrons on the innermost shell 1 s is about 60% of the speed of light [40]. As a consequence, the *relativistic mass* of such an electron is higher than the standard electron mass m_e by about 25%, which in turn reduces its average distance to the nucleus. This effect potentially also impacts s- and p-electrons of higher principal quantum numbers, which are, on the contrary, less bound to the nucleus as a result. This includes valence electrons, which are especially important for chemical applications. See the references [40, 41, 44] for an in-depth discussion on relativism in quantum chemical applications.

A wave-function based physical model, which also accounts for relativistic effects, provides the *Dirac equation* [45]. This equation can be seen as an extended Schrödinger equation. The relativistic extensions therein have been incorporated in the Hohenberg–Kohn theorems and the Kohn–Sham equations, introduced in Section 2.2.2, as a relativistic version of density functional theory, see [40]. In these *Dirac–Kohn–Sham (DKS)* equations,

$$h^{(4)}_{\mathrm{DKS}} \varphi^{(4)}_i = \varepsilon_i \varphi^{(4)}_i \,, \tag{4.1}$$

the DKS–Hamiltonian $h^{(4)}_{\mathrm{DKS}}$ is a 4×4 operator (indicated by the $^{(4)}$-superscript), $\varphi^{(4)}_i$ its eigenfunctions, and ε_i the corresponding eigenvalues. In the DKS–Hamiltonian,

$$h^{(4)}_{\mathrm{DKS}} = H^{(4)}_{\mathrm{kin,rel}} + V_{\mathrm{ne}} + V_{\mathrm{coul}} + V_{\mathrm{XC}} \,, \tag{4.2}$$

only the kinetic energy operator $H^{(4)}_{\mathrm{kin,rel}}$ has a nontrivial *four-component* structure. The remaining terms resemble the electrostatic potential in the “classic” KS–formalism (compare to Equation 2.34). In a complete relativistic description, the exchange-correlation potential, V_{XC}, also requires adaption. However, these contributions are usually of little importance for the majority of chemical applications and are, thus, often neglected.

A popular approach to implement the formalism introduced in Equation (4.1) is to use a common non-relativistic solution, and augment it with approximate relativistic corrections. In this way, well-established electronic structure codes can be used with some modifications, while the most essential physics due to relativistic effects are captured in the final result. The theoretical basis for these corrections is given by the *Douglas–Kroll (DK)* formalism [46], which facilitates the construction of an approximate, relativistic two-component Kohn–Sham operator $\hat{\mathrm{f}}^{\mathrm{KS}}$. There are two approaches available to this problem [47], however here we discuss only the one implemented in ParaGauss [40, 41]. Therefore, a series of *DK–transformations* are applied to the original DKS–Hamiltonian, resulting in an effectively relativistic Hamiltonian. The *Douglas–Kroll–Hess (DKH)* method [48] provides a practical formulation of Douglas–Kroll, supporting discrete, matrix-based, operators. The method allows to establish a *DKH–Hamiltonian* up to any order and, thus, accuracy [49]. For most chemical applications, a second- or third order transformation provides sufficiently accurate results, however, transformations up to the fourteenth order on chemical benchmark systems have been reported [49, 50]. In this work, these DKH–transformations are also referred to as *relativistic transformations*. Relativistic transformations imply complicated algebraic expressions containing scalars and diagonal- and dense matrices, as well as dense eigenvalue problems. The complexity of the algebraic expressions increases with the order of the transformation, thus implying not only additional computational effort, but also increased code complexity. See [41] for a formal introduction to Douglas–Kroll and Douglas–Kroll–Hess.

In the Dirac formalism, relativistic effects primarily impact the kinetic energy operator, see Equation (4.2). However, the DK–transformations effect all operators comprised in the DK–Hamiltonian. For many chemical applications, a “simplified” DK–Hamiltonian gives sufficiently accurate results [40, 51, 52]: here, one only applies the transformations to the operators $H^{(4)}_{\mathrm{kin,rel}}$ and V_{ne}. The remaining operators, V_{coul} and V_{XC}, are added *a posteriori*. The work of A. V. Matveev and N. Rösch [53] introduces a formalism which goes beyond this theory, including also the Coulomb potential V_{coul} into the DK–transformations. This improves the accuracy of relativistic approximations and, thus, the spectrum of possible chemical applications. Furthermore, the work of A. V. Matveev [41] presents the implementation of the resulting DKH–transformations into ParaGauss, employing sequential BLAS- and LAPACK routines [32] for computational efficiency. However, as we will see in Section 4.2, the transformations involve expensive $\mathcal{O}(N^3)$ operations, such as dense matrix multiplications and eigenvalue problems, which makes an efficient treatment necessary. One goal of the work presented in this chapter is to augment the approach by Matveev, by using *parallel* routines from the PBLAS- and ScaLAPACK libraries [33], aiming thus at a higher (parallel) performance in large-scale applications.

4.2. Relativistic Transformations in ParaGauss

For the construction of relativistic versions of matrix operators, ParaGauss requires four input matrices:

- the overlap matrix S;
- the kinetic energy matrix T;
- two matrices, which contain potential terms, V and O.

These matrices generally correspond to the single components of the Hamiltonian, introduced in Section 3.1. Hence, they are dense and symmetric, but their dimension is typically by several factors greater: the accuracy of the relativistic transformations strongly depends on the size and flexibility of the underlying finite basis. The common way to achieve this is to carry out the transformations in a yet *uncontracted* Gaussian-type basis (see Section 3.1.3), and contract the matrices afterward. As also mentioned in Section 3.1.3, a contracted basis function typically consists of three to six primitive Gaussians, which corresponds to the factor by which the matrices S, T, V and O are greater than the dimension of the Hamilton matrix H, L. We will refer to this "uncontracted dimension" as L_u.

A relativistic transformation proceeds in several steps. The first task is to solve the generalized eigenvalue problem

$$TU = SUt\,, \tag{4.3}$$

where T, S and U are dense symmetric matrices, and t is a diagonal matrix, such that $U^TTU = t$ and $U^TSU = 1$. The problem-specific function,

$$t_\mathrm{rel}, e, a, b, r = \mathrm{factors}(2t)\,, \tag{4.4}$$

computes relativistic factors from the kinetic energy eigenvalues, stored on the diagonal of t. All resulting factors are diagonal matrices of the same dimension as t, and required later by the actual transformations.

The second major step is to transform the basis of the potential matrices, V and O, into *momentum space*:

$$\tilde{V} = U^TVU, \quad \tilde{O} = U^TOU\,. \tag{4.5}$$

This representation of the DK–operators entails some beneficial properties: some of the (intermediate) matrices, which are dense in real space, have a diagonal representation in momentum space. Furthermore, the algebraic expressions, which transform the involved matrix operators into their relativistic forms, are generally more concise. This reduces the implied computational effort, which is especially important when L_u is large.

A *second order* transformation of the non-relativistic potential matrices V and O into their relativistic counter-part states the following equation:

$$\tilde{V}_\mathrm{rel} = a\tilde{V}a + b\tilde{O}b + R^Ter^{-2}R + (eR^Tr^{-2}R + R^Tr^{-2}Re)/2\,. \tag{4.6}$$

Here, the intermediate matrix R is defined as

$$R = \mathrm{rpt}(e, r^2a\tilde{V}a - b\tilde{O}b)\,, \tag{4.7}$$

with the matrix-valued function

$$\mathrm{rpt} : (e, X) \to Y, \quad Y_{mn} = X_{mn}/(e_m + e_n)\,. \tag{4.8}$$

```
    ! kind of double precision numbers:
    integer, parameter :: DP = kind(1.0d0)
    real(DP) :: A(n, n), B(n, n)
    ...
    A = A + 2 * B
    A = matmul(A, B - 2 * A)
```

Figure 4.1.: A Fortran 90 code snippet which shows simple matrix arithmetics. One can see that the use of dedicated operators results in concise code, which clearly exhibits the mathematical semantics.

The final step is a back-transformation of the relativistic operators into real space:

$$T_{\mathrm{rel}} = U^{-T} t_{\mathrm{rel}} U^{-1}, \quad V_{\mathrm{rel}} = U^{-T} \tilde{V}_{\mathrm{rel}} U^{-1} \,. \tag{4.9}$$

This second order transformation is the most popular form of DKH transformations in electronic structure codes. An alternative class of arbitrarily accurate implementations of a relativistic transformation is given by an iterative scheme with more compact expressions. A crucial step is the recurrent evaluation of the intermediate matrix $X^{(n)}$:

$$X^{(n+1)} = \mathrm{rpt}(e, O - X^{(n)} E_{11} + E_{22} X^{(n)} - X^{(n)} O^T X^{(n)}) \,, \tag{4.10}$$

with E_{11} and E_{22} being also dense matrices. We will address this scheme only briefly in this work, for the complete formalism see the parallel implementation in Figure 4.9 and the publication [54].

Sequential Implementation in Fortran

From the revision 90 on, the Fortran standard provides an intrinsic programming model, which allows to formulate basic matrix-algebra in a pseudo-mathematical notation. Here, scalars, vectors and matrices are represented by numeric variables, and one- and two-dimensional arrays, respectively. Operations between these data-objects can be expressed by the general form

[operand1] operator operand2 ,

using the common operators `+`, `-`, `*`, `/`, and `**`. The operators `+` and `-` can be used either as unary or binary operators. Alternatively, these operations can also be expressed as calls to intrinsic subroutines, such as `matmul(A,B)`. Figure 4.1 shows an example. For more details, please refer to the Fortran standard [55].

In ParaGauss, the relativistic transformations have been implemented using the Fortran "operator syntax" described in the previous paragraph, see Section 2.2.2 of [41]. The use of this technique allows a short and concise representation of the transformations in the source code, see Figure 4.2 for an example Fortran 90 implementation of a second order transformation. As the figure shows, the algebraic expressions are almost equal to those given in the introduction of the DKH–formalism at the beginning of this section. The implementation of the relativistic transformation is very concise, revealing clearly the intended mathematical semantics. At the same time, the compiler has the opportunity to efficiently implement the implied numerical operations, e.g. by linking to optimized BLAS routines [56].

The second order DKH–transformation introduced here involves several matrix-matrix multiplications, as well as the solution of a generalized eigenvalue problem (among other, less expensive, linear algebra operations). Formally, these operations require $\mathcal{O}(L_{\mathrm{u}}^3) = \mathcal{O}(N^3)$ steps. Thus, the relativistic transformations belong to the same, most expensive, computational category as the construction of the Coulomb[1]– and Exchange–Correlation contribution (see Sections 3.1.4 and 3.1.5, respectively), as well as the solution of the generalized Eigenvalue problem (see Chapter 5). The implementation presented in Figure 4.2, as realized in ParaGauss, relies on the numerical routines provided by the employed, sequential, compiler. However, the implied computational costs require parallelization of this expensive step, to avoid bottlenecks in large applications. Furthermore, the involved matrix data structures occupy $\mathcal{O}(L_{\mathrm{u}}^2) = \mathcal{O}(N^2)$ space. They are usually required before the SCF, when the transformations are applied, and after the SCF. Thus, to avoid memory bottlenecks on single nodes, it is common practice in ParaGauss to buffer the matrices on disc. As the matrices can become large, with their dimension L_{u} being up to several thousand, the involved I/O operations also state a potential bottleneck. The rest of this chapter presents a parallelization technique, which overcomes these problems, while requiring only minimal changes to the original code.

4.3. A Fortran Interface to Parallel Matrix Algebra

The core contribution of this chapter is a novel Fortran interface, introduced in this section, and its implementation to parallel matrix algebra, presented in Section 4.4. The main motivation to develop this technique was to parallelize relativistic transformations, implemented in ParaGauss and introduced in the previous section. We designed a parallel library interface, which resembles the original, sequential, Fortran operator syntax as good as possible. It augments the intrinsic Fortran arrays and the corresponding operator syntax with distributed data types as abstract matrix representations, and parallel routines. In this way, the existing implementation in ParaGauss can be reused, while only a few additional lines of code are necessary. After these changes, the expensive linear algebra operations (mainly the matrix products and the generalized eigenvalue problems) are executed by parallel routines. Furthermore, the distributed data objects can be used to conveniently distribute the matrix structures over the existing compute

[1]Given the *density fit* technique is applied, see Section 3.1.4.

```fortran
1   subroutine reltrans(S, T, V, O, T_rel, V_rel)
2    implicit none
3    real(DP), intent(in)  :: S(:,:), T(:,:), V(:,:), O(:,:)
4    real(DP), intent(out) :: T_rel(:,:), V_rel(:,:)
5    real(DP) :: U, U_inv, R, V_mom, O_mom, aVa, aOa
6    type(rdmatrix) :: td, td_rel, e, a, b, r2
7
8    ! allocate the intermediate variables
9    ! U, td, td_rel, e, a, b, r2
10   allocate( U(size(S,1), U(size(S,2)) )
11   ...
12
13   ! call the generalized eigenvalue solver
14   call geigs(T, S, td, U)
15
16   ! call the internal routine to compute the relativistic factors
17   call factors(2.0d0 * td, td_rel, e, a, b, r2)
18
19   ! transformation into momentum space
20   V_mom = tr(U) * V * U
21   O_mom = tr(U) * O * U
22
23   ! intermediate results
24   aVa = a * V * a
25   bOb = b * O * b
26   R = rpt(e, r2 * aVa - bOb)
27   W22 = tr(RW) * (0.5d0 * r2**(-1)) * RW
28   U_inv = tr(U) * S
29
30   ! transformation of V and O into V_rel
31   V_rel = aVa + bOb + tr(RW) * (e * r2**(-1)) * RW + e * W22 + W22 * e
32
33   !back-transformation into real space
34   T_rel = tr(U) * t_rel * U_inv
35   V_rel = tr(U) * V_rel * U_inv
36  end subroutine reltrans
```

Figure 4.2.: Fortran 90 implementation of a second order relativistic transformation. The implementations of the functions `geigs`, `factors`, and `rpt` are omitted for brevity, as well as the explicit array allocation, indicated by lines 8 to 11. The function `tr(A)` returns the transpose of the matrix array `A`. The specific data type `rdmatrix` represents diagonal matrix. The `*`-operator is overridden by the intrinsic routine `matmul` for the dense matrix product, and by a manually implemented routine for the product of a dense- and a diagonal matrix.

nodes. Thus, memory bottlenecks can be compensated by increasing the *global memory space* (i.e. the accumulated memory of all compute nodes), which can be achieved by rising the number of employed nodes. This ensures an efficient use of the available memory, and avoids slow I/O operations.

However, though this work was motivated by a specific problem, we would like to emphasize that the developed library is not limited to this case. Scientific codes often contain linear algebra expressions, such as vector- and matrix additions and multiplications, systems of linear equations or eigenvalue problems. It is common practice to rely on abstract matrix representations and appropriate operators, which model linear algebra characteristics. Such abstractions facilitate a *separation of concerns*, where mathematics is separated from technical implementation details. It allows a quick implementation of matrix- and vector transformations, which contributes to the productivity of the development of numerical routines. Furthermore, as these transformations have the appearance of mathematical expressions, the application semantics are easily comprehensible, hence, the readability of the code is improved.

There exist several languages or library extensions which provide such a functionality. *Matlab* and *Octave* are high-level scripting languages, which operate mainly with mathematical objects, but are usually not suitable for high performance codes and big software projects. The C++ libraries *Armadillo* [57], *uBLAS* (as part of *BOOST* [58]) and *MTL* [59], among others, provide template-based matrix classes with comprehensive functionality, and partially also advanced linear algebra operations, such as factorizations or eigenvalue problems. As already mentioned, *Fortran* provides an intrinsic programming model, which has been applied for the implementation of relativistic transformations in ParaGauss, see Section 4.2. The *Matran* library [60] provides further matrix functionality for Fortran, together with advanced operations. However, except for a commercial version of the MTL (called "*Supercomputing Edition*"), software or literature about abstractions supporting parallelism and data distribution, especially for Fortran, is difficult to find. Our interface specification, introduced in the next Section, aims to fill this gap and provide a high-level matrix abstraction for Fortran, with an opaque support for distributed memory parallelism.

Before introducing the actual application programming interface, we present some of the criteria, based on which the API was designed.

4.3.1. Design Criteria

Separation of concerns. The high-level mathematical programming model, used to express the relativistic transformations (see Figure 4.2), entails several benefits concerning programming efficiency and code readability. Mathematical expressions can be typed almost directly into the source code. This supports a *separation of concerns*: the mathematical semantics are separated from technical implementation details. Thus, algebraic expressions can be implemented quickly, while the resulting code is clear, concise, and easily comprehensible. With our library API, this separation of concerns shall be retained: the library itself is rather seen as an additional, intermediate, layer between the code and the compiler. If used correctly, this layer is invisible in the actual program semantics.

Re-usability of existing code. As already mentioned, the main motivation behind this interface was to provide a parallel solution to the already implemented relativistic transformations. We want to achieve this parallelization step with no or only minor changes to the original code. One attribute to achieve this is to design the syntax of this matrix abstraction as close to the original Fortran 90 syntax as possible. Another attribute is the SIMD programming model, which is introduced in Section 4.3.2.

Distributed data and easy data management. Within this specification, we refer to the data structures, which represent mathematical objects such as matrices as *data objects.* In the original Fortran syntax, these objects are represented by standard arrays. This API defines a new set of abstract matrix data objects, which refers to data, physically distributed over the address spaces of the involved processes. One important requirement is that the management of these objects shall be as easy as possible. Low-level tasks, such as the physical distribution of the data, are hidden from the user and be executed automatically in the background. Thus, *separation of concerns*—a main design aspect—is facilitated.

Parallel performance. The relativistic transformations contain compute-intensive tasks, especially matrix multiplications and generalized eigenvalue problems. Thus, high parallel performance is indispensable. For the execution of these expensive tasks, the incorporation of external, optimized, numerical libraries must be facilitated.

Technical dependencies. The interface must be implementable using standard techniques for parallel codes: a Fortran compiler in combination with an MPI library [11]. However, an implementation is not limited to these technologies – as mentioned in the previous paragraph, the use of additional libraries, e.g. high performance numerical libraries, is desired.

4.3.2. SIMD Programming Model

Today, distributed memory architectures are most commonly programmed using a *message-passing* paradigm, standardized e.g. in the widely used *Message-Passing Interface (MPI)* [11]. Different from our approach, the MPI implements the *Single Program Multiple Data (SPMD)* programming model: as the name already indicates, in this model, all involved processes execute the same program on different data. However, in those programs, it is common practice to distinct between processes. For example, in the most basic communication pattern, the *send-receive* operation, one process sends a message, another process receives it. Obviously, at the time of this operation, both processes execute different *branches* of the code. Another example is the *master-worker model*, frequently implemented in MPI codes: a dedicated process—the master—sends data to the worker processes, which independently work on the data and send the results back to the master. Also here, the master and the workers execute different branches of the code. This SPMD programming style allows to express very detailed communication patterns, offering thus a lot of space for optimizations. At the other hand, a lot of branches can result in codes which are large, confusing, and difficult to maintain. Furthermore, the parallelization of an existing, sequential, code typically requires a lot of refactorization.

Here we propose a different, more strict, approach. Different from MPI, the distributed data object API implements the *Single Instruction Multiple Data (SIMD)* programming model [61] on the code level: all routines and operators are collective, i.e. they must be called by all involved processes, and in the same sequence. A distinction between the processes is not possible, as it is often the case in message-passing codes. Instead, the resulting code corresponds to a sequential code, parallelism is achieved by using parallel data objects. In comparison with a typical message-passing code, this approach entails some advantages:

- existing code can be adopted with very few changes;
- less code is necessary;
- the program semantics can be expressed deterministically, hence coherence problems can be avoided;
- the resulting code is clearer and more concise, hence the readability is improved.

A downside of this approach is that the user has few control over the physical data distribution – he has to rely on the capabilities of the library implementation. In our implementation (see Section 4.4), we choose a homogeneous block-cyclic data distribution. If, for example, the employed parallel computer had a heterogeneous architecture, this distribution scheme might result in load-imbalance. In this case, a different, architecture-specific, implementation would be necessary.

Another downside of the SIMD programming model is that independent operations can not be executed simultaneously, e.g. to achieve a better parallel scalability. However, our API does still allow a certain degree of concurrency between the operations, by employing several communication contexts. Section 4.4 gives more details on this issue.

4.3.3. Application Programming Interface

Here we introduce the application programming interface (API) specification. It consists of a collection of

- parallel data objects (“data types” in the Fortran syntax), representing mathematical operands, and a supporting routine for technical reasons;

- support routines, necessary to manage the data objects (e.g. the physical data distribution);
- operators and subroutines, which carry out parallel linear algebra operations on or between the data objects.

We chose this set to solve our specific problem from relativistic quantum chemistry, introduced in Section 4.2. As a consequence, "specialized" operands, such as diagonal matrices, are also part of the specifications, whereas other, more common, linear algebra operands, such as vectors, do not appear. The interface, as presented here, is not intended to be complete for any problem at hand. However, we emphasize that this is not a limitation, as the set of data objects to any other mathematical objects (subject to a feasible technical realization).

Data Types

We define one auxiliary data type (*communication object*), and two data objects, representing mathematical operands:

- `rmatrix` (data object): represents a distributed real dense matrix.
- `rdmatrix` (data object): represents a distributed real diagonal matrix.
- `pm_ctxt` (communication object): the communication context. It represents a set of processes, over which the data is distributed, and which are included into parallel operations. Each data object needs to be assigned one context, and two data objects are only compatible for operations if they are assigned the same context.

Furthermore, the relativistic transformations contain scalar numbers. Here, a special, distributed, type is not necessary – we simply utilize a standard (non-distributed) variable of type `real`.

The data- and communication objects are Fortran data types, so variables are declared by the standard `type` keyword. Before the declared variables can be used, they have to be initialized (one could also refer to this as "instantiation"). This is done by the auxiliary routines, introduced in the next section. See also the example in Figure 4.3.

Auxiliary Routines

The auxiliary routines provide the basic functionality, necessary for the organization of the data types introduced in the previous section. In particular, the following three routines are declared:

`context =` **pm_create_ctxt**`( system_comm )`

Arg. name	direction	type	description:
`system_comm`	IN	`integer`	A system communicator
`context`	RET	`pm_ctxt`	A communication object

This function constructs a communication object ("`context`"). It requires a system communicator – in our MPI-based implementation an MPI communicator, such as `MPI_COMM_WORLD`. With the help of the MPI grouping functionality, the user can determine, which processes are associated with a data object, and thus involved in its parallel operations.

`pA =` **matrix**`( sA[, context] )`

Arg. name	Direction	Type	Description
`sA`	IN	`real(:)/(:,:)`	A non-distributed array
`context`	IN	`context`	(optional) A communication object
`pA`	RET	`r(d)matrix`	A distributed (diagonal) matrix

This function handles the transitions from intrinsic Fortran arrays to distributed data objects. It requires a real, sequential, array (indicated by `sA`) as input argument. An optional input argument is a communication object (`context`). If this argument is omitted, the routine uses a standard communication object, which represents all processes – technically, this would refer to the MPI communicator `MPI_COMM_WORLD`. From these input data, it creates a distributed matrix object (`pA`). The function is overloaded, so dependent on the input type (one- or two-dimensional array), it returns a diagonal- or dense distributed matrix, respectively. An important aspect is that the matrix objects are only interoperable if they are created with the same communication object.

`sA =` **array**`( pA )`

Arg. name	Direction	Type	Description
`pA`	IN	`r(d)matrix`	Distributed (diagonal) matrix
`sA`	RET	`real(:)/(:,:)`	Non-distributed output array

This function handles the reverse transition of the function `matrix`: it converts a distributed data object into an intrinsic Fortran array. It requires a data object of type `rdmatrix` or `rmatrix` as input argument, and returns a one- or two-dimensional array, respectively, of type `real`.

Operators

Fortran defines a fixed set of unary and binary operators, to be applied on intrinsic data types. For the relativistic transformations, we override some of them, to make them compatible with the distributed data types (symbol):

- +: addition, binary
- −: binary subtraction, unary negation
- *: matrix multiplication
- **: exponentiation

Furthermore, we define one function and one subroutine:

`AT = tr(A)`

Arg. name	Direction	Type	Description
`A`	IN	`rmatrix`	A distributed dense matrix
`AT`	RET	`rmatrix`	The transpose of `A`

This function returns the transpose of a distributed dense matrix object, given as input argument. It is only defined for data objects of type `rmatrix`, the usage with diagonal matrices of type `rdmatrix` is undefined.

`geigs(A, B, E, C)`

Arg. name	Direction	Type	Description
`A`	IN	`rmatrix`	The first dense input matrix
`B`	IN	`rmatrix`	The second dense input matrix
`E`	OUT	`rdmatrix`	The matrix containing the eigenvalues
`C`	OUT	`rmatrix`	The matrix containing the eigenvectors

This routine solves the dense generalized eigenvalue problem $AC = BCE$. The computed eigenvalues are stored on the diagonal of E, so the data type of this output argument is a distributed diagonal matrix. The eigenvectors are stored as columns of the distributed dense matrix C.

Example

Figure 4.3 gives a simple example of how the API can be used to execute parallel matrix algebra. Note that the algebraic expressions in Subroutine `arithmetics1` could also be expressed using the intrinsic Fortran operator syntax. A parallelization of this routines only requires few changes in the code – see therefore also the caption of Figure 4.3. The second example, shown in Figure 4.4, demonstrates how two different branches of parallel matrix algebra can be executed. Here, the API allows to deviate from the SIMD programming model, introduced in Section 4.3.2, however SIMD is still enforced within a branch. Thus, the parallel scalability of a code can be improved, with the cost of a slightly more complicated program structure.

```
 1  program matrix_example1
 2    use matrix_parallel
 3    implicit none
 4    real(DP) :: A(n)
 5    real(DP) :: B(n, n), C(n, n)
 6    type(rdmatrix) :: pdA
 7    type(rmatrix) :: pB, pC
 8    type(pm_ctxt) :: ctxt
 9
10   ! distributed data object creation
11   pdA = matrix(A)
12   pB = matrix(B)
13
14   ! call a subroutine which does the matrix computation
15   call arithmetics1(pdA, pB, pC)
16
17   ! transition from a data object to an array
18   C = array(pC)
19  end program matrix_example1
20
21  contains
22   subroutine arithmetics1(A, B, C)
23     use matrix_parallel
24     implicit none
25     type(rdmatrix), indent(in) :: A, B
26     type(rmatrix), indent(in) :: B
27     type(rmatrix), indent(out) :: C
28
29     ! parallel matrix algebra
30     C = A * B * 2.0d0B
31   end subroutine arithmetics1
```

Figure 4.3.: A demonstration of the API usage. The example application is written in Fortran 90, and is split up into a `program` and a `subroutine`. Thus, a separation of concerns is complied with: the mathematical semantics are formulated in line 30 of the subroutine `arithmetics1` using a clear pseudo-mathematical syntax, whereas the management of the data objects is executed in the calling program. In this subroutine, parallelism can only be concluded from the invocation of the module `matrix_parallel` in line 23, and the data types of the operands (lines 25 to 27). The algebraic expression in line 30 is equal in both, the sequential Fortran 90 version, and in the parallel version using our API. Thus, a parallelization of this subroutine would only require an additional `use`-parameter, and different data types.

```
 1    program parallel_example
 2      use mpi
 3      use matrix_parallel
 4      implicit none
 5      real(DP) :: A, B
 6      integer :: comm1, comm2, myrank, ierror
 7      type(pm_ctxt) :: ctxt1, ctxt2
 8      type(rmatrix) :: pA, pB, pC
 9      type(rmatrix) :: pdE
10
11      ! fill arrays A, B with data
12      ...
13
14      ! create two independent MPI communicators
15      ! from MPI_COMM_WORLD and store them in ctxt1, ctxt2
16      ...
17
18      ! transition of the MPI contexts into API-specific contexts
19      ctxt1 = pm create ctxt(comm1)
20      ctxt2 = pm create ctxt(comm2)
21
22      call MPI_Comm_rank(comm1, myrank, ierror)
23      if(myrank != MPI_UNDEFINED) then
24        pA = matrix(A, ctxt1)
25        pB = matrix(B, ctxt1)
26        call arithmetics1(pA, pB, pC)
27      end if
28      call MPI_Comm_rank(comm2, myrank, ierror)
29      if(myrank != MPI_UNDEFINED) then
30        pA = matrix(A, ctxt2)
31        pB = matrix(B, ctxt2)
32        call arithmetics2(pA, pB, pC)
33      end if
34      ...
35    end program parallel_example
```

Figure 4.4.: An example application, written in Fortran 90, which demonstrates how the API can be used to execute two independent branches. The *MPI grouping facilities* [11] allow the organization of the available MPI processes into groups, represented by MPI communicators, and assign them to different tasks which can be executed in parallel. This MPI grouping must be done before the data object creation. In this example, the grouping is indicated by the comment in lines 14 and 15 (for the explicit syntax please refer to the MPI standard [11]), and the object creation is executed in lines 24, 25, 30 and 31. The two branches—in this example the bodies of the two `if`-statements in lines 23 and 29—call the two independent routines `arithmetics1` and `arithmetics2`, respectively, which contain algebraic expressions. The former routine links to the subroutine given in Figure 4.3, the latter links to another, independent, routine, which is not given here explicitly for brevity.)

4.4. Interface Implementation

The API, specified in Section 4.3.3, has been implemented as a library with Fortran 90 bindings. A complete description of the implementation would exceed the scope of this work, instead we present some technical details, important for the implementer of this interface.

The implementation language is Fortran 95. For the communication infrastructure, we use a standard MPI library. Additionally, we invoke performance optimized routines from the library PBLAS/ScaLAPACK [33] for the numerically expensive operations:

1. `PDGEMM` for the dense matrix multiplication;

2. `PDSYGVX` for the dense generalized eigenvalue problem;

3. `PDTRAN` for the matrix transpose.

There exist various parallel library implementations, which accomplish these tasks. See therefore the discussion in Section 5.1. Among the available libraries, we chose ScaLAPACK for practical reasons: it is a wide-spread standard, and as a consequence, it is part of many commonly used high-performance mathematical libraries, such as Intel MKL. Furthermore, it belongs to the standard software repository of many high-performance computing centers. This makes it easier to port the application to other machines, which might also be installed at other computing centers, and avoids technical difficulties due to library dependencies.

A technical constraint is that the employed compiler has to implement the Fortran 95 standard, together with the *enhanced data type facilities* (EDF), documented in the technical report TR15581 [62]. Features documented in this report became a part of the subsequent Fortran 2003/2008 standards and are implemented in most modern compilers, including GFortran and Intel Fortran Compiler.

Technically, the abstract data objects are represented by data types. The objects are opaque, enforced by the `private`-modifier, so they are only accessible by defined auxiliary routines. Internally, the data types contain an array which holds the actual data, and implementation-specific meta data, necessary for MPI communication, and PBLAS- and ScaLAPACK usage. See Figure 4.5 for the explicit implementation.

As the figure shows, the data type representing a dense matrix, `rmatrix`, contains the two-dimensional array `m(:,:)`, which holds the matrix elements. These data are distributed over the involved processes, defined by the MPI communicator stored in the field `mpi_comm`. Here, we decided for a *block-cyclic* distribution scheme, which is also the native distribution scheme of PBLAS and ScaLAPACK (see the ScaLAPACK user's guide [33]). This has the advantage that the distributed arrays can be handed over

```fortran
type, public :: rmatrix
  private
  integer :: mpi_comm = MPI_COMM_NULL
  integer :: desc(9) = -1
  real(DP), allocatable :: m(:, :)
end type rmatrix

type, public :: rdmatrix
  private
  integer :: mpi_comm = MPI_COMM_NULL
  integer :: blacs_ctxt = -1
  real(DP), allocatable :: d(:)
end type rdmatrix
```

Figure 4.5.: Declaration of the opaque `rmatrix` and `rdmatrix` data types, in Fortran notation. Integer array of length 9 is used by ScaLAPACK for a matrix descriptor and holds, among other data, matrix dimensions and a BLACS communicator.

directly to the routines `PDGEMM`, `PDSYGVX` and `PDTRAN`, without any additional communication. With this data distribution scheme, the array constructor function `array` is the only auxiliary routine that requires communication. The associated data collection algorithm has been implemented by hand, basically by using nested loops and the collective `MPI_Bcast` function. However, it is planned to replace this implementation by employing the *distributed array* data type, specified by the MPI 2.2 standard (see Section 2.5 of [11]).

The data type representing a diagonal matrix, `rdmatrix`, holds the one-dimensional array `d(:)`, which contains the diagonal matrix elements. We decided not to distribute this data, but to hold a complete copy of this array in each process. As the memory requirement for a diagonal matrix is far less demanding than that for a dense matrix, we believe that the potential memory savings in case of a distributed data structure would not outbalance the implied communication overhead and implementation effort.

For arithmetic operations between these data types, we overload the existing intrinsic operators with the `interface` construct. Figure 4.6 demonstrates this for the multiplication operator `*`. At compile time, the compiler does a static type check and resolves an operation to a specific function call, see Figure 4.7. Technically, each arithmetic operation creates a new data object as (intermediate) result. An example is the result of `mult_r_r(A, B)` in Figure 4.7. The allocation of this data object is not managed by the compiler automatically, hence, this has to be done explicitly in the function body of all operator routines. At the other hand, the deallocation of the intermediate objects can obviously not be done by the programmer. We therefore rely on a feature of the above mentioned EDF: `allocatable` components of a data type are automatically deallocated when its scope is lost, i.e. when the reference to the data type is overwritten by another data object or the declaring routine terminates [62]. This important feature facilitates arithmetic expressions between self-defined data types containing `allocatable` components, without introducing memory leaks.

```
interface operator(*)
  ! dense * dense:
  module procedure mult_r_r
  ! diagonal * diagonal:
  module procedure mult_d_d
  ! dense * diagonal:
  module procedure mult_r_d
  ! diagonal * dense:
  module procedure mult_d_r
  ! dense * scalar:
  module procedure mult_r_0
  ...
end interface operator(*)
```

Figure 4.6.: Multiplication Fortran operator interface.

```
type(rmatrix) :: A, B, D
type(rdmatrix) :: C

D = A * B * C
```

Figure 4.7.: Example code snippet. The arithmetic expression in line 4 translates into `D = mult_r_d(mult_r_r(A, B), C)`.

The distinction between physically distributed and non-distributed data objects requires also a distinction between local and global indices to ensure interoperability. The diagonal matrix is a data type, which is specific to our domain problem, thus usually not part of standard numerical libraries such as LAPACK. We therefore implemented the operations which involves diagonal matrices manually. Here, the most expensive operation is the product of a dense `rmatrix` and a diagonal `rdmatrix` as implemented in the routines `mult_r_d` and `mult_d_r` (see Figure 4.6). Formally, this multiplication is defined as follows:

$$\begin{aligned} C = A \cdot D \Leftrightarrow C_{ij} = A_{ij} \cdot d_j \,, \\ C = D \cdot A \Leftrightarrow C_{ij} = d_i \cdot A_{ij} \,, \end{aligned} \tag{4.11}$$

where $A, C \in \mathbb{R}^{m\times n}$, $D = \mathrm{diag}(d)$, $d \in \mathbb{R}^m$ or $d \in \mathbb{R}^n$, as appropriate. A favorable property of this operation is that the total $m \cdot n$ multiplications can be executed without communication. Still, one has to distinguish between the indices of a distributed and a non-distributed array. We therefore introduce the *local indices* $\{i_{\text{loc}}, j_{\text{loc}}\}$ for distributed arrays, and the *global indices* $\{i_{\text{glob}}, j_{\text{glob}}\}$ for non-distributed arrays. The multiplication can thus be defined as:

$$\begin{aligned} C = A \cdot D &\Leftrightarrow C_{i_{\text{loc}} j_{\text{loc}}} = A_{i_{\text{loc}} j_{\text{loc}}} \cdot d_{j_{\text{glob}}} \,, \\ C = D \cdot A &\Leftrightarrow C_{i_{\text{loc}} j_{\text{loc}}} = d_{i_{\text{glob}}} \cdot A_{i_{\text{loc}} j_{\text{loc}}} \,. \end{aligned} \tag{4.12}$$

For the implementation, a function is required which maps a local index to a global index. The ScaLAPACK library provides the function `INDXL2G`, which can be used to do this translation.

4.5. Evaluation

We evaluate the proposed library API and its implementation by means of two factors: the readability of the resulting code and its parallel performance. The former point has a special focus on existing Fortran 90 code which is parallelized, to accomplish a main objective of this API.

4.5.1. Code Evaluation

The code in Figure 4.2, which shows a sequential Fortran 90 implementation of a second order relativistic transformation, has been parallelized using our newly established library, introduced in Section 4.3. The explicit source code is listed in Figure 4.8. The listing shows that for the parallelization step, only the following three changes to the original code were necessary:

- inclusion of the statement `use matrix_parallel`;
- change of the dense matrix data types to `rmatrix`;
- the array allocation is not necessary anymore.

Note that all lines of code which contain mathematical semantics (lines 10 to 31) are completely unchanged. The sequential Fortran code is augmented by parallelism almost invisibly, and with very few changes to the original source code.

Figure 4.9 shows the Fortran 90 listing of the alternative iterative procedure for relativistic transformations, briefly introduced in Equation (4.10). This implementation is parallelized using the distributed matrix library, and part of ParaGauss. Together with the listing for the second order transformation in Figure 4.8, already discussed in the previous paragraph, it shows how our API facilitates the expression of mathematical semantics in a pseudo-mathematical notation, directly in a parallel high-performance Fortran 90 code. By our subjective metric, the mathematical statements are as close to the abstract mathematical expressions, introduced in Section 4.2, as it could get with

```
subroutine reltrans_parallel(S, T, V, O, T_rel, V_rel)
 use matrix_parallel
 implicit none
 type(rmatrix), intent(in)  :: S, T, V, O
 type(rmatrix), intent(out) :: T_rel, V_rel
 type(rmatrix) :: U, U_inv, R, V_mom, O_mom, aVa, aOa
 type(rdmatrix) :: td, td_rel, e, a, b, r2

 ! call the generalized eigenvalue solver
 call geigs(T, S, td, U)

 ! call the internal routine to compute the relativistic factors
 call factors(2.0d0 * td, td_rel, e, a, b, r2)

 ! transformation into momentum space
 V_mom = tr(U) * V * U
 O_mom = tr(U) * O * U

 ! intermediate results
 aVa = a * V * a
 bOb = b * O * b
 R = rpt(e, r2 * aVa - bOb)
 W22 = tr(RW) * (0.5d0 * r2**(-1)) * RW
 U_inv = tr(U) * S

 ! transformation of V and O into V_rel
 V_rel = aVa + bOb + tr(RW) * (e * r2**(-1)) * RW + e * W22 + W22 * e

 !back-transformation into real space
 T_rel = tr(U) * t_rel * U_inv
 V_rel = tr(U) * V_rel * U_inv
end subroutine reltrans_parallel
```

Figure 4.8.: The parallel version of the sequential routine `reltrans`, listed in Figure 4.2. Here, the data object creation of the intermediate results (`U`, `td`, . . .) has been moved to the respective routines, e.g. `factors`. Note that the input- and output data objects have to be created explicitly by the calling routine. Thus, the only differences to the sequential version in Figure 4.2 are the data type of the dense matrices (`rmatrix`), the `use matrix_parallel`-statement, and the missing `allocate`-statements. All expressions containing algebraic expressions are unchanged, and are displayed in a comprehensible pseudo-mathematical notation.

```fortran
function rico_parallel(e, O, E1, E2) result(X)
use matrix_parallel
implicit none
type(rdmatrix), intent(in) :: e
type(rmatrix), intent(in) :: O, E1, E2
type(rmatrix) :: X
! *** end of interface ***

integer :: iter
type(rmatrix) :: O1, X1
real(DP) :: cond, tol

X = rpt(e, O)

cond = maxabs(X)

if (cond == 0) return

tol = huge(tol)
iter = 0
do while (tol / cond > 10 * epsilon(tol))
  iter = iter + 1

  O1 = O - X * E1 + E2 * X

  if (iter > 1) then
    O1 = O1 - X * tr(O) * X
  end if

  X1 = X
  X = rpt(e, O1)
  tol = maxabs(X - X1)
end do
end function rico_parallel
```

Figure 4.9.: Fortran code for the iterative procedure along the lines of Eq. (4.10) illustrating the use of the matrix algebra library in an imperative code with multiple assignments to the same structure. The domain specific function `maxabs(X)` returns a $\max |X_{mn}|$ for a matrix X.

the Fortran syntax. There is obviously no "abstraction leaking": none of the implementation details of the distributed data objects and corresponding operations is exposed. At the other hand, all computationally expensive operations are executed by highly optimized parallel routines, invisibly in the background. Thus, our library facilitates clear, concise, and comprehensible code at a very high level of mathematical abstraction, which is executed fast and efficiently, as the performance evaluation in Section 4.5.2 shows. This generally contributes to the implementation efficiency and the code quality in high-performance scientific Fortran codes.

Note that declarations of distributed matrices do not reserve space for actual data. It is the responsibility of the primitive operations, introduced in Section 4.3.3, to reserve space, initialize and finally fill the structure with data. Upon return from the subroutine, all intermediate structures declared in the scope of the subroutine, will be automatically freed by a standard conforming compiler. This makes explicit deallocation dispensable and simplifies, thus, the introduced programming model further.

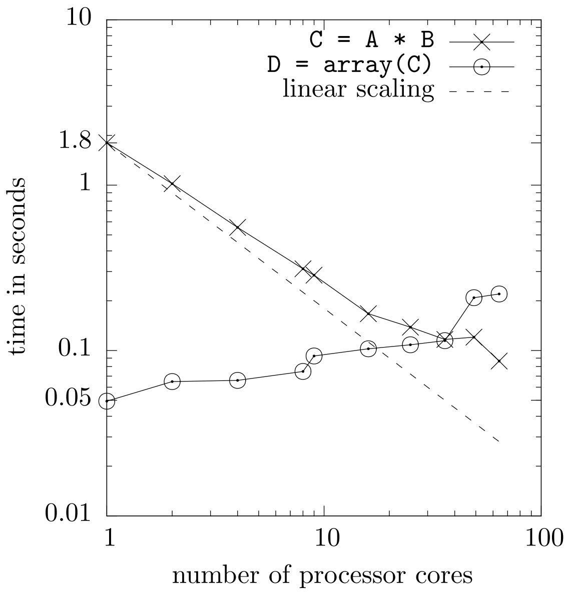
4.5.2. Performance Results

The run time performance has been evaluated on the supercomputer system SuperMIG, built by IBM, and installed at the Leibniz Rechenzentrum (LRZ)[2]. The machine contains 205 nodes, each hosting four 10-core Intel Xeon Westmere-EX processors. The nodes are equipped with 256 GB of memory and interconnected by an Infiniband QDR network. With this topology calculations involving, for example, 36, 64 and 81 processor cores were scheduled on one, two, or three nodes, respectively. For the benchmarks we employed the default vendor supplied MPI library implementation, IBM MPI v5.2, and BLACS/ScaLAPACK v1.8, linking to the highly optimized BLAS library distributed with Intel's MKL v10.3.

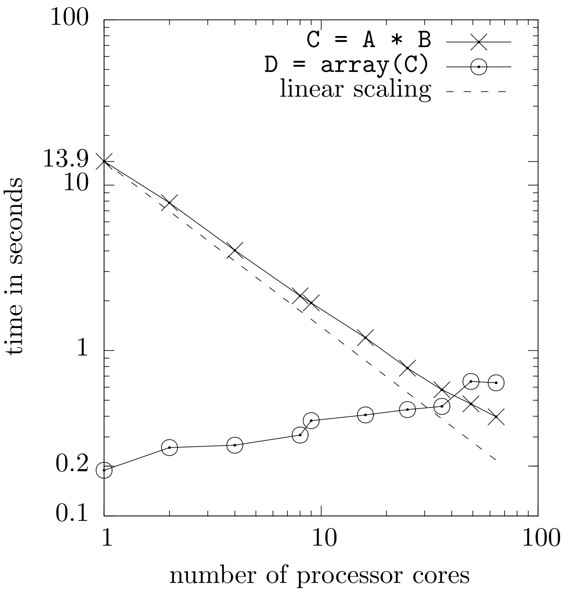
The two graphs in Figure 4.10 demonstrate the scaling behavior of a single dense matrix product as one of the primitive linear algebra operations. The parallel efficiency is limited by the underlying PBLAS implementation [33], provided by the LRZ. For a typical matrix dimension of 2000 to 4000, the available implementation shows an efficiency above 50% for a processor core number below 100. For higher matrix dimensions one may expect the parallel efficiency to degrade slower with the number of processors. A noteworthy observation in both graphs is the "jump" in the graph related to the `array`-curve. This jump is accompanied with a slight "bend" in the scalability curve related to the parallel matrix multiplication. This jump happens when the total processor core count crosses the number of cores comprised in a shared memory node—in this case 40—and communication over the physical network is starting to be involved.

For the matrix dimensions used in Figure 4.10, the overhead implied by the use of distributed data objects—in this example case the execution of the auxiliary routine

[2]www.lrz.de



(a) $n = 2000$



(b) $n = 4000$

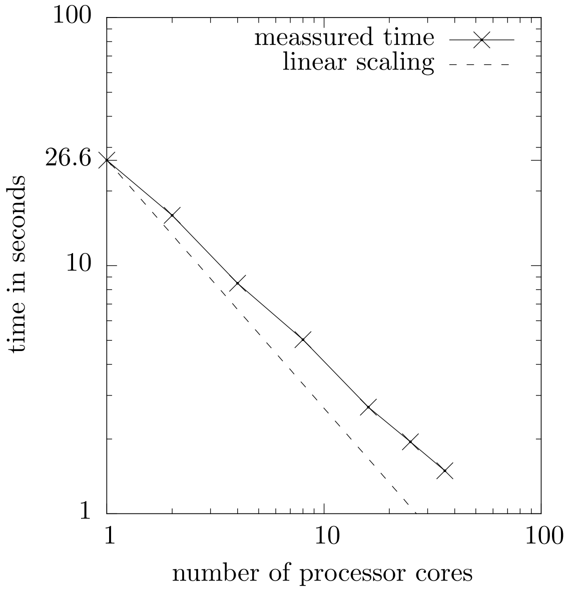
Figure 4.10.: Log-log parallel time diagrams of a *single* dense matrix product (labeled "`C = A * B`"), opposed to the overhead implied by the creation of a non-distributed array from a distributed data object (labeled "`D = array(C)`"). We considered two cases, involving test matrices of dimensions 2000 and 4000, respectively. The graphs demonstrate, how much distribution overhead is produced by the auxiliary routine `array`, and how this compares to the desired parallel matrix multiplication. With growing processor core numbers, we can see a clear decrease of time for the matrix multiplication, which is the desired performance improvement, and a slight increase of the distribution overhead.

`array`—becomes comparable with the costs of a *single* matrix multiplication at about 40 involved cores. Of course, this number depends heavily on the underlying network infrastructure. A real-world application typically involves considerably more arithmetic operations, which amortizes this overhead quicker and allows, thus, the use of higher processor numbers in an efficient way. However, this demonstration should also be seen as a hint to the programmer, to apply transitions from and to native arrays wisely and, if possible, sparsely.

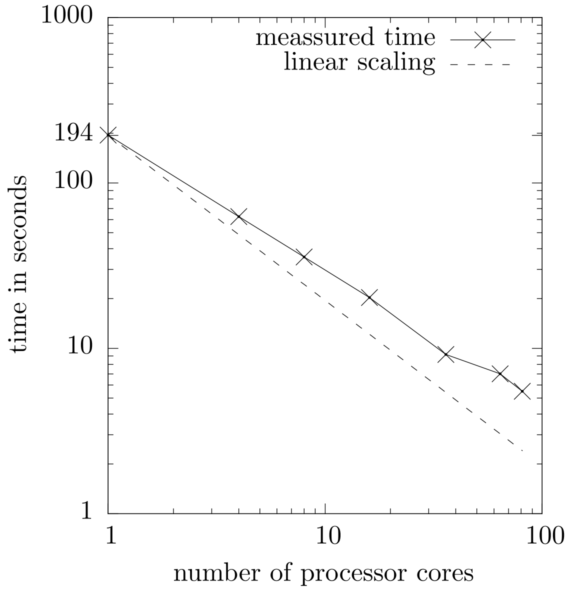
The graphs in Figure 4.12 show the scalability of the two parallelized routines for relativistic transformations, `reltrans_parallel` and `rico_parallel`, see listings 4.8 and 4.9, respectively. The transition costs from- and especially to native arrays are not considered in these values. We evaluated two different problem sizes: the involved matrix dimensions $n = 2000$ and $n = 4000$. The scalability of the transformation routines is determined by the underlying parallel numerical routines, mainly the PBLAS implementation, as well as the eigensolver from ScaLAPACK in the case of the routine reltrans_parallel. Their execution involves $\mathcal{O}(n^3)$ FLOPs, and intensive communication. The impact of the self-implemented routine, responsible for the operation "diagonal matrix * dense matrix", should be minor: its execution involves $\mathcal{O}(n^2)$ FLOPs, and no communication, see Section 4.4. The graphs show that the execution time of both routines are reduced significantly, by employing our distributed matrix library at high processor numbers. For example, the execution time of the routine rico_parallel with input matrices of dimensions $n = 4000$ could be reduced from about 194 seconds down to 5.5 seconds, which is a speedup factor of roughly 35. Note also the characteristic "bend" in Graphs 4.11b and 4.11d, when the processor core count crosses the "40"-barrier, as already discussed in earlier in this Section.

The parallel efficiency of the performance runs is given by the graphs in Figure 4.12. One can observe that the routine rico_parallel has generally a better scalability behavior than the routine reltrans_parallel: for $n = 4000$, rico_parallel still has an efficiency of roughly 0.6 when employing 81 processor cores, whereas the efficiency of reltrans_parallel drops down to 0.45. This effect is mostly imposed by the parallel eigensolver, which scales typically worse than a parallel matrix multiplication.

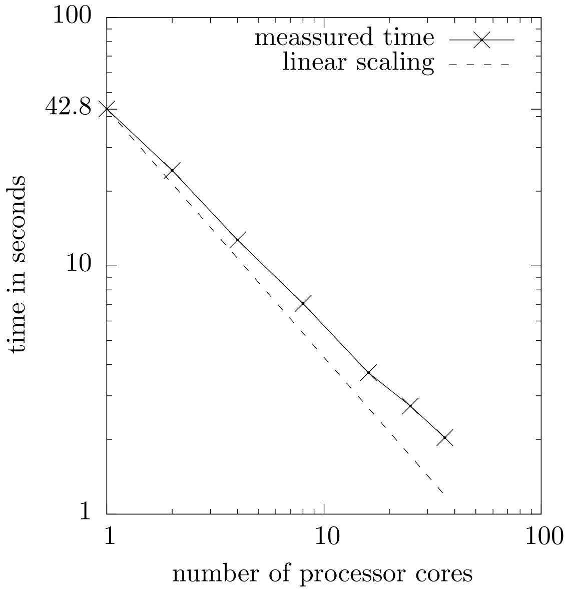
To briefly summarize what the results from this Evaluation section show: the proposed library specification for distributed matrices, which states the central point of the work presented in this chapter, facilitates an easy parallelization of the existing routines responsible for the relativistic transformations in ParaGauss. The entailed API retains the intrinsic Fortran 90 syntax for linear algebra operations, which allows for a clear, concise, and comprehensible expression of mathematical semantics. Thus, with very few changes to the original code, our test runs could be accelerated up to a factor of 35. Furthermore, memory-intensive data objects can now be distributed over the global memory of potentially all involved processor nodes. Apart from this specific application from quantum chemistry, we would like to emphasize that the use of the this library is also suitable for new developments of parallel codes: its pseudo-mathematical syntax allows for a quick implementation, while producing high-quality source code. Thus, it contributes to the programming productivity in scientific high-performance applications.



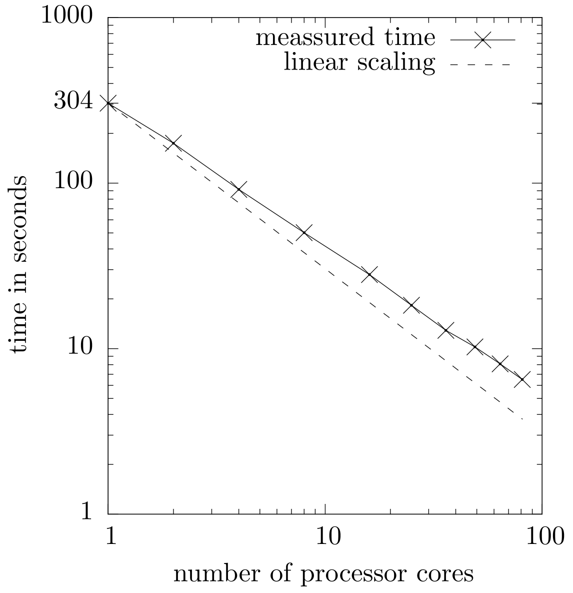
(a) `reltrans_parallel`, $n = 2000$



(b) `reltrans_parallel`, $n = 4000$

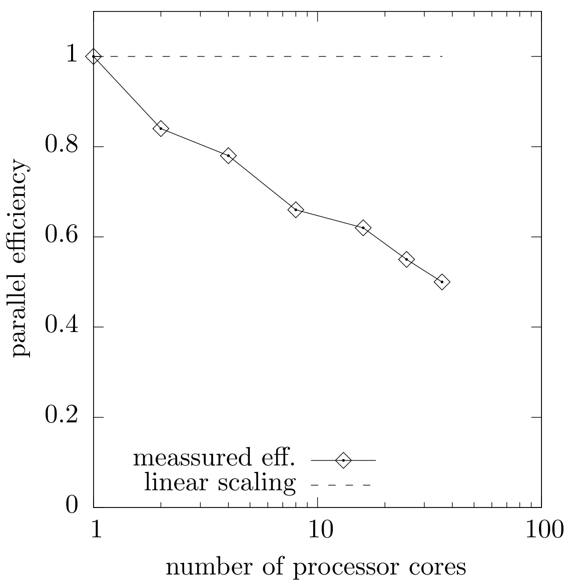


(c) `rico_parallel`, $n = 2000$

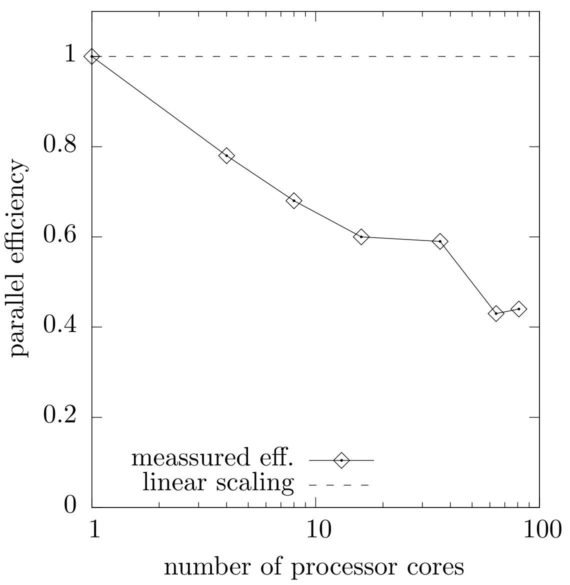


(d) `rico_parallel`, $n = 4000$

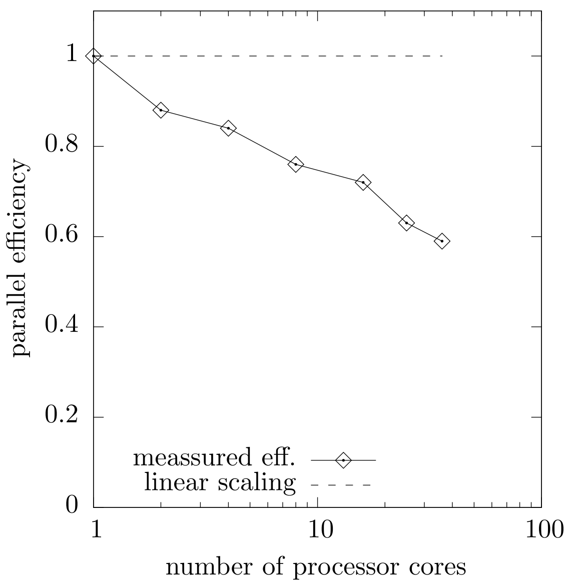
Figure 4.11.: Log-log time diagrams of the routines `reltrans_parallel` ((a) and (b)) and `rico_parallel` ((c) and (d)). The source codes of the routines are listed in Figures 4.8 and 4.9, respectively. Considered is the wall-clock time of one call to a routine, labeled “meassured time”,using test matrices of dimensions 2000 and 4000.



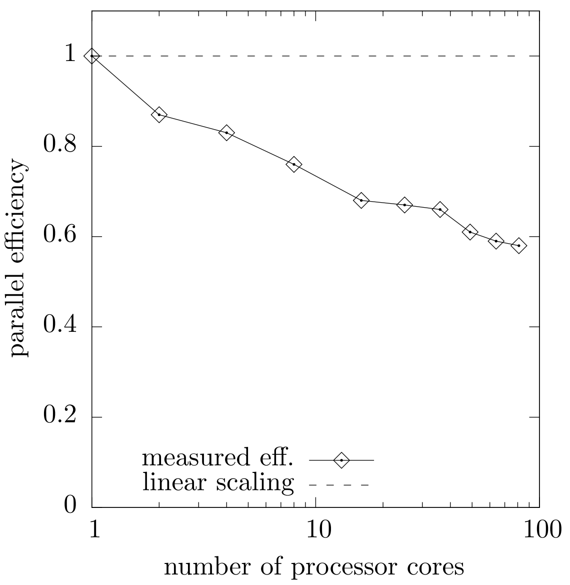
(a) `reltrans_parallel`, $n = 2000$



(b) `reltrans_parallel`, $n = 4000$



(c) `rico_parallel`, $n = 2000$



(d) `rico_parallel`, $n = 4000$

Figure 4.12.: Semi-log efficiency diagrams of the routines `reltrans_parallel` ((a) and (b)) and `rico_parallel` ((c) and (d)). The curves labeled "measured eff." show the parallel efficiency of the executed routine, and correspond to the times exhibited in Figure 4.12.

5. Scheduling Parallel Eigenvalue Computations in SCF

The solution of the generalized matrix eigenvalue problem,

$$HC = SCE\,, \tag{5.1}$$

is a central step of the LCGTO-formalism, presented in Chapter 3. It gives an approximate solution of the Schrödinger equation in each iteration of the SCF algorithm. Next to the computation of the Coulomb- and exchange-correlation contributions, this linear algebra problem states a major computational step of existing density functional codes. Within the LCGTO-formalism, the Hamilton- and overlap-matrix, H and S, respectively, are symmetric real and can be assumed to be dense in most applications. Their dimensions correspond to the number of employed basis functions, L, and add up to ten thousand in large applications. The problem in Equation (5.1) is well studied, see [63, 64] for a collection of standard numerical algorithms. However, the special block-symmetric matrix structure, implied by the basis transformations introduced in Section 3.2, make the efficient parallel computation difficult. In this chapter we present a novel technique and its implementation, which employs a *malleable parallel task scheduling (MPTS)* algorithm to make most efficient use of existing parallel eigenvalue solvers in this special matrix problem. Thus, we provide a scalable solution, which reduces the time spent in this important step significantly, especially in large chemical applications. Therefore, we first introduce the standard approaches to solve dense matrix eigenvalue problems, and give common parallel implementations in Section 5.1. Then, Section 5.2 presents the previous parallelization strategy, which exhibits very limited parallel scalability. Thereafter, in Sections 5.3 and 5.4, we establish a scheduling algorithm, which is the core component of our novel approach. Furthermore, in Section 5.5 we give a technique for the practical implementation of a cost function, required by the algorithm. Section 5.6 shows some details of the implemented ParaGauss module and its software structure. Finally, in Section 5.7 we evaluate the parallel performance of our implementation in real chemical applications.

5.1. The Generalized Matrix Eigenvalue Problem

In common approaches, the first step to solve Equation (5.1) is to compute the Cholesky factorization $S = GG^T$. By applying $A = G^{-1}HG^T$, the generalized symmetric eigenvalue problem is transformed into a standard symmetric eigenvalue problem of the form

$$AC = CE\,, \tag{5.2}$$

where $C^T A\, C = E = \mathrm{diag}(\lambda_k)$, $k = 1 \dots L$. Here, L is the matrix dimension, λ_k are the eigenvalues of the matrix A, and the columns of C constitute the corresponding eigenvectors: $AC[:,k] = \lambda_k C[:,k]$. To tackle the problem in Equation (5.2) efficiently, most approaches compute a matrix Q such that

$$Q^T A\, Q = T\,, \tag{5.3}$$

where T has a tridiagonal band form. This *tridiagonalization* step is usually done by applying subsequent Householder transformations to A and has an overall runtime of $\mathcal{O}(N^3)$ (recall that $L \propto N$). A common method to compute the eigenvalues of T, which equal the eigenvalues of A, is the iterative *QR algorithm* (see Section 8.3 of [63]), which requires $\mathcal{O}(N^2)$ operations for this task. If additionally all eigenvectors, here stored in the columns of the matrix V_T, are required, the costs expand to $\mathcal{O}(N^3)$ steps. Recently, the *Multiple Relatively Robust Representations (MRRR)* algorithm [65] has gained popularity, which computes all eigenvectors from T in $\mathcal{O}(N^2)$ steps at the cost of lower accuracy for some matrix structures.

Finally, the eigenvectors of the original matrix H, stored in the columns of C, can be achieved by the back-transformation

$$C = Q^{-1} V_T\,. \tag{5.4}$$

The overall process requires $\mathcal{O}(N^3)$ operations, and represents, as already indicated, a major computational step of existing density functional codes. Thus, an efficient and scalable parallelization strategy is of crucial importance to avoid computational bottlenecks in large-scale applications. This performance issue becomes even more important if one considers that the generalized eigenvalue problem is part of the SCF, therefore the eigenvalue solution is computed up to a hundred times in a single electronic structure calculation. In case the molecule geometry is optimized, the electronic structure has to be determined up to several hundred times, which adds up to $10^3 - 10^5$ eigenvalue computations in a typical quantum chemical application.

Note that different discretization approaches than LCGTO, such as plane waves, usually cope with sparse matrices, so corresponding codes resort to iterative methods, such as the Davidson algorithm [66]. These iterative algorithms are very different from the ones introduced in the first part of this section. For more information, please refer to Chapter 5 of [23].

Linear algebra problems, such as the dense eigenvalue problem, appear in a large variety of applications from scientific computing. Hence, much effort has been invested in the design of parallel libraries and their efficient implementation. The two most common libraries for distributed memory architectures are *ScaLAPACK* [33] (a parallel version of the popular *LAPACK* library [32]), and *PLAPACK* [35]. Both were developed in the 1990s when supercomputers started to comprise more and more massive parallelism. Their use is widely spread, consequently, library installations belong to the standard software repository of most supercomputing centers. More recent developments are the *ELPA library* [6], which has a special focus on the symmetric eigenvalue problem. Furthermore, the *Elemental Framework* [67] aims at hybrid supercomputer architectures (clusters of multicore CPUs). Different from that, the *FLAME*- [68] and the *PLASMA project* [69] both develop scalable linear algebra routines for single-chip environments, especially for many-core CPUs, by employing dependency-aware dynamic schedulers. However, their library implementations are still actively developed and do not yet provide full functionality – especially the latter one does not yet compute eigenvectors. Another interesting project is the *MAGMA project* [69], closely related to PLASMA, which develops high-performance routines for emerging massively parallel SIMD hardware accelerators, such as GPUs.

However, the direct application of existing parallel eigenvalue solvers (abbreviated: eigensolvers) to H and S is only possible to a very limited extent when the spatial symmetry of a molecule is exploited as described in Section 3.2. The transformation of the basis functions into a symmetry adapted form gives rise to a transformation of the dense structure of the two matrices into a special block-diagonal form,

$$\begin{aligned} H &= H^{\Gamma_1} \oplus H^{\Gamma_2} \oplus \cdots \oplus H^{\Gamma_n} \,, \\ S &= S^{\Gamma_1} \oplus S^{\Gamma_2} \oplus \cdots \oplus S^{\Gamma_n} \,. \end{aligned} \tag{5.5}$$

The blocks on the diagonal correspond to the irreducible representations (IRREPs) Γ of the molecule, and add up to 4 and 10 in typical applications. They are real, dense, and symmetric, as the original matrices, and can vary in size up to one order of magnitude. The great advantage of this method is that the space spanned by the L basis functions is split into several smaller sub-spaces, which reduces the problem size significantly. Thus, electronic structure calculations can be sped up by several orders of magnitude.

5.2. Previous Parallelization Strategy

This section describes the previous approach, implemented in ParaGauss, to parallelize the blocked generalized eigenvalue problem.

The sub-spaces, in their discrete form represented by the sub-matrices H^{Γ} and S^{Γ}, can be treated independently. This important property offers a first straightforward parallelization strategy, in which the single matrix blocks are distributed over the available processing units and diagonalized independently by a sequential eigensolver, e.g. the routine `DSYEV` from *LAPACK*. Here it is important to choose an appropriate scheduling scheme, which distributes the matrix blocks in a way that provides good load-balancing and minimizes, thus, the *makespan* (the overall time to process all blocks). A common scheme is given by the *LPT algorithm* [70] as a special form of *list scheduling* [71]. Practically, the algorithm can be implemented by using the *master-worker model*: a dedicated process—the master—holds a list of *tasks*. In our case, a task represents the diagonalization of a sub-matrix. The tasks are sorted by the size of the sub-matrices in descending order, with a pointer to the first (largest) task. Next to the master, there is a set of worker processes, responsible for the execution of the tasks. In the initial status of the master-worker-model, the master is waiting for queries from the workers. Each time the master receives a request, he responds with the task the pointer points to, and sets the pointer to the next-largest task. These steps are repeated until the pointer reaches the end of the list: all tasks are sent to the workers for their execution. All subsequent queries are answered with a termination signal, until the signal has been sent to all workers. The workers start by querying the master for a task. Once they receive a task, they execute it (diagonalize the sub-matrix), and when finished, send a query to the master again. This is repeated until they receive a termination signal.

However, this approach shows very limited scalability: the sequential execution time for the largest matrix is a lower bound on the overall execution time, and the number of matrices is an upper bound on the number of processors which can be used. Furthermore, due to the variation of block-sizes, the workload is often poorly balanced and the available computing resources are used inefficiently. The example in Figure 5.1 demonstrates this behavior. Although exploiting the symmetry reduces the total problem size significantly, the dimension of the sub-matrices still grows proportionally to the number of involved electrons, $L^\Gamma \propto N$. Accordingly, the blocked eigenvalue problem still is a $\mathcal{O}(N^3)$-problem, which requires more elaborate parallel treatment to avoid bottlenecks in large applications.

5.3. Malleable Parallel Task Scheduling

5.3.1. Abstract Formulation and Notation

Here we present a technique, which improves the previous LPT-approach by employing existing parallel eigensolvers for the diagonalization of the sub-matrices H^Γ. This technique has been reported in [72]. Therefore, we define a set of *tasks* $\mathcal{T} = \{T_1, \dots, T_n\}$, where a task represents the—possibly parallel—execution of computing all generalized eigenvalues and -vectors of a sub-matrix H^Γ with corresponding overlap sub-matrix S^Γ. (For a better readability we switch to the number-based indices $i \in \mathbb{N}$ instead of the group-specific identifier Γ, with the relation $\Gamma \mapsto i$.) For the computation, there is a set $\mathcal{P} = \{P_1, \dots, P_m\}$ of m identical processors P_j available. The tasks are independent and *nonpreemptable* (i.e. not interruptible). Furthermore they are *malleable*: a task may be executed by a number of processors $p \leq m$, resulting in different execution times.

We define the *cost function* as

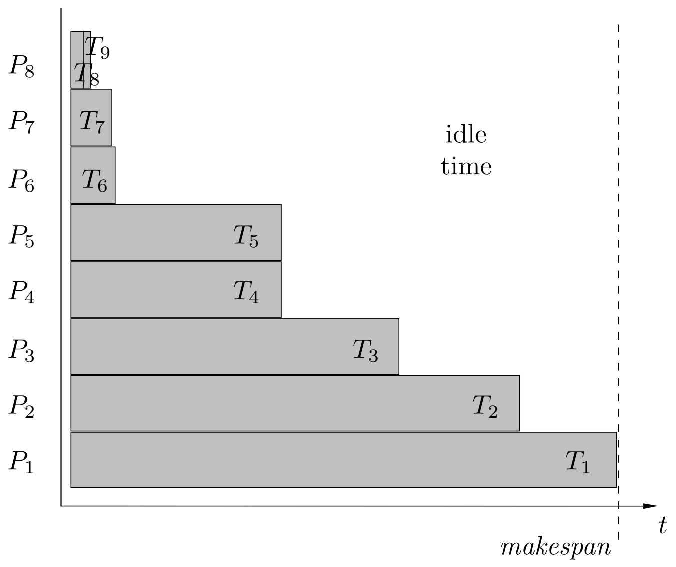


Figure 5.1.: Example of an LPT scheduling of the tasks $T_1, \dots, T_9$ on 8 processors. Each gray bar represents a task, with the width being its execution time. One can easily see that the *makespan* is determined by the largest task, here scheduled on P_1.

$$t : (T_i, p) \mapsto t_{i,p} \, . \tag{5.6}$$

It predicts the execution time of task T_i, based on its size and the number of processors p, used to execute the task. The total time required to process all tasks is denoted as the *makespan*. The goal of the malleable parallel task scheduling (MPTS) technique is to minimize the makespan, by parallelizing single tasks and scheduling the workload over the available processor resources. This guarantees:

- a most efficient use of the available computing resources,
- overall scalability of the blocked eigenvalue step, and
- minimal time spent in this significant computational step.

This minimization problem is well studied, see the discussion in the next section. Figure 5.2 shows an illustration of an MPTS example.

5.3.2. Related Work

MPTS is a common scheduling problem and has been subject of frequent discussion over the last decades [73, 74, 75]. It is a generalization of a sequential scheduling problem, which has been proven to be NP-complete in the strong sense [76]. Therefore, a variety of approximation algorithms exist. Approximation algorithms are used to give

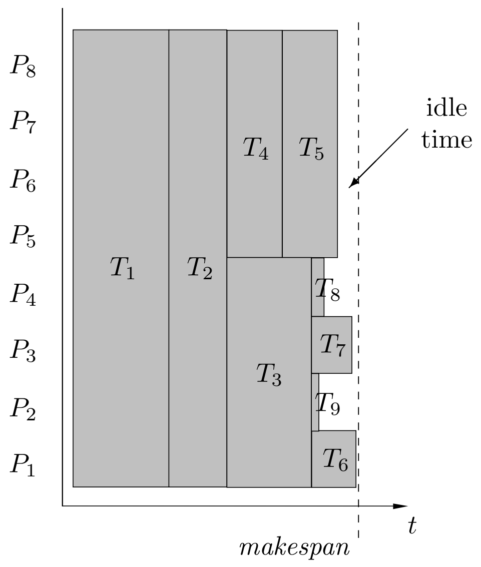


Figure 5.2.: Example of an MPTS scheduling of the tasks $T_1, \ldots, T_9$ on 8 processors. The tasks are equivalent to those from the LPT-example in Figure 5.1. In contrast to the LPT-scheduler, here the tasks are considered "malleable", i.e. can be executed by more than one processor at the same time.

near-optimal solutions to NP-problems in polynomial time. Their quality is mainly determined by the *approximation factor*, which is an upper bound on how much worse the approximate solution can be, compared to the *optimal solution*. Another criterion is the (asymptotic) time required by the algorithm to generate the approximate solution.

A common approach to this solution is based on a two-phase strategy, first introduced by Turek et al. [77]. The idea is to find a processor allotment for each task in the first step and to solve the resulting *nonmalleable parallel task scheduling* (NPTS) problem in the second step. Ludwig and Tiwari [78] suggested such an algorithm with an approximation factor of 2. This algorithm is also the basis of the strategy proposed here. Mounié et al. [79] followed a different approach by formulating a Knapsack problem as the core component. They provide the currently best practical algorithm with an approximation factor of $\frac{3}{2} + \epsilon$ for any fixed $\epsilon > 0$. When the tasks have to be executed on processors with successive indices, Steinberg [80] proposed an adapted strip-packing algorithm with an approximation factor of 2. Furthermore, Jansen [81] gave an approximation scheme with makespan at most $1 + \epsilon$ for any fixed $\epsilon > 0$. There exist other algorithms for special cases of the MPTS with approximation factors close to one (e.g. [82] for identical malleable tasks), but those do not apply for our case.

5.4. The Employed MPTS Algorithm

The problem stated in Section 5.3.1 is a standard MPTS problem, so the algorithms mentioned above can in principle be applied. However, as will be shown, the limited number of tasks allows us to modify the algorithm by Ludwig and Tiwari [78] in order to get an improved solution.

The algorithm proceeds in a two-phase approach. The first phase generates a *processor allotment*: it assigns each task a fixed number of processors. Thus, the *malleable* scheduling problem is transformed into a *nonmalleable* one (MPTS $\Rightarrow$ NPTS). In the second phase, we construct an *optimal* solution for the NPTS problem by using a combinatorial approach. At this point, our approach differs from the originally proposed algorithm in [78], which constructs an approximate solution.

5.4.1. Phase 1: Processor Allotment

Here we present the basic statements of the first phase of the algorithm presented in [78], necessary for practical implementations. For details and proofs, we refer to [78].

Recall that the predicted execution time of a task is denoted by $t_{i,p}$, i being the task index and p the number of processors assigned to that task, see Equation (5.6). The algorithm assumes t to be *monotonic*: $t_{i,p} \geq t_{i,p'}$ for $p < p'$. This property is not generally given – practically, more processors can unintentionally result in a higher execution time. Thus, the monotonic assumption needs to be considered in the implementation of the cost function (see Section 5.5). Furthermore, we introduce the set $\mathcal{D}$, which contains all processor assignments to a task. The set may generally contain the integers, $\mathcal{D} = \{1, 2, \ldots, m\}$, however, technical constraints might allow only a certain subset. For example, one such constraint can be implied by the network topology (e.g. a hypercube allows only $\mathcal{D} = \{2^0, 2^1, 2^2, \ldots, 2^r\}$), or other constraints or simplifications, as will be introduced later.

Let $\bar{p} = (p_1, p_2, \ldots, p_n)$ be an *allotment*, where $p_i \in \mathcal{D}$ denotes the chosen processor assignment to task T_i. The longest execution time of all tasks in an allotment is defined by $h(\bar{p}) = \max_i\{t_{i,p_i}\}$. Furthermore, define

$$\omega(\bar{p}) = \max\{\frac{1}{m}\sum_{i=1}^{n} p_i t_{i,p_i}, h(\bar{p})\} \tag{5.7}$$

as the lower bound on the makespan of the allotment $\bar{p}$. Hence, the minimization problem

$$\omega = \min_{\bar{p} \in \mathcal{D}^n} \omega(\bar{p}) \tag{5.8}$$

yields a lower bound on the makespan of the MPTS (there is possibly more than one ω). The goal of this phase of the algorithm is to find a feasible ω with corresponding allotment. This accomplishes the transformation from an MPTS problem to an NPTS problem.

Therefore, define

$$v[i,1] < v[i,2] < \cdots < v[i,z[i]] \tag{5.9}$$

as all possible number of processors that can be assigned to task T_i. A number $v[i,j]$ can be assigned if the resulting execution time $t[i,j] = t_{i,v[i,j]}$ meets the monotonic assumption stated above. Thus,

$$t[i,j] \geq t[i,j+1] \quad \forall \;\; j \in \{1, \ldots, z[i]-1\}\,. \tag{5.10}$$

Equation (5.7) can also be written as

$$\omega = \min_{\tau} \max\{\tau, \frac{1}{m} \min_{\bar{p} \in \mathcal{D}^n} \{\sum_{i=1}^{n} p_i t_{i,p_i}\}\}\,, \tag{5.11}$$

where for now, let $\tau \in \mathbb{R}^+$ be an arbitrary time replacing $h(\bar{p})$. Define

$$j_i(\tau) = \min\{j : t[i,j] \leq \tau\}\,, \tag{5.12}$$

which yields the smallest number of processors required to achieve an execution time smaller or equal to τ for task T_i. Furthermore,

$$\min_{\bar{p} \in \mathcal{D}^n} \{\sum_{i=1}^{n} p_i t_{i,p_i}\} = \sum_{i=1}^{n} v[i, j_i(\tau)] \cdot t[i, j_i(\tau)]\,. \tag{5.13}$$

The proof of Equation (5.13) is given in [78]. With

$$W(\tau) = \frac{1}{m} \sum_{i=1}^{n} v[i, j_i(\tau)] \cdot t[i, j_i(\tau)] \tag{5.14}$$

Equation (5.11) now has the form of

$$\omega = \min_{\tau} \max\{\tau, W(\tau)\}\,. \tag{5.15}$$

Note that the original minimization problem from Equation (5.7), which searches for an allotment $\bar{p}$ in the n-dimensional space $\mathcal{D}^n$, has been transformed into a minimization problem which searches a single value τ in the one-dimensional space $\mathbb{R}^+$. Furthermore, with Equation (5.11), we can restrict τ to the values which $h(\bar{p})$ can possibly have, comprised by the set $X = \{t[i,j]\}$.

This suggests a viable strategy to tackle the central minimization problem. The arrays $t[i,*]$ are already sorted by definition (see Equation (5.9) and the monotonic assumption). Hence, they can be merged into a single list of size $\mathcal{O}(|\mathcal{D}|n)$ in time $\mathcal{O}(|\mathcal{D}|n \log n)$. The time τ, which yields the minimum bound ω, can then be determined by binary search, which requires $\mathcal{O}(\log(|\mathcal{D}|n))$ probes. Each probe requires $\mathcal{O}(n \log |\mathcal{D}|)$ operations (see Equations. (5.12) and (5.14)). However, the total runtime is dominated by the sorting algorithm, which requires $\mathcal{O}(|\mathcal{D}| n \log n)$ operations. In case of successive processor numbers ($\mathcal{D} = \{1, \ldots, m\}$), the computational costs can also be denoted as $\mathcal{O}(mn \log n)$. The paper [78] also suggests an alternative algorithm, which improves this runtime to $\mathcal{O}(mn)$. However, for our practical solution, the presented algorithm shall be sufficient.

5.4.2. Phase 2: The NPTS Problem

A central statement of [78] is that any solution to the NPTS problem can be used to solve the MPTS problem with the *same approximation factor*, using the above presented transformation. See therefore Theorem 3.1 and the following proof in [78]. In our specific case, the number of tasks, given by the number of IRREPs of the applied point group, is not more than 10 in the great majority of applications. This includes the important point groups O_h and I_h and their subgroups as well as in all point groups with up to fivefold rotational axes as symmetry elements [83]. Based on this assumption of the maximum problem size, we show how to compute an *optimal* solution by using a combinatorial approach in relatively few computing time. While the running time of this approach is higher than computing an approximate solution, we show that this invested time is amortized by a faster computation of the actual eigenvalue problem in Section 5.7.

In our approach, a feasible scheduling is represented ambiguously by a particular *sequence* of tasks. In turn, this task sequence is represented by a permutation σ of the sequence of integer numbers $(1, 2, \ldots, n)$:

$$\sigma = \begin{pmatrix} 1 & 2 & \ldots & n \\ \sigma(1) & \sigma(2) & \ldots & \sigma(n) \end{pmatrix} , \tag{5.16}$$

where $\sigma(i) \in \{1, \ldots, n\}$. To map a number to a distinct task, we define the function

$$T : i \mapsto T_i \, , \tag{5.17}$$

such that $T(\sigma(i))$ is the i-th task in the permutation sequence σ.

In Phase 1 of the algorithm, each task T_i is already assigned a number of processors p_i. Now, we give an algorithm, which converts the permutation σ into a feasible scheduling, stored in a data structure for its execution in a parallel software code. Therefore, we expand the already introduced task- and processor objects, T_i and P_j, respectively, by some additional fields. To address a field of an object, we use the practical dot-notation: *object.field* refers to a property or a nested data structure (i.e. a set or list) comprised by a specific object. For examples see the following paragraph.

Each processor $P_j \in \mathcal{P}$ is assigned a list $T[]$, that holds those tasks the processor has to process in the right order: $P_j.T[i] \in \mathcal{T}$, $i \in \{1 \dots n_j\}$ refers to the i-th task to be processed by processor P_j, with n_j being the total number of assigned tasks. We will refer to this list as the *processor stack*. These stacks can be easily implemented with standard (dynamic) arrays, comprised in a data structure representing a task. When the tasks are finally executed, each process simply has to execute the tasks in his stack in the right order. Furthermore, the attribute $P_j.t$ stores the (predicted) *stack availability time*, which is the time when processor P_j has finished the execution of all tasks in $P_j.T[]$, and is available for the processing of new tasks. To finally execute the tasks, the processor also has to know, which other processors are possibly involved. This information is stored in the task-private set $T_i.\mathcal{P} = \{P_1, \dots, P_{m_i}\}$, where m_i is the number of processors assigned to task T_i. Finally, the number of assigned processors, p_i, is stored in the attribute $T_i.p = p_i$.

Algorithm 1 shows how to construct a practical scheduling from a sequence, given by $\sigma(i)$. Thus, the variational freedom for the minimization of the NPTS makespan lies in the possible permutations σ:

$$\min_{\sigma}\{\textsc{GetMakespan}(\textsc{MakeSchedule}(\sigma))\}\,. \tag{5.18}$$

The function MakeSchedule is given by Algorithm 1, and the function GetMakespan by Algorithm 2.

Algorithm 1 The procedure to generate a practical scheduling, stored in the fields of the objects from $\mathcal{T}$ and $\mathcal{P}$, from a permutation σ. The for-loop in lines $4-10$ processes all tasks in the sequence given by σ. The min-function in line 6 looks for those processors, whose stacks have the earliest availability time, and the task is scheduled on those which have first finished according to the current scheduling. Finally, the finalization time of these processors is updated in line 8. This procedure requires $\mathcal{O}(nm)$ steps, n being the number of tasks and m the number of available processors.

```
1:  function MakeSchedule($\sigma$)
2:      Declare a new set $\mathcal{P}$ of size $m$.
3:      Initialize all $P_i.t$ from $\mathcal{P}$ with 0.
4:      for $i = 1 \dots n$ do
5:          $T = T(\sigma(i))$
6:          $T.\mathcal{P} \leftarrow T.p$ processors with $\min_{P_i \in \mathcal{P}}\{P_i.t\}$
7:          for all $P_i \in T_i.P$ do
8:              $P_i.t = \max_{P_i \in T_i.\mathcal{P}}\{P_i.t\} + t_{i,p_i}$
9:          end for
10:     end for
11:     return $\mathcal{P}$
12: end function
```

Algorithm 2 The function yields the makespan of a generated scheduling, stored in the set P. Its execution requires m steps.

```
function GetMakespan($P$)
    return $\max_{P_i \in P}\{P_i.t\}$
end function
```

A sequence of n tasks allows a total of $n!$ possible permutations. Hence, a combinatorial "brute-force" search would require $n! \cdot \mathcal{O}(nm)$ steps. With the assumption of the maximum problem size, $n \leq 10$, the number of permutations to search through would add up to $10! \approx 3.6 \cdot 10^6$ in the worst case. The real time required for the computation of this worst case is still low, in comparison to the time required to solve the actual eigenvalue problem (given a moderate problem size). Section 5.7 compares these times from real applications. However, to lower the probability that this case happens, we introduce two simplifications, in which the problem size can be reduced for most of the common scheduling constellations:

Simplification 1

In the first phase of the algorithm, described in Section 5.4.1, some tasks might be assigned all available processors, $p_i = m$. See, for example, the tasks T_1 and T_2 in Figure 5.2. Clearly, those tasks can be taken out of the combinatorial part of the algorithm and simply scheduled before all other tasks. We therefore define the set

$$\mathcal{T}_m = \{T_i \in \mathcal{T} : p_i = m\}\,, \tag{5.19}$$

which contains all tasks that do not have to be considered by the combinatorial routine, and the set

$$\mathcal{T}_r = \mathcal{T} \setminus \mathcal{T}_m\,, \tag{5.20}$$

which contains all other tasks in the combinatorial algorithm for the minimization problem from Equation (5.18).

Simplification 2

Before Algorithm 1 starts to schedule the tasks from $\mathcal{T}_r$ on the processors from $\mathcal{P}$, all processor stacks are available at an initial time, which we denote as t_0: either the processor stacks are initially empty (if $\mathcal{T}_m = \emptyset$), or they have a task from $\mathcal{T}_m$ on top of their stack and are available after its common execution. In each permutation σ, there is a set of tasks $\mathcal{T}_0 \subseteq \mathcal{T}_r$, $n_0 = |\mathcal{T}_0|$, which are scheduled at t_0 – typically the tasks from the first few iterations. See for example the tasks T_3 and T_4 in Figure 5.2. However, depending on the number of tasks n, the number of available processors m, and the chosen processor allotment $\bar{p}$, these tasks may be the majority or even all tasks from $\mathcal{T}_r$: $1 \leq n_0 \leq |\mathcal{T}_r|$.

The presented simplification is based on the assumption that a different scheduling of the tasks from $\mathcal{T}_0$ does not change the makespan of a scheduling of the tasks from $\mathcal{T}_r$, and consequently of all tasks $\mathcal{T}$. To demonstrate how this assumption helps to reduce the total number of permutations, we take out one random combination and split it up into two parts: $\sigma_0(i) = \sigma(i) : i = 1, \dots, n_0$; $\sigma_p(i) = \sigma(i) : i = n_0 + 1, \dots, n$. Let us assume that σ_0 is fixed and the algorithm searches through all $(n - n_0)!$ permutations of σ_p. Clearly, these permutations constitute a subset of the set of all possible permutations of σ. If now the permutation of σ_0 is changed, a "brute force" approach will go through all possible permutations of σ_p again, so this particular constellation causes a total of $n_0! \cdot (n - n_0)!$ combinations. However, if we assumed that a different permutation of σ_0 does not result in a scheduling with an improved makespan, we would need only one random permutation of σ_0 instead of $n!$, and could, thus, reduce the number of permutations necessary here to $(n - n_0)!$. Note that this works only for one distinct partitioning of σ into σ_0 and σ_p. Once the elements of σ_0 and σ_p are interchanged, the permutations of σ_p have to be computed again.

To show that this assumption is true, we establish the following Theorem:

Theorem 1. *There is an initial time t_0, when no task from $\mathcal{T}_r$ has been scheduled yet and all processors are available. In each iteration i of Algorithm 1, there is a set of tasks $\mathcal{T}_0 \subseteq \mathcal{T}_r$, $|\mathcal{T}_0| = n_0$, which start at t_0. Let σ_0 be the sequence of the tasks from $\mathcal{T}_0$: $\sigma_0(i) = \sigma(i), i = 1, \dots, n_0$. When establishing a scheduling with Algorithm 1, the makespan of the whole set $\mathcal{T}$ is independent from the sequence σ_0.*

The proof of Theorem 1 is given by the following paragraphs and lemmas.

Algorithm 1 schedules each task $T_i \in \mathcal{T}$ one after another on the processors from $\mathcal{P}$ in order of their appearance in σ, see the loop in lines $4 - 10$. The processors assigned to a specific task, $T_i.\mathcal{P}$, refer to those processors, whose stacks have the earliest availability time, $P_j.t$, in this iteration of the loop (line 6). The starting time of T_i, here denoted as t_i^{start}, is the highest availability time of the stacks from $T_i.\mathcal{P}$ (line 8). Furthermore, let t_i^{fin} be the finishing time of T_i:

$$t_i^{\text{fin}} = t_i^{\text{start}} + t_{i,p_i} \,. \tag{5.21}$$

Lemma 1. *After the tasks from $\mathcal{T}_0$ have been scheduled, the finishing time of the processor stacks must be either t_0 or one of their finishing times t_i^{fin}.*

Proof. Each processor stack can only contain either no task or exactly one task. Those which contain no task, are still available at t_0; all others after the execution of the scheduled task, which is at $t_i^{\text{fin}} = t_0 + t_{i,p_i}$. □

Lemma 2. *The finishing time t_i^{fin} of each task $T_i \in \mathcal{T}_0$ is independent from σ_0.*

Proof. There are two parameters which determine the finishing time t_i^{fin}: the starting time t_i^{start} and the assigned processor count p_i. The starting time of all tasks in $\mathcal{T}_0$ is per definition t_0 and hence equal, and the assigned processor count p_i is fixed Phase 2 of the algorithm. $\square$

Lemma 3. *The finishing time t_i^{fin} of each task $T_i \in \mathcal{T} \setminus \mathcal{T}_0$ is independent from σ_0.*

Proof. The time t_i^{fin} is determined in Equation 5.21. The only variable in Phase 2 of the algorithm is the starting time t_i^{start}. Furthermore, t_i^{start} is the availability time of a processor stack from $\mathcal{P}$, which in turn is the finishing time of a previously scheduled task: $t_i^{\text{start}} = t_j^{\text{fin}}, j \in \{\sigma(1), \dots, \sigma(i-1)\}$. Let T_i be the first task scheduled after the tasks from $\mathcal{T}_0$: $T_i = T(\sigma[n_0 + 1])$. Its starting time must be the finishing time of one of the tasks from $\mathcal{T}_0$, according to Lemma 1. These times are independent from σ_0, shown in Lemma 2, hence, t_i^{fin} is also independent from σ_0. Thus, for each task $T = T(\sigma[i])$ scheduled in iteration $i = n_0 + 1, \dots, n$, the finishing time of the previously scheduled tasks are independent from σ_0, and so is the finishing time of this task, t_i^{fin}. $\square$

Clearly, the makespan of the whole scheduling is the finishing time of one of the tasks from $\mathcal{T}$. Lemmas 2 and 3 show that these times are independent from σ_0. This proofs Theorem 1.

How many of the $n!$ permutations of σ can finally be saved by this simplification, depends on the constellation of m, $\bar{p}$, and n. For example, if $\max_i\{p_i\} \ll m$, the cases in which $\mathcal{T}_0$ is large will appear frequently, which reduces the total number of permutations necessary to a minor fraction of $n!$. Section 5.7 also addresses this issue.

The Final Algorithm

Finally, Algorithm 3 brings everything together and solves the NPTS minimization problem from Equation (5.18), according to the scheme presented above.
Figure 5.3 illustrates how this algorithm generates a different makespan from a varying number of available processors m, making, thus, the overall eigensolver step scalable.

5.5. Cost Function

The practical implementation of Algorithm 3 requires a cost function, as given in Equation (5.6) and stated here again for convenience,

$$t : (T_i, p) \mapsto t_{i,p} \,,$$

that works well in practical applications. In our case, this function refers to the *wall-clock time* performance of the employed eigensolver – here the (Sca)LAPACK routines `DSYGV` and `PDSYGV`. However, accurate performance prediction, especially of parallel routines, proves to be difficult. The ScaLAPACK User's Guide [33] proposes a general performance model, which depends on machine-dependent parameters such as floating point

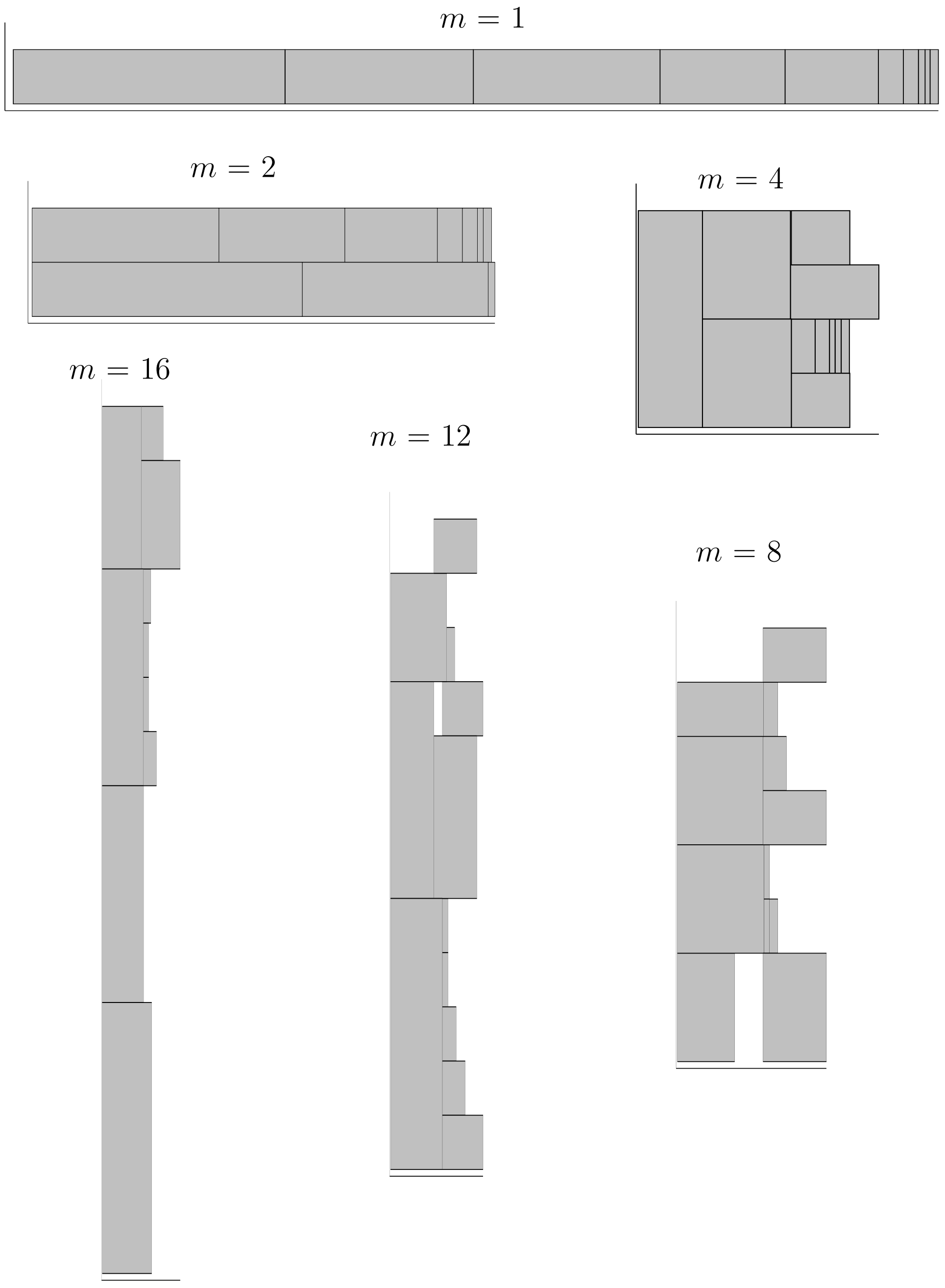


Figure 5.3.: "Strong scaling" illustration of ten tasks on a varying number of processors m. A task is represented by a gray box, with its width being its predicted execution time, and its height indicating the number of processors assigned to it. When all tasks are executed on a single processor ($m = 1$), all tasks are simply executed one after another. On higher processor numbers, the scheduling algorithm starts to distribute the tasks on the available processors, where some of the tasks may also be executed using a parallel eigensolver (starting from $m = 4$). Thus, by employing the proposed scheduling scheme, the overall eigensolver step for all IRREPs has also been made "malleable".

Algorithm 3 Finds the scheduling sequence σ^*, from which MAKESCHEDULE (Algorithm 1) generates a scheduling with the minimum NPTS makespan. An algorithm to generate permutations of a sequence of objects is part of most standard literature on algorithms, and for reasons of brevity not provided explicitly here.

```
Find the tasks which have been assigned $m$ processors and create the sets $\mathcal{T}_m$ and $\mathcal{T}_r$
Schedule $\mathcal{T}_m$ at the beginning
makespan$^* \leftarrow \infty$
for all permutations $\sigma$, except those neglected due to Simplification 2 do
    makespan$_{\text{tmp}} \leftarrow$ GETMAKESPAN(MAKESCHEDULE($\sigma$))
    if makespan$_{\text{tmp}} <$ makespan$^*$ then
        makespan$^* \leftarrow$ makespan$_{\text{tmp}}$
        $\sigma^* \leftarrow \sigma$
    end if
end for
```

performance or network bandwidth, and data- and routine-dependent parameters such as total FLOP or communication count. In [84], Demmel and Stanley used this approach to evaluate the general performance behavior of the ScaLAPACK routine `PDSYEVX`. The validation of the models shows that the accuracy is relatively poor – the prediction error lies between 10 and 30%. Apart from that, for the practical use in a computer program, it exhibits another important drawback: to establish such an analytical model, detailed knowledge of many implementation details is required, such as the underlying numerical algorithms, the data distribution, or the communication patterns. Furthermore, each routine needs its own specific model; thus, if the routine changes (e.g. due to a new revision or the use of a different library), the model has to be adapted and validated as well. Hence, we believe that using an analytical performance model does not result in a good, generic solution.

Here we follow a different approach: the numerical routine—the eigensolver—is treated as a "black box". Predictions of its execution time are based on empirical data, which are recorded by test runs with a set of randomly generated matrices. As Equation 5.6 indicates, the cost function depends on two parameters:

- the task T_i or, more precisely, its size. Since each task corresponds to a symmetric sub-matrix block H^Γ with the relation $\Gamma \mapsto i$, we refer to its size as the sub-matrix dimension N_i. All possible (or considered) dimensions are comprised in the set $\mathcal{N} \subset \mathbb{N}$;

- the number of employed processors, $p_i \in \mathcal{D}$.

Both parameters are discrete. However, practically, the set of possible processor numbers $\mathcal{D}$ is usually much smaller than the set of possible sub-matrix dimensions $\mathcal{N}$. We therefore propose the following approach: for each processor number $p \in \mathcal{D}$ we choose a certain subset of $\mathcal{N}$. On these "points" along the p-axis, we make a test run with a

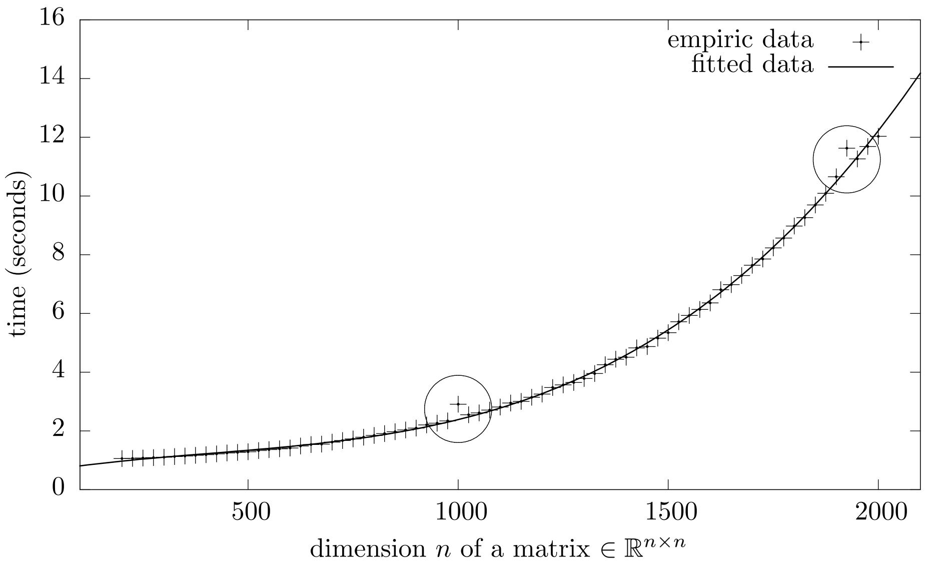


Figure 5.4.: Example benchmark run of the *Cost Function Generator* (see Section 5.6.1) for one $p_i \in \mathcal{D}$. The points labeled "empiric data" refer to the measured wall-clock execution times required by the routine `PDSYGV` to diagonalize randomly generated matrices. The curve labeled "fitted data" represents a polynomial function of degree 3, which models the empiric data and is used by the cost function for execution time predictions.

randomly generated matrix of that size. Then, a continuous cost function t_j is generated using a one-dimensional curve-fitting algorithm. See Figure 5.4 for an example. Thus, each $p_j \in \mathcal{D}$ has a related cost function, which we denote by

$$t_j : N \mapsto t_{p_j,N} \,. \tag{5.22}$$

We use the method of least squares to fit the data, see Chapter 15 of [27] for an introduction. For our purposes, it shows two beneficial properties:

- the data can be fitted by polynomial functions, in our case of degree three. This function type is easy to handle and allows to generate an estimated execution time t_j with low computational effort. The chosen polynomial degree of three refers to the execution time behavior of the eigensolver routine, which scales as $\mathcal{O}(N^3)$;
- during the benchmark run, the processing of a matrix may take longer than expected, e.g. due to unexpected hardware delays (see circular marks in Figure 5.4). As long as those cases are rare, their influence on the cost function is minor and can be neglected.

To keep the effort of the test runs small, we introduce a reduction of the set $\mathcal{D}$: ScaLAPACK uses a two-dimensional block cyclic data distribution. For each instance of a routine, a $m_r \times m_c$ process grid has to be allocated with m_r process rows and m_c

process columns. However, the ScaLAPACK User's Guide [33] suggests to use a square grid ($m_r = m_c = \lfloor\sqrt{m}\rfloor$) for $m \geq 9$ and a one-dimensional grid ($m_r = 1; m_c = m$) for $m < 9$. Following this suggestion results in a reduced set of processor configurations, e.g. $\mathcal{D} = \{1, 2, \dots, 8, 9, 16, 25, \dots, \lfloor\sqrt{m}\rfloor^2\}$, which will be used here.

Finally, we combine the emerging set of p-related cost functions to form the general cost function from Equation (5.6). However, in practice, when a certain number of allotted processors is exceeded, parallel routines no longer feature a speedup or even slow down, see also [85]. This behavior does not comply with the assumption of a general monotonic cost function. To satisfy this constraint, we redefine Equation (5.6) as

$$t_{i,p} = \min_{p_j \in \mathcal{D}} \{t_{p_j,N_i} : p_j \leq p\}\,. \tag{5.23}$$

All possible processor counts $p_j \in \mathcal{D}$ are considered which are smaller than or equal to the primary parameter p. The p_j which results in the smallest execution time for the given N also determines the p-related cost function and thus the result t of the general cost function.

The accuracy requirements on the cost function are difficult to determine upfront. However, the evaluation of the presented method on real chemical systems in Section 5.7 will show that the error is below 10% in all relevant cases, and works well in practice.

5.6. Implementation: Software Components

This section gives an overview of the software, in which we implemented the eigenvalue scheduling technique, introduced in this chapter. It consists of two main components:

Cost Function Generator An independent program, which carries out benchmark runs with the employed eigensolver and creates, consequently, a continuous cost function

Eigenscheduler A Fortran library, to be included into ParaGauss, which provides modules for

1. the generation of a scheduling (the implemented MPTS algorithm);
2. the computation of the eigenvalues and eigenvectors of the Hamiltonian, according to the generated scheduling.

An overview of the software structure is sketched in Figure 5.5. The following Sections 5.6.1 and 5.6.2 describe these main components in detail.

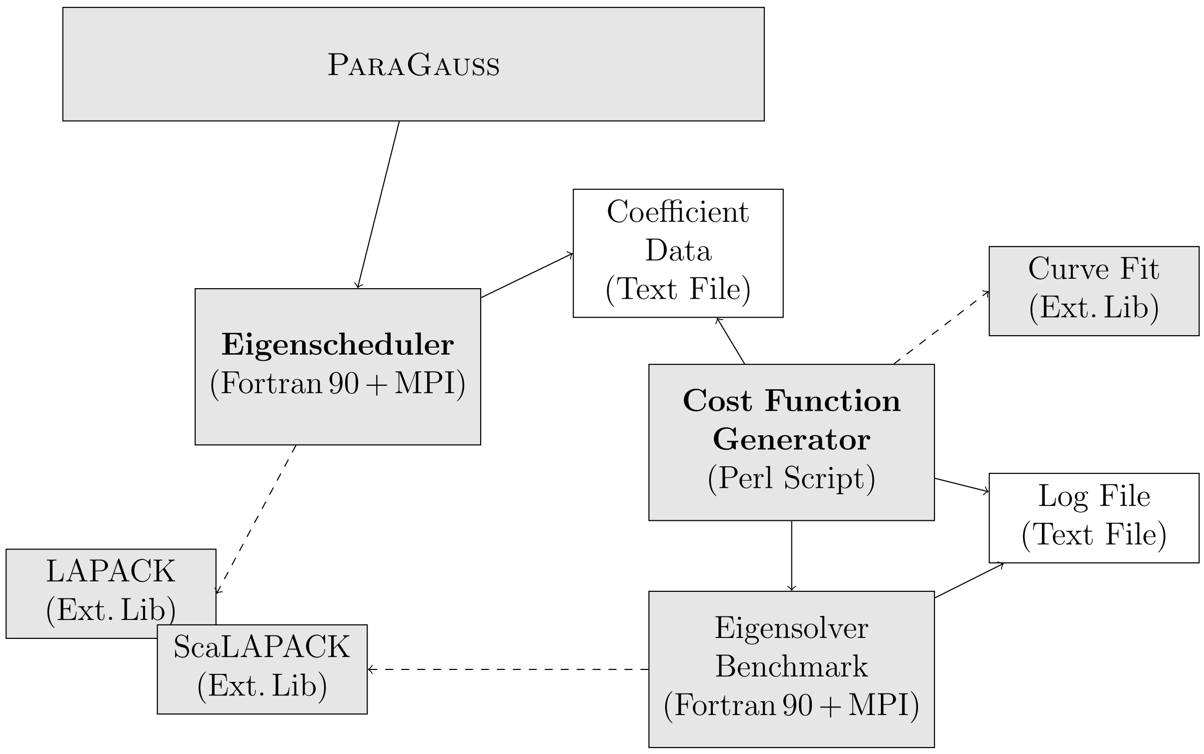

ParaGauss
Coefficient Data (Text File)
Curve Fit (Ext. Lib)
Eigenscheduler (Fortran 90 + MPI)
Cost Function Generator (Perl Script)
Log File (Text File)
LAPACK (Ext. Lib)
ScaLAPACK (Ext. Lib)
Eigensolver Benchmark (Fortran 90 + MPI)

Figure 5.5.: Structure of the software components implementing the introduced eigenscheduler. The *Cost Function Generator* (see Section 5.6.1) script calls a program labeled *Eigensolver Benchmark*. This program executes the benchmark runs of randomly generated matrices (see Figure 5.4) using the external libraries *LAPACK* and *ScaLAPACK*. The timings are stored in an external *Log File*. Then, the script extracts the times from the log files and passes them to a *Curve Fit* routine, which generates the polynomials of degree three. The coefficients of the polynomials are then stored in a text file labeled *Coefficient Data*. The *Eigenscheduler* (see Section 5.6.2) is a module invoked by ParaGauss in each SCF cycle for the solution of the generalized eigenvalue problem in Equation (5.1). The cost function, as part of the eigenscheduler, requires the coefficients, stored in the text file *Coefficient Data*.

5.6.1. Cost Function Generator

As the name indicates, this software component is responsible for the generation of the cost function, required later by the scheduling algorithm. It consists of two components:

- a *Perl* script ("*Cost Function Generator*", see Figure 5.5), which
  - calls the program for the eigensolver benchmark ("*Eigensolver Benchmark*");
  - parses the "*Log File*", generated by the benchmark program;
  - generates polynomials, which are fitted to the benchmark data;
  - writes the polynomial coefficients to an external text file ("*Coefficient Data*").
- a *Fortran 90* program ("*Eigensolver Benchmark*"), which carries out the actual benchmark runs by invoking the external routines `DSYGV` and `PDSYGV`. The measured timings are written to an external "*Log File*".

How To – User Information

To start the cost function generation process, one calls the script

`${PG_DIR}/schedeig/se_runrec/run.sh`,

where the environmant variable `PG_DIR` indicates the ParaGauss root directory. Within the script `run.sh`, there is a number of parameters (plus default values), which allow some adjustments of the cost function generation process. Here we give a list of the available parameters and default values:

`MPIPROCESSES=36` The number of processes which are used to schedule the eigensolver instances on. Can be greater than `MAXPROCCONF`.

`MINPROCCONF=1` The minimum number of processes to be used for a single test run in a benchmark. See also `MAXPROCCONF`.

`MAXPROCCONF=36` The maximum number of processes to be used for a single test run. Corresponds to m. Test runs are recorded for $\mathcal{D} = \{1, 2, \ldots, 8, 9, 16, 25, \ldots, \lfloor\sqrt{m}\rfloor^2\}$.

`BLOCKSIZE=64` Determines the size of the blocks, into which the matrices are split up for their distribution. For more information, we refer to the ScaLAPACK User's Guide [33].

`MINMATSIZE=200` The smallest possible matrix size used in a benchmark run.

`MAXMATSIZE=2500` The greatest matrix size, to which a benchmark run is generated.

`STEPSIZE=25` The step size, in which a test series for a $p \in \mathcal{D}$ proceeds from `MINMATSIZE` to `MAXMATSIZE`.

According to these values, the benchmark program generates a test run for each $p \in \mathcal{D}$, starting from $N = \max\{\texttt{MINMATSIZE}, p_j \cdot \texttt{BLOCKSIZE}\}$, and proceeding to `MAXMATSIZE` in steps of size `STEPSIZE`. In case $\lfloor \texttt{MPIPROCESSES}/p_j \rfloor \geq 2$, the benchmark schedules several instances of test runs from a series at the same time, which speeds up the whole benchmark process significantly.

5.6.2. Eigenscheduler

The Eigenscheduler is the component of the software, which is invoked by ParaGauss to carry out scheduled eigenvalue computations of the Hamiltonian sub-matrices H^Γ. It is an external Fortran 90 library, comprising two main modules and one helper module, which will be described more in detail below. The interface to these modules has been designed such, to allow an easy incorporation into ParaGauss, with few changes to the original code.

se_scheduling_module

This module establishes a static scheduling, according to the algorithm presented in Section 5.4. The scheduling is stored in the data type `se_scheduling_scheduletype`, also declared in this module. The public interface is defined by only one routine:

```
se_scheduling_run(scheduling, taskArray,
                  availableProc, coeffFileName, iostatus).
```

The parameters are:

- `scheduling` (output): is of type `se_scheduling_scheduletype`, and stores the scheduling for further processing in `se_eigen_module`.
- `taskArray` (input): an integer-array, which stores the dimensions of the IRREPs. It is required as input parameter for the algorithm, and corresponds to the set $\mathcal{N}$.
- `availableProc` (input): an integer value, which gives the number of available processors. It is required as input parameter for the algorithm, and corresponds to m.
- `coeffFileName` (input): a string, which contains the path to the coefficient data file.
- `iostatus` (output): an integer value, which contains an error value. 0 indicates success.

The routine must be called collectively by all involved processes. After exit, the scheduling data is provided on the MPI master process (process number 0).

se_eigen_module

This module does the actual eigenvalue computations of the sub-matrices H^{Γ}, according to the previously established scheduling. It requires, thus, a feasible scheduling, stored in a variable of type `se_scheduling_scheduletype`. The public interface is defined by the subroutine

```
se_eigen_compeigs(scheduling, ham_tot,
                overlap, eigval, eigvec, mpi_communicator, blacsExit).
```

The parameters are:

- `scheduling` (input): static scheduling information, previously generated by the routine `se_scheduling_run`, of type `se_scheduling_scheduletype`.
- `ham_tot` (input): The Hamiltonian sub-matrices, refers to H^{Γ}.
- `overlap` (input): The overlap sub-matrices, refers to S^{Γ}.
- `eigval` (output): The computed eigenvalues.
- `eigvec` (output): The computed eigenvectors.
- `mpi_communicator` (input): A valid MPI communicator, which provides at least m processes.
- `blacsExit` (input): A logical value, which determines if the BLACS environment shall be closed after execution (`true`) or left unchanged for further use (`false`).

The data types of the input- and output-matrices, `ham_tot`, `overlap`, and `eigvec`, as well as the output vector `eigval`, are the ParaGauss-specific data structures `arrmat2` and `arrmat3`. The routine is called by all processes in the same manner – a differentiation e.g. between a master- and a slave process is not necessary. However, it is assumed that the input matrices `ham_tot` and `overlap` are provided by the MPI master (process number 0), which also holds the output data `eigval` and `eigvec` after the routine has terminated.

5.7. Performance Results

We evaluated the presented scheduler on four molecular systems of medium and large size as example applications: the gold cluster compound $Au_{55}(PH_3)_{12}$ in symmetry S_6 (for brevity also referred to as Au_{55}) and the palladium clusters Pd_{344}, Pd_{489} and Pd_{670}, all in symmetry O_h. The symmetry S_6 results in four large IRREPs with few variation in size, and the O_h symmetry in ten blocks, whose size varies much more. Thus, the latter system states a more difficult scheduling problem. Table 5.1 lists the dimensions

S_6	Au_{55}
PGC	N_i
A_g	782
E_g	1556
A_u	782
E_u	1560

O_h	Pd_{344}	Pd_{489}	Pd_{670}
PGC	N_i	N_i	N_i
A_{1g}	199	652	857
A_{2g}	317	447	622
E_g	513	1093	1473
T_{1g}	838	1204	1680
T_{2g}	956	1370	1870
A_{1u}	1110	305	432
A_{2u}	317	447	622
E_u	471	749	1048
T_{1u}	785	1550	2105
T_{2u}	956	1366	1870

Table 5.1.: The resulting point group classes (PGC) of the example systems $Au_{55}(PH_3)_{12}$ in symmetry S_6, and Pd_{344}, Pd_{489} and Pd_{670} in symmetry O_h.

of the resulting sub-matrix blocks.

The examples were executed on two different computer systems, both installed at the Leibniz Rechenzentrum[1], with different hardware characteristics. Test platform for the examples Au_{55} and Pd_{344} was the former national supercomputer HLRB2, which is an Altix 4700 from SGI. The system uses between one and two Intel Itanium2 Montecito Dual Cores as CPUs on one compute node, and has an SGI NUMAlink 4 as interconnect. Each core has 4 GByte of memory available. The numerical library SCSL from SGI was used to provide BLAS, LAPACK, BLACS and ScaLAPACK support.

For the test runs of the palladium clusters Pd_{489} and Pd_{670}, the migration system SuperMIG, built by IBM, was used, which provides a more recent supercomputer architecture. The machine contains 205 nodes, each hosting four 10-core Intel Xeon Westmere-EX processors. The nodes are equipped with 256 GB of memory and are interconnected by an Infiniband QDR network. For the benchmarks we employed the default vendor supplied MPI library implementation, IBM MPI v5.2, and BLACS/ScaLAPACK v1.8, linking to the BLAS library distributed with Intel's MKL v10.3. For further details on the hard- and software specification of HLRB2 and SuperMIG, please refer to [86].

We measured the execution times of the complete eigensolver step in a *single* SCF iteration, see Figure 5.6. Technically, this refers to a call to the routine `se_eigen_compeigs` (see Section 5.6.2). Recall that typical quantum chemical applications require between 10^3 and 10^5 eigenvalue computations. This should be taken into consideration when

[1]www.lrz.de

examining the real time savings achieved by this parallelization technique.

Figure 5.6 shows that the cost function works accurate in most of the cases, with an error well below 10%. Interestingly, this does not apply to small processor numbers ($m = \{1, 2\}$), where the error is quite significant (up to $\approx 40\%$ in the Pd_{670} test runs). Fortunately, these cases still provide good scheduling results, even with a poorly accurate execution time predictions. In the cases where m is higher, the cost function works sufficiently accurate to facilitate the practical use of the scheduling algorithm.

The figure also shows a lower bound on the execution time of a sequential scheduler ("LPT"-line). To recapitulate the basic idea of the previously used LPT-scheduler: all matrices are sorted by their size and accordingly scheduled on any processor which becomes available. There the matrix is diagonalized by a sequential (LAPACK) eigensolver routine (see Figure 5.1). However, this performance bound is now broken and the execution time is improved below this barrier by our new algorithm, as the next paragraph will show.

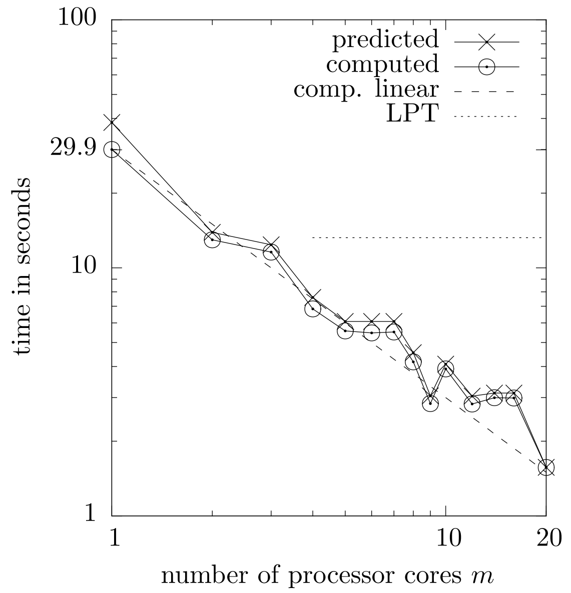
Test runs were performed employing up to 80 and 120 processor cores on SuperMIG for the test systems Pd_{489} and Pd_{670}, respectively. On HLRB2, timings were recorded using up to 20 and 28 cores for the test systems Au_{55} and Pd_{344}, respectively. Beyond the processor numbers presented, no significant speedup could be reached anymore. The graphs in Figure 5.6 show that the previously existing barrier, implied by the LPT-scheduler and indicated by the line labeled "LPT", could be broken, with significantly lower execution times: on SuperMIG, the diagonalization step of the test system Pd_{489} was sped up by a factor of up to 11.1, compared to LPT, and by a factor of 31.6 compared to a sequential run. For the system Pd_{670} we achieved a maximum speedup factor of 10.5 and 40, compared to LPT and a sequential run, respectively. For the two systems tested on HLRB2, Au_{55} and Pd_{344}, the execution times were improved by a factor of 8.4 and 4.1, respectively, compared to the LPT-approach, and by factors of 19 and 13.8, respectively, compared to a sequential run. For all test systems, the execution time of the overall eigensolver step now lies well below one second, except for Au_{55}, where the execution requires 1.6 seconds. Thus, we have shown that the execution time of this important step can be reduced down to a minor fraction of time, compared to the overall electronic structure calculation.

Figure 5.7 shows the parallel efficiency of the scheduled eigensolver step, according to the times given in Figure 5.6. One can see that the test system Au_{55} in S_6 symmetry shows overall a good efficiency, sometimes greater than 1 and never worse than 0.6. In contrast, the other systems in O_h symmetry, Pd_{344}, Pd_{489} and Pd_{670}, show a much worse efficiency at higher processor numbers: above 8 processors, the efficiency drops below 0.6 in all test systems. This shows that a symmetry such as S_6 comprises a submatrix structure much easier to handle for our scheduler: there are only four blocks which are relatively large, and relatively equal in size (2 blocks of dimension ≈ 1560 and two blocks of ≈ 780). Here, the parallel solver can parallelize the two large blocks efficiently (its efficiency is always above 0.9), only the idle times cause some efficiency loss.

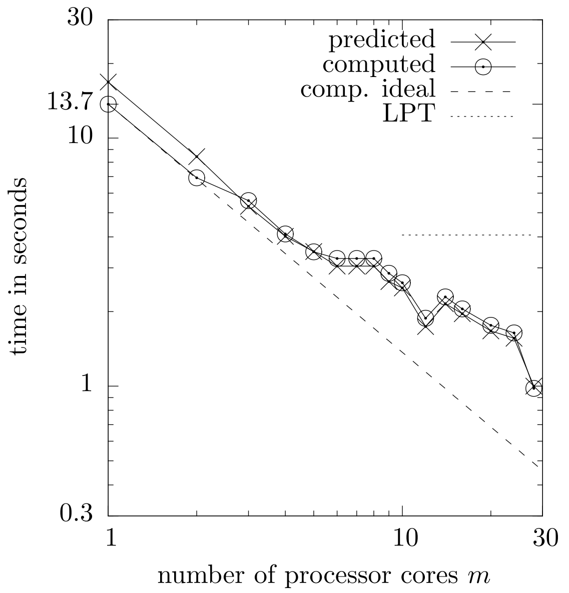
The processor core numbers shown in this evaluation section are relatively small, compared to other eigenvalue computations in electronic structure codes [87]. However, the chemical systems used for our benchmarks are among the largest systems used in practical *ab initio* DFT calculations. In these examples, the computational effort has been reduced significantly by exploiting the molecular symmetry, which also results in comparatively small matrix sizes. The computational workload could be increased by using a lower or no symmetry at all, resulting in much larger core numbers. However, the scalability of these calculations would depend (almost) solely on the employed eigensolver. The improvement of parallel eigensolvers is an active research field in scientific computing [87, 88], but shall not be discussed in this work.

One also has to consider the costs of establishing the scheduling, mainly implied by the combinatorial Phase 2 of the algorithm (see Section 5.4.2). This applies especially, if the chosen point group exhibits high symmetry, such as O_h, where ten IRREPs are involved (see Table 5.1). The two simplifications, introduced in Section 5.4.2, make the cases, where it is necessary to go through all 10! permutations, very rare – none of our 47 test runs created this worst case. The wall-clock time required for the algorithm was always clearly below one second, in most cases a few milliseconds. However, we provide a "rule of thumb" to estimate the costs of the worst case: in our Fortran 90 implementation, executed on the SuperMIG computer system [86], each considered processor caused a computing time of roughly one second if the algorithm has to go through all 10! combinations, so $t_{\text{worst}} \approx m$ seconds.

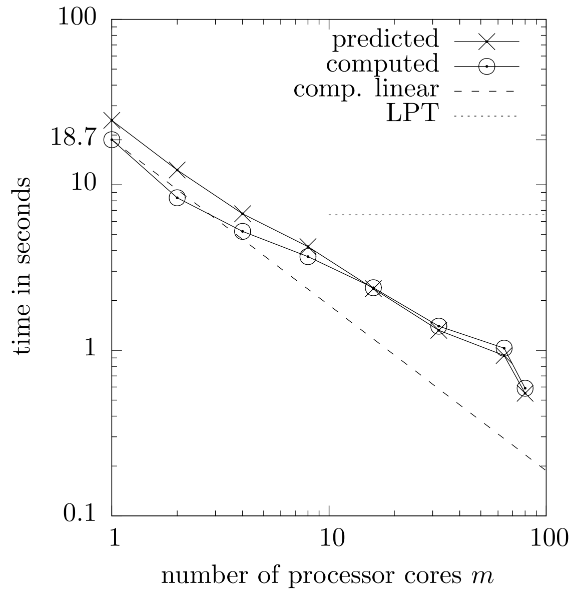
To finally summarize what the technique presented in this chapter has achieved: a parallel bottleneck of the eigensolver in ParaGauss, imposed by the previous "LPT"-based parallelization approach, could be eliminated. This was achieved by employing a more sophisticated "MPTS" scheduler, together with parallel eigensolver routines from ScaLAPACK. The overall scalability of the eigensolver step was significantly improved, being now able to use processor core numbers up to 120 efficiently. The time spent in this step was reduced to a minor fraction of what was necessary before, requiring now less than one second in one SCF iteration for most of the test cases considered in this section. This approach also goes beyond the use of parallel eigensolvers in other Gaussian-based DFT codes [87, 89]: to our knowledge, it is the first technique, which allows an efficient parallel treatment of Hamilton matrices with a block-diagonal structure. Thus, DFT-based methods which achieve computational benefits by exploiting molecular symmetries (see Section 3.2) have now been augmented with a specific, efficient, parallelization technique.



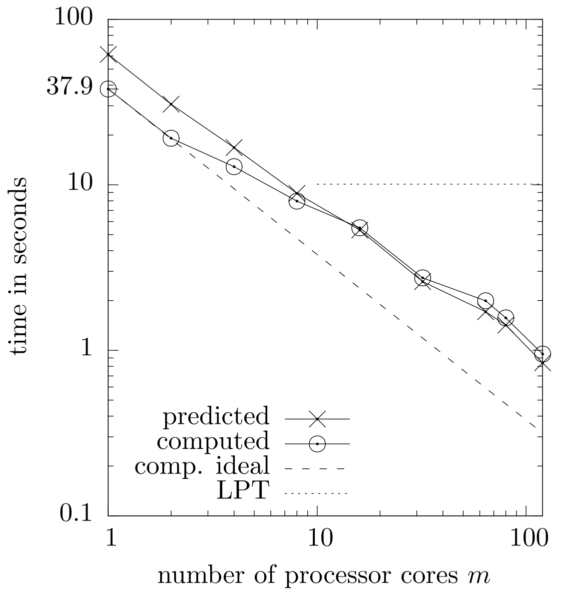
(a) $Au_{55}(PH_3)_{12}$ in S_6 on HLRB2



(b) Pd_{344} in O_h on HLRB2

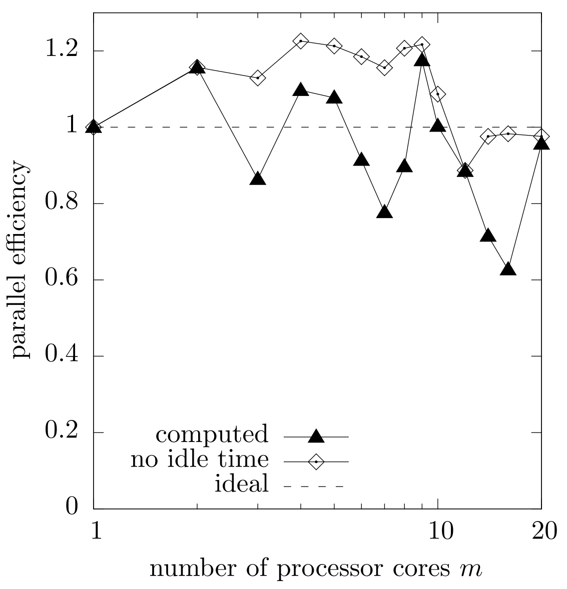


(c) Pd_{489} in O_h on SuperMIG

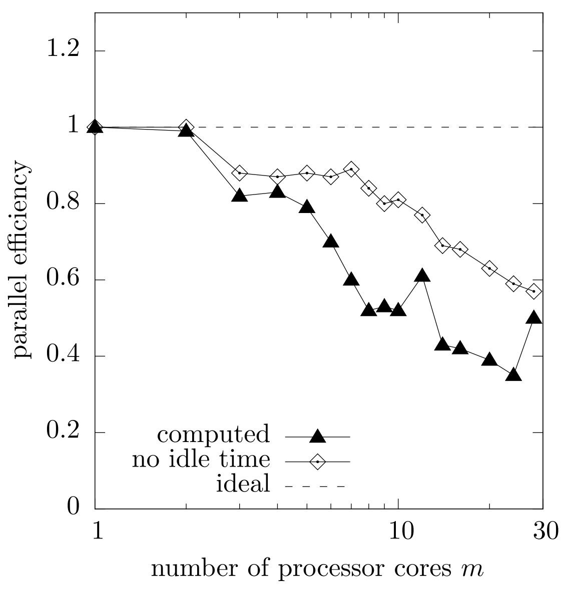


(d) Pd_{670} in O_h on SuperMIG

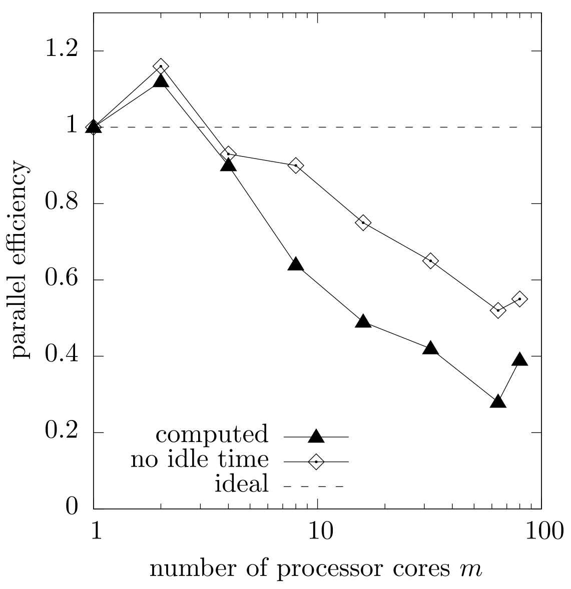
Figure 5.6.: Log-log time diagrams of the complete eigensolver step of the four test systems $Au_{55}(PH_3)_{12}$, Pd_{344}, Pd_{489} and Pd_{670}. Considered is the wall-clock time of the diagonalization module during one SCF iteration. The curves labeled "predicted" show the makespan of the scheduling algorithm, predicted by the cost function. The curves labeled "computed" provide the real execution time of the scheduled eigensolvers. The lines labeled "LPT" indicate the execution time of the sequential LAPACK routine computing the largest matrix from $\mathcal{N}$ and yield, thus, the best possible performance of the previously used LPT-scheduler.



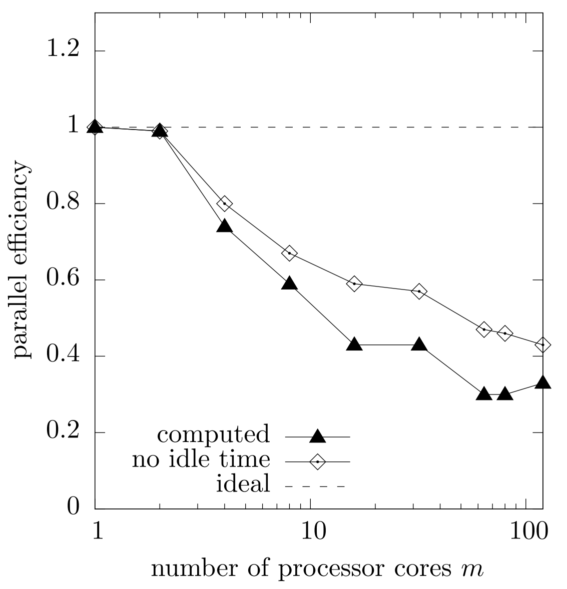
(a) $Au_{55}(PH_3)_{12}$ in S_6 on HLRB2



(b) Pd_{344} in O_h on HLRB2



(c) Pd_{489} in O_h on SuperMIG



(d) Pd_{670} in O_h on SuperMIG

Figure 5.7.: Semi-log diagrams of the parallel efficiency of the complete eigensolver step of the four test systems $Au_{55}(PH_3)_{12}$, Pd_{344}, Pd_{489} and Pd_{670}. The curves labeled "computed" show the parallel efficiency according to the real execution time required by the scheduled eigensolvers and correspond to the timing curves labeled "computed" in Figure 5.6. The curves labeled "no idle time" show the same efficiency, in which the idle time of a scheduling is subtracted from the execution time. Thus, it indicates the efficiency behavior only of the eigensolvers. The difference between "computed" and "no idle time" can be interpreted as the efficiency loss due to idle times in the scheduling. Finally, the line labeled "ideal" indicates the efficiency in case of a linear speedup.

6. Summary

In this thesis, we presented advancements in computational methods, addressing parallelization problems arising in Gaussian-based density-functional software for chemical applications. These advancements are accompanied by implementations, incorporated in the quantum chemistry software ParaGauss. In particular, we presented a parallel programming interface, which facilitates easy data management and the expression of matrix operations in pseudo-mathematical notation. We demonstrated that this technique is suitable for the expression of parallel relativistic transformations, implemented in ParaGauss. The resulting code has indeed the appearance of the original abstract mathematical formulation, and provides a clear and comprehensible representation of the original semantics. The adoption of the existing, sequential, implementation of the relativistic transformations was possible with only minor changes to the original code. We furthermore showed that with our implementation, which partly relies on the performance-optimized parallel libraries PBLAS and ScaLAPACK, the relativistic transformations scale up to 81 cores for input matrices of dimension 4000, requiring now less than one second for the overall relativistic transformation. This states a major improvement compared to the previous sequential implementation. Consequently, this technique brings together programming productivity, code quality and parallel performance, which we also consider a useful contribution to software engineering in high-performance computing.

Furthermore, a parallel bottleneck of the eigenvalue solver step in ParaGauss, imposed by the previous LPT–based parallelization approach, could be eliminated. This was achieved by employing a sophisticated MPTS scheduler, together with parallel routines from ScaLAPACK. The overall scalability of the eigenvalue solver step was significantly improved, being now able to use processor core numbers up to 120 efficiently in large-scale chemical applications. The time spent in this step was reduced to a minor fraction of what was necessary for the previous solution, requiring less than one second in one SCF iteration for all except for one of the test cases considered. This approach also goes beyond the use of parallel eigenvalue solvers in other Gaussian-based DFT codes [6, 89]: to our knowledge, it is the first technique that allows an efficient parallel treatment of dense Hamilton matrices with a block-diagonal structure. Thus, Gaussian-based DFT methods which achieve computational benefits by exploiting molecular symmetries are now augmented with an efficient parallelization technique.

A. Atomic Units

In quantum mechanics, quantities are usually measured in atomic units (a.u.), given as multiples of fundamental constants. This appendix gives a list of all relevant constants and their translation to the more common *International System of Units (SI)*.

Quantity	**Name**	**Symbol**	**SI unit**
Action	Reduced Planck's constant	$\hbar$	$1.0546 \times 10^{-34} Js$
Length	Bohr radius	a_0	$5.2918 \times 10^{-11} m$
Energy	Hartree	E_h	$4.3597 \times 10^{-18} J$
Charge	Elementary charge	e	$1.6022 \times 10^{-19} C$
Mass	Electron mass	m_e	$9.1094 \times 10^{-31} kg$

The energy is sometimes also given in other units, e.g. in Rydberg or electron Volts:

$$1 \text{ Hartree} = 2 \text{ Rydberg} = 27.2114\, eV$$

The equations in Chapter 2 are all given without units, by expressing quantities as multiples of the above given constants. For example, the mass $m = 7.2$ implicitly refers to a multiple of the electron mass, $7.2\, a.u. = 7.2\, m_e$.

Other Constants

Quantity	**Name**	**Symbol**	**SI unit**
Velocity	Speed of light	c	299,792,458 m/s

Bibliography

[1] M. R. Benioff and E. D. Lazowska. Computational science: Ensuring Americas competitiveness. Technical report, Presidents Information Technology Advisory Committee (PITAC), June 2005.

[2] Cern experiments observe particle consistent with long-sought higgs boson. CERN Press Release, July 2012.

[3] Kommission für Informatik. Jahrbuch 2011. Technical report, Leibniz Rechenzentrum, München, Germany, 2011.

[4] E. Schrödinger. An Undulatory Theory of the Mechanics of Atoms and Molecules. *Physical Review*, 28:1049–1070, December 1926.

[5] Tom Geller. Supercomputing's exaflop target. *Commun. ACM*, 54(8):16–18, August 2011.

[6] T. Auckenthaler, V. Blum, H.-J. Bungartz, T. Huckle, R. Johanni, L. Krämer, B. Lang, H. Lederer, and P. R. Willems. Parallel solution of partial symmetric eigenvalue problems from electronic structure calculations. *Parallel Computing*, 37:783–794, 2011.

[7] J. Dongarra. The linpack benchmark: An explanation. In *Proceedings of the 1st International Conference on Supercomputing*, pages 456–474. Springer-Verlag, 1988.

[8] T. Belling, T. Grauschopf, S. Krüger, F. Nörtemann, M. Staufer, M. Mayer, V. A. Nasluzov, U. Birkenheuer, A. Hu, A. V. Matveev, A. V. Shor, M. S. K. Fuchs-Rohr, K. M. Neyman, D. I. Ganyushin, T. Kerdcharoen, A. Woiterski, A. B. Gordienko, S. Majumder, and N. Rösch. PARAGAUSS, version 3.1. Technische Universität München, 2006.

[9] T. Belling, Thomas Grauschopf, S. Krueger, M. Mayer, F. Noertemann, S. Staufer, Christoph Zenger, and N. Roesch. Quantum chemistry on parallel computers: Concepts and results of a density functional model. *High Performance Scientific and Engineering Computing*, 8:441–455, 1999.

[10] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel virtual machine: a users' guide and tutorial for networked parallel computing.* MIT Press, Cambridge, MA, USA, 1994.

[11] Message Passing Interface Forum. Mpi: A message-passing interface standard, version 2.2. Specification, September 2009.

[12] W. Koch and M.C. Holthausen. *A chemist's guide to density functional theory.* Wiley-VCH, 2000.

[13] A. Szabo and N. S. Ostlund. *Modern quantum chemistry.* Dover Publishing, Mineola, New York, 1996.

[14] M. Valiev, E.J. Bylaska, N. Govind, K. Kowalski, T.P. Straatsma, H.J.J. Van Dam, D. Wang, J. Nieplocha, E. Apra, T.L. Windus, and W.A. de Jong. Nwchem: A comprehensive and scalable open-source solution for large scale molecular simulations. *Computer Physics Communications*, 181(9):1477 – 1489, 2010.

[15] P. Hohenberg and W. Kohn. Inhomogeneous electron gas. *Phys. Rev.*, 136:B864–B871, November 1964.

[16] W. Kohn and L. J. Sham. Self-consistent equations including exchange and correlation effects. *Phys. Rev.*, 140:A1133–A1138, November 1965.

[17] R. G. Parr and W. Yang. *Density-Functional Theory of Atoms and Molecules.* Oxford University Press, New York, 1989.

[18] D. R. Bowler and T. Miyazaki. $\mathcal{O}(n)$ methods in electronic structure calculations. *Reports on Progress in Physics*, 75(3):036503, 2012.

[19] M. J. Frisch, G. W. Trucks, H. B. Schlegel, G. E. Scuseria, M. A. Robb, J. R. Cheeseman, G. Scalmani, V. Barone, B. Mennucci, G. A. Petersson, H. Nakatsuji, M. Caricato, X. Li, H. P. Hratchian, A. F. Izmaylov, J. Bloino, G. Zheng, J. L. Sonnenberg, M. Hada, M. Ehara, K. Toyota, R. Fukuda, J. Hasegawa, M. Ishida, T. Nakajima, Y. Honda, O. Kitao, H. Nakai, T. Vreven, J. A. Montgomery, Jr., J. E. Peralta, F. Ogliaro, M. Bearpark, J. J. Heyd, E. Brothers, K. N. Kudin, V. N. Staroverov, R. Kobayashi, J. Normand, K. Raghavachari, A. Rendell, J. C. Burant, S. S. Iyengar, J. Tomasi, M. Cossi, N. Rega, J. M. Millam, M. Klene, J. E. Knox, J. B. Cross, V. Bakken, C. Adamo, J. Jaramillo, R. Gomperts, R. E. Stratmann, O. Yazyev, A. J. Austin, R. Cammi, C. Pomelli, J. W. Ochterski, R. L. Martin, K. Morokuma, V. G. Zakrzewski, G. A. Voth, P. Salvador, J. J. Dannenberg, S. Dapprich, A. D. Daniels, Ö. Farkas, J. B. Foresman, J. V. Ortiz, J. Cioslowski, and D. J. Fox. Gaussian 09 Revision A.1. Gaussian Inc. Wallingford CT 2009.

[20] TURBOMOLE V6.4 2012, a development of University of Karlsruhe and Forschungszentrum Karlsruhe GmbH, 1989-2007, TURBOMOLE GmbH, since 2007; available from `http://www.turbomole.com`.

[21] G. Kresse and J. Furthmüller. Efficient iterative schemes for ab initio total-energy calculations using a plane-wave basis set. *Phys. Rev. B*, 54:11169, 1996.

[22] P. Giannozzi, S. Baroni, N. Bonini, M. Calandra, R. Car, C. Cavazzoni, D. Ceresoli, G. L. Chiarotti, M. Cococcioni, I. Dabo, A. Corso, S. Gironcoli, S. Fabris, G. Fratesi, R. Gebauer, U. Gerstmann, C. Gougoussis, A. Kokalj, M. Lazzeri, L. Martin-Samos, N. Marzari, F. Mauri, R. Mazzarello, S. Paolini, A. Pasquarello, L. Paulatto, C. Sbraccia, S. Scandolo, G. Sclauzero, A. Seitsonen, A. Smogunov, P. Umari, and R. Wentzcovitch. Quantum espresso: a modular and open-source software project for quantum simulations of materials. *Journal of Physics: Condensed Matter*, 21(39):395502, 2009.

[23] Y. Saad, J. Chelikowsky, and S. Suzanne. Numerical methods for electronic structure calculations. *SIAM Rev.*, 52:3–54, March 2010.

[24] P Pulay. Convergence acceleration of iterative sequences. the case of scf iteration. *Chemical Physics Letters*, 73(2):393–398, 1980.

[25] J. M. Soler, E. Artacho, J. D. Gale, A. García, J. Junquera, P. Ordejón, and D. Sánchez-Portal. The siesta method for ab initio order-n materials simulation. *J. Phys.: Condens. Matter*, 14:2745–2779, 2002.

[26] G. T. Velde, F. M. Bickelhaupt, E. J. Baerends, C. F. Guerra, S. J. A. Van Gisbergen, J. G. Snijders, and T. Ziegler. Chemistry with adf. *Journal of Computational Chemistry*, 22(9):931–967, 2001.

[27] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes 3rd Edition: The Art of Scientific Computing.* Cambridge University Press, New York, NY, USA, 3 edition, 2007.

[28] S. Krüger, S. Vent, F. Nortemann, M. Staufer, and N. Rösch. The average bond length in pd clusters pd[sub n], n = 4–309: A density-functional case study on the scaling of cluster properties. *The Journal of Chemical Physics*, 115(5):2082–2087, 2001.

[29] B. I. Dunlap and N. Rösch. The gaussian-type orbitals density-functional approach to finite systems, in: Density functional theory of many-fermion systems. *Adv. Quantum Chem.*, 21:317–399, 1990.

[30] B. I. Dunlap, N. Rösch, and S. B. Trickey. Variational fitting methods for electronic structure calculations. *Molecular Physics: An International Journal at the Interface Between Chemistry and Physics*, 108:3167–3180, 2010.

[31] K. Eichkorn, O. Treutler, H. Ohm, M. Haser, and R. Ahlrichs. Auxiliary basis sets to approximate Coulomb potentials. *Chemical Physics Letters*, 240(4):283–289, June 1995.

[32] 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.* SIAM, Philadelphia, PA, third edition, 1999.

[33] L. S. Blackford, J. Choi, A. Cleary, E. D'Azeuedo, J. Demmel, I. Dhillon, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, R. C. Whaley, and J. Dongarra. *ScaLAPACK user's guide.* SIAM, Philadelphia, 1997.

[34] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. Efficient management of parallelism in object-oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhauser Press, 1997.

[35] R. A. van de Geijn. *Using PLAPACK: Parallel Linear Algebra Package.* The MIT Press, 1997.

[36] A. D. Becke. A multicenter numerical integration scheme for polyatomic molecules. *The Journal of Chemical Physics*, 88(4):2547, 1988.

[37] F. P. Preparata and M. I. Shamos. *Computational geometry: an introduction.* Springer-Verlag New York, Inc., New York, NY, USA, 1985.

[38] V.I. Lebedev. Values of the nodes and weights of ninth to seventeenth order gauss–markov quadrature formulae invariant under the octahedron group with inversion. *USSR Computational Mathematics and Mathematical Physics*, 15(1):44–51, 1975.

[39] V. I. Lebedev and D. N. Laikov. A quadrature formula for the sphere of the 131st algebraic order of accuracy. *Doklady Mathematics*, 59(3):477–481, 1999.

[40] M. Mayer. *A Parallel Implementation of the Density Functional Method: Implementation of the Two-Component Douglas-Kroll-Hess Method and Application to Relativistic Effects in Heavy Element Chemistry.* PhD thesis, Technische Universität München, 1999.

[41] A. V. Matveev. *ParaGauss – A Parallel Implementation of the Density Functional Method: Spin-Orbit Interaction in the Douglas-–Kroll-–Hess Approach and a Novel Two-Component Treatment of Spin-Independent Interaction Terms.* PhD thesis, Technische Universität München, 2004.

[42] A V. Matveev, M. Mayer, and N. Rösch. Efficient symmetry treatment for the nonrelativistic and relativistic molecular Kohn–Sham problem. the symmetry module of the program ParaGauss. *Computer Physics Communications*, 160(2):91 – 119, 2004.

[43] Derek F. Holt, Bettina Eick, and Eamonn A. O'Brien. *Handbook of Computational Group Theory (Discrete Mathematics and Its Applications).* Chapman and Hall/CRC, 1 edition, January 2005.

[44] M. Reiher and A. Wolf. *Relativistic Quantum Chemistry: The Fundamental Theory of Molecular Science.* John Wiley & Sons, 2009.

[45] P. A. M. Dirac. The quantum theory of the electron. *Proc. R. Soc. London*, 117(778):610, 1928.

[46] M. Douglas and N. M. Kroll. Quantum electrodynamical corrections to the fine structure of helium. *Ann. Phys. (NY)*, 82:89, 1974.

[47] Notker Rösch and Oliver D. Häberlen. Reply to the comment on: Relativistic linear combination of gaussian-type orbitals density functional method based on a two-component formalism with external field projectors. *The Journal of Chemical Physics*, 96(8):6322–6323, 1992.

[48] R. J. Buenker, P. Chandra, and B. A. Hess. Matrix representation of the relativistic kinetic energy operator: Two-component variational procedure for the treatment of many-electron atoms and molecules. *Chem. Phys.*, 84:1–9, 1984.

[49] M. Reiher and A. Wolf. Exact decoupling of the dirac hamiltonian. II. the generalized Douglas–Kroll–Hess transformation up to arbitrary order. *J. Chem. Phys.*, 121:10945, 2004.

[50] C. Wüllen. Relation between different variants of the generalized Douglas–Kroll transformation through sixth order. *J. Chem. Phys.*, 120:7307, 2004.

[51] V. A. Nasluzov and N. Rösch. Density functional based structure optimization for molecules containing heavy elements: analytical energy gradients for the Douglas–Kroll–Hess scalar relativistic approach to the LCGTO-DF method. *Chem. Phys.*, 210:413, 1996.

[52] J. M. Seminario. *Recent Developments And Applications Of Modern Density Functional Theory.* Theoretical and Computational Chemistry. Elsevier, 1996.

[53] A. V. Matveev and N. Rösch. The electron-electron interaction in the Douglas–Kroll–Hess approach to the Dirac–Kohn–Sham problem. *J. Chem. Phys.*, 118:3997–4012, 2003.

[54] A. V. Matveev and N. Rösch. Atomic approximation to the projection on electronic states in the Douglas–Kroll–Hess approach to the relativistic Kohn–Sham method. *J. Chem. Phys.*, 128:244102, 2008.

[55] J3 The Fortran Standard Commite. J3/97-007r2. Technical report, http://www.fortran.com/, 1997. Working Draft of the Fortran 95 Standard.

[56] R. C. Whaley and A. Petitet. Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software: Practice and Experience*, 35(2):101–121, February 2005.

[57] C. Sanderson. Armadillo: An open source C++ linear algebra library for fast prototyping and computationally intensive experiments. Technical report, NICTA, Australia, October 2010.

[58] B. S. Ling. *The Boost C++ Libraries*. XML Press, 2011.

[59] P. Gottschling, D. S. Wise, and M. D. Adams. Representation-transparent matrix algorithms with scalable performance. In *Proceedings of the 21st annual international conference on Supercomputing*, ICS '07, pages 116–125, New York, USA, 2007. ACM.

[60] G. W. Stewart. Matran: A Fortran 95 matrix wrapper. Technical report, UMIACS, 2003.

[61] M. Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Trans. Comput.*, C-21:948+, 1972.

[62] Enhanced data type facilities. Technical report, ISO/IEC TR 15581, Second edition, 2001.

[63] G. H. Golub and C. F. van Van Loan. *Matrix Computations (Johns Hopkins Studies in Mathematical Sciences)(3rd Edition)*. The Johns Hopkins University Press, 3rd edition, October 1996.

[64] J. Demmel, J. Dongarra, A. Ruhe, and H. van der Vorst. *Templates for the solution of algebraic eigenvalue problems: a practical guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2000.

[65] I. S. Dhillon and B. N. Parlett. Multiple representations to compute orthogonal eigenvectors of symmetric tridiagonal matrices. *Linear Algebra and Appl*, 387:1–28, 2004.

[66] E. R. Davidson. The iterative calculation of a few of the lowest eigenvalues and corresponding eigenvectors of large real-symmetric matrices. *J. Comput. Phys.*, 17:87, 1975.

[67] Elemental: A new framework for distributed memory dense matrix computations. *ACM Transactions on Mathematical Software*. To appear.

[68] J. A. Gunnels, F. G. Gustavson, G. M. Henry, and R. A. van de Geijn. Flame: Formal linear algebra methods environment. *ACM Trans. Math. Softw.*, 27(4):422–455, December 2001.

[69] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov. Numerical linear algebra on emerging architectures: The plasma and magma projects. *Journal of Physics: Conference Series*, 180(1), 2009.

[70] L. Graham, R.˙ Bounds on multiprocessing timing anomalities. *SIAM J. Appl. Math.*, 17:263–269, 1969.

[71] L. Graham, R.˙ Bounds for certain multiprocessing anomalies. *Bell Syst. Tech. J.*, 45:1563–1581, 1966.

[72] M. Roderus, A. Berariu, H.-J. Bungartz, S. Krüger, A. V. Matveev, and N. Rösch. Scheduling parallel eigenvalue computations in a quantum chemistry code. In *Euro-Par (2)'10*, pages 113–124, 2010.

[73] J. Blazewicz, K. Ecker, E. Pesch, G. Schmidt, and J. Weglarz. *Handbook on scheduling: from theory to applications.* Springer, Heidelberg, 2007.

[74] J. Blazewicz, M. Y. Kovalyov, M. Machowiak, D. Trystram, and J. Weglarz. Scheduling malleable tasks on parallel processors to minimize the makespan. *Ann. Oper. Res.*, 129:65–80, 2004.

[75] G. Mounié, C. Rapine, and D. Trystram. Efficient approximation algorithms for scheduling malleable tasks. In *SPAA '99*, pages 23–32, 1999.

[76] M. Garey and D. Johnson. *Computers and intractability: a guide to the theory of NP-completeness.* W.H. Freeman and Company, 1979.

[77] J. Turek, J. Wolf, and P. Yu. Approximate algorithms for scheduling parallelizable tasks. In *SPAA'92*, pages 323–332, 1992.

[78] W. Ludwig and P. Tiwari. Scheduling malleable and nonmalleable parallel tasks. In *SODA '94*, pages 167–176, 1994.

[79] G. Mounié, C. Rapine, and D. Trystram. A 3/2-approximation algorithm for scheduling independent monotonic malleable tasks. *SIAM J. Comp.*, 37(2):401–412, 2007.

[80] A. Steinberg. A strip-packing algorithm with absolute performance bound 2. *SIAM J. Comp.*, 26(2):401–409, 1997.

[81] K. Jansen. Scheduling malleable parallel tasks: an asymptotic fully polynomial time approximation scheme. *Algorithmica*, 39:59–81, 2004.

[82] T. Decker, T. Lücking, and B. Monien. A 5/4-approximation algorithm for scheduling identical malleable tasks. *Theor. Comput. Sci.*, 361(2):226–240, 2006.

[83] S. L. Altmann and P. Herzig. *Point-group theory tables.* Clarendon, Oxford, 1994.

[84] J. Demmel and K. Stanley. The performance of finding eigenvalues and eigenvectors of dense symmetric matrices on distributed memory computers. In *Proc. seventh SIAM conf. on parallel processing for scientific computing*, pages 528–533, 1995.

[85] R. C. Ward, Y. Bai, and J Pratt. Performance of parallel eigensolvers on electronic structure calculations II. Technical report, The University of Tennessee, 2006.

[86] Leibniz rechenzentrum. `http://www.lrz.de/`.

[87] T. Auckenthaler, V. Blum, H.-J. Bungartz, T. Huckle, R. Johanni, L. Krämer, B. Lang, H. Lederer, and P. Willems. Parallel solution of partial symmetric eigenvalue problems from electronic structure calculations. *Parallel Computing*, 2011.

[88] T. Auckenthaler, H.-J. Bungartz, T. Huckle, L. Krämer, B. Lang, and P. Willems. Developing algorithms and software for the parallel solution of the symmetric eigenvalue problem. *Journal of Computational Science*, May 2011.

[89] J. Hein. Improved parallel performance of SIESTA for the HPCx Phase2 system. Technical report, The University of Edinburgh, 2004.