Hybrid Parallelization and Performance Optimization of the FLEUR Code: New Possibilities for All-electron Density Functional Theory

Uliana Alekseeva, Gregor Michalicek, Daniel Wortmann, and Stefan Blügel

Institute for Advanced Simulation and Peter Grünberg Institut,
Forschungszentrum Jülich and JARA, 52425 Jülich, Germany
{u.alekseeva,g.michalicek,d.wortmanm,s.bluegel}@fz-juelich.de
http://www.fz-juelich.de/pgi

Abstract. A hybrid MPI+OpenMP parallelization strategy has been implemented into the density functional theory code FLEUR. Based on the full-potential linearized augmented plane-wave (FLAPW) method, FLEUR is a well-established all-electron code specialized on the simulation of materials properties of crystalline bulk solids and surfaces with significant electronic and magnetic complexity. Developed in over 30 years the Fortran implementation included two layers of MPI-based distributed memory parallelization that serves as a reference for our work. The revised code version shows superior performance, improved scalability and thereby opens the path to exploit current and future high performance computing architectures efficiently. Multiple threads per MPI process can be utilized by interfacing with optimized linear algebra subroutines from the BLAS and LAPACK libraries as well as in code sections with explicit OpenMP statements. We demonstrate that the additional multithreading helps to avoid the communication induced scalability limit of the pure-MPI version and simultaneously boosts the single node-performance on current multi-core systems. This enables FLEUR calculations for unit cells with over 1000 atoms to simulate extended defects, surfaces and disordered solids.

Keywords: DFT, FLAPW, Hybrid parallelization

1 Introduction

Over the last decades density functional theory (DFT) calculations [10] have become an indispensable tool for the simulation of material properties and the prediction of new materials showing novel functionality. The increasing computational resources together with algorithmic advances and methodological developments make the calculation of more and more properties for more and more complex materials feasible. Due to the large variety of properties, physical effects and the difference in the computational challenges that arise, many established DFT-codes have been developed [4] that typically implement different algorithms.

The increase in computational resources, however, also comes with a change of the hardware architectures. Decades ago a typical mainframe computer featured a small number of computational cores and parallelism utilized few of these single-core nodes with distributed memory. Nowadays HPC machines typically are cluster systems consisting of many shared memory nodes connected through a communication network and featuring several multi-core CPUs each. The additional parallelization layers in such architectures together with the larger but also shared memory capacity on each node entail the requirement to adapt the software parallelization strategies.

We perform this adaption for the FLEUR [2] code developed at the Research Center Jülich. This is a full-potential linearized augmented-plane-wave (FLAPW) [5, 11, 19, 18] implementation of DFT. Being an all-electron code FLEUR is employable to perform highly precise simulations for solids, surfaces and molecular systems consisting of arbitrary compositions of chemical elements and it has its particular strength in the simulation of magnetism and relativistic effects.

To utilize modern hierarchical architectures efficiently, a "hierarchical", hybrid parallelization is implemented, i.e. the distributed memory paradigm (MPI) and a multi-threaded shared memory paradigm are combined. The aim of the new hybrid parallelization scheme presented here is not only to make the intra-node CPU usage effective, but also to enable simulations of big unit cells using many nodes. To achieve this, the "top-down" approach [17] was applied, i.e. for a given test case, first the efficiency of MPI parallelization was investigated and improved when needed, then multi-threading was added, either as calls to external multi-threaded libraries or as direct implementation of OpenMP pragmas. We show that we obtained significant performance and scalability enhancements pushing the limit of applicability of the code to simulations with over 1000 atoms.

While we implemented many improvements throughout the code, we will concentrate the discussion on the setup of the matrices and the subsequent matrix diagonalization. The latter part can efficiently be solved using standard libraries for dense generalized eigenvalue problems. The first of these two most time consuming parts of the code other authors also discussed before in detail [16, 14]. While we agree with those works in the aspect of stressing the benefit of using standard matrix operations, we use a significantly different algorithm exploiting in addition the analytic properties of the problem and thereby reducing the number of computations needed. We tested our approach against simpler schemes and found it to show superior performance and scaling.

In the next chapter we introduce the FLAPW algorithm. In Chapter 3 we discuss the parallelization and optimization performed to achieve the benchmark results presented in Chapter 4, Chapter 5 concludes the paper.

2 Density Functional Theory and the FLAPW Method

According to density functional theory [9, 12], the total energy of a system of interacting atoms and electrons is a functional of its electron density $n(\boldsymbol{r})$. Hence, the Hamiltonian (the energy operator) of the system

$$\hat{H}[n(\boldsymbol{r})] = \hat{T} + V_{\text{eff}}[n(\boldsymbol{r})], \tag{1}$$

which is the sum of the kinetic energy operator $\hat{T}$ and the effective potential V_{eff}, depends directly on the electron density. The electron density can be expressed in terms of N_{occ} occupied single-particle orbitals $\psi_\nu(\boldsymbol{r})$:

$$n(\boldsymbol{r}) = \sum_{\nu}^{N_{\text{occ}}} |\psi_\nu(\boldsymbol{r})|^2 \ , \tag{2}$$

where ν labels the states. The single-particle orbitals $\psi_\nu(\boldsymbol{r})$ are the solutions of the Kohn-Sham equations, an eigenvalue problem with eigenvalues ϵ_ν:

$$\hat{H}[n(\boldsymbol{r})]\psi_\nu(\boldsymbol{r}) = \epsilon_\nu \psi_\nu(\boldsymbol{r}) \ . \tag{3}$$

Since the Hamiltonian in the equation depends on its solution, this is a self-consistency problem which has to be solved iteratively: starting with an initial guess the ground-state density is therefore obtained in an iterative scheme that produces a new density in each iteration. The new input density is obtained by a mixing procedure from the old input density, the output density, and optionally further densities related to earlier iterations of this self-consistency cycle. The final ground-state density is self-consistent with respect to this procedure.

We solve the Kohn-Sham equations for crystalline solids described by a unit cell with a finite number of atoms, which is repeated indefinitely in all three spatial dimensions to fill up the whole space. For such solids the Hamiltonian matrix can be block-diagonalized and each block provides an independent eigenvalue problem. Each block is indexed by the so called Bloch vector $\boldsymbol{k}$, hence in the following the matrices, their eigenvalues and eigenvectors feature an extra $\boldsymbol{k}$-index [7].

2.1 FLAPW Method

A common approach to solving Eq. (3) is to expand the wave functions in terms of a set of basis functions $\{\phi_{\boldsymbol{k}}^{\boldsymbol{G}}\}$ as

$$\psi_{\nu,\boldsymbol{k}}(\boldsymbol{r}) = \sum_{\boldsymbol{G}} c_{\nu,\boldsymbol{k}}^{\boldsymbol{G}} \phi_{\boldsymbol{k}}^{\boldsymbol{G}}(\boldsymbol{r}) \ . \tag{4}$$

By this the Hamiltonian becomes a Hermitian matrix and the eigenvalue problem is solved by a matrix diagonalization. Equation (3) becomes the generalized eigenvalue problem

$$\sum_{\boldsymbol{G}'} H_{\boldsymbol{G},\boldsymbol{G}'}^{\boldsymbol{k}} c_{\nu,\boldsymbol{k}}^{\boldsymbol{G}'} = \epsilon_{\nu,\boldsymbol{k}} \sum_{\boldsymbol{G}'} S_{\boldsymbol{G},\boldsymbol{G}'}^{\boldsymbol{k}} c_{\nu,\boldsymbol{k}}^{\boldsymbol{G}'} \ , \tag{5}$$

where

$$H_{\boldsymbol{G},\boldsymbol{G}'}^{\boldsymbol{k}} = \int (\phi_{\boldsymbol{k}}^{\boldsymbol{G}})^* \hat{H} \phi_{\boldsymbol{k}}^{\boldsymbol{G}'} \, d\boldsymbol{r} \ \text{ and } \ S_{\boldsymbol{G},\boldsymbol{G}'}^{\boldsymbol{k}} = \int (\phi_{\boldsymbol{k}}^{\boldsymbol{G}})^* \phi_{\boldsymbol{k}}^{\boldsymbol{G}'} \, d\boldsymbol{r} \tag{6}$$

are the Hamiltonian matrix and the overlap matrix.

In the all-electron full-potential linearized augmented-plane-wave method (FLAPW) [5, 11, 19, 18] the basis functions are linearized augmented-plane-waves (LAPWs) which are based on a partitioning of space into non-overlapping but nearly touching muffin-tin (MT) spheres centered at each atom and an interstitial region (INT) in between the spheres. Formally a LAPW is given by

$$\phi_{\boldsymbol{k}}^{\boldsymbol{G}}(\boldsymbol{r}) = \begin{cases} \frac{1}{\sqrt{\Omega}} e^{i(\boldsymbol{k}+\boldsymbol{G})\boldsymbol{r}} & \text{in INT} \\ \sum\limits_{\alpha} \sum\limits_{L}^{l_{\max}^{\alpha}} \sum\limits_{p} a_{L,\alpha}^{\boldsymbol{k},\boldsymbol{G},p} u_{l,\alpha}^{p}(r_\alpha) Y_L(\hat{\boldsymbol{r}}_\alpha) & \text{in MT}_\alpha \end{cases} , \tag{7}$$

where $\boldsymbol{G}$ is a reciprocal lattice vector used to index the LAPW, Ω is the volume of the unit cell, and $\boldsymbol{r}_\alpha = \boldsymbol{r} - \boldsymbol{\tau}_\alpha$ is the position vector relative to atom α at $\boldsymbol{\tau}_\alpha$. The MT part of the function is a linear combination of radial functions $u_{l,\alpha}^{p}$ times spherical harmonics Y_L, where $p \in \{0, 1\}$ is an index to select one of the radial functions. The coefficients $a_{L,\alpha}^{\boldsymbol{k},\boldsymbol{G},p}$ are determined by matching the MT part of the LAPW in value and slope to the plane wave in the interstitial region. The set of LAPW basis functions is defined by the reciprocal plane wave cutoff parameter $K_{\max} = |\boldsymbol{K}|_{\max} = |\boldsymbol{k} + \boldsymbol{G}|_{\max}$ and its MT representation is bounded by the angular momentum cutoffs $l_{\max}^{\alpha}$ for the sum over the composite index $L = (l, m)$. Typically one needs about 100 basis functions per atom and an $l_{\max}^{\alpha}$ between 8 and 12 to obtain converged FLAPW results.

Besides the basis functions, the representations of the density and the potential are FLAPW specific and their constructions are important parts of an FLAPW program. However, the runtime of an FLAPW calculation is typically strongly dominated by the setup and solving of the generalized eigenvalue problem. In the following we therefore focus on the computation of the Hamiltonian and overlap matrices.

2.2 Hamiltonian and Overlap Matrices

After integrating (Eq. 6) over the LAPWs (Eq. 7), the Hamiltonian and overlap matrices are given as sums over the MT contributions from each atom and the INT contribution as

$$H_{\boldsymbol{G},\boldsymbol{G}'}^{\boldsymbol{k}} = H_{\boldsymbol{G},\boldsymbol{G}'}^{\boldsymbol{k},\text{INT}} + \sum_{\alpha} H_{\boldsymbol{G},\boldsymbol{G}'}^{\boldsymbol{k},MT_\alpha} = H_{\boldsymbol{G},\boldsymbol{G}'}^{\boldsymbol{k},\text{INT}} + \sum_{\alpha} H_{\boldsymbol{G},\boldsymbol{G}'}^{\boldsymbol{k},\alpha,\text{sph}} + H_{\boldsymbol{G},\boldsymbol{G}'}^{\boldsymbol{k},\alpha,\text{nsph}} \tag{8}$$

and

$$S_{\boldsymbol{G},\boldsymbol{G}'}^{\boldsymbol{k}} = S_{\boldsymbol{G},\boldsymbol{G}'}^{\boldsymbol{k},\text{INT}} + \sum_{\alpha} S_{\boldsymbol{G},\boldsymbol{G}'}^{\boldsymbol{k},\alpha} , \tag{9}$$

where we also distinguish for each MT sphere between the spherical contributions to the Hamiltonian matrix $H_{\boldsymbol{G},\boldsymbol{G}'}^{\boldsymbol{k},\alpha,\text{sph}}$ and those due to the non-spherical part of the potential $H_{\boldsymbol{G},\boldsymbol{G}'}^{\boldsymbol{k},\alpha,\text{nsph}}$.

Since an interstitial LAPW is a plane wave the calculation of the related matrix contributions is fast. Its time requirements only scale quadratically with the system size. We discuss the more challenging MT setup.

The MT contributions to the Hamiltonian are given as

$$H_{\boldsymbol{G},\boldsymbol{G}'}^{\boldsymbol{k},\alpha} = \sum_{L,L'} \sum_{p,p'} \left(a_{L,\alpha}^{\boldsymbol{k},\boldsymbol{G},p}\right)^* t_{L,L'}^{\alpha,p,p'} a_{L',\alpha}^{\boldsymbol{k},\boldsymbol{G}',p'} \tag{10}$$

in which $t_{L,L'}^{\alpha,p,p'}$ denotes the local Hamiltonian matrix for the respective atom in the basis of the radial functions times spherical harmonics. The calculation of $H_{\boldsymbol{G},\boldsymbol{G}'}^{\boldsymbol{k},\alpha}$ is computationally expensive and in comparision to a simple implementation we use several measures to reduce these computational demands.

The first of these makes use of analytical calculations that can be performed for the spherical contributions, by making use of the addition theorem for spherical harmonics [6]. One obtains

$$\begin{aligned} H_{\boldsymbol{G},\boldsymbol{G}'}^{\boldsymbol{k},\alpha,\mathrm{sph}} &= \sum_{L} \sum_{\sigma} \left(a_{L,\alpha}^{\boldsymbol{k},\boldsymbol{G},\sigma}\right)^* \sum_{\sigma'} t_{L,L}^{\alpha,\sigma,\sigma'} a_{L,\alpha}^{\boldsymbol{k},\boldsymbol{G}',\sigma'} \\ &= \sum_{l=0}^{l_{\mathrm{max}}^{\alpha}} \frac{2l+1}{4\pi} P_l \left(\frac{\boldsymbol{K}\boldsymbol{K}'}{|\boldsymbol{K}\boldsymbol{K}'|}\right) \left[\sum_{\sigma} \left(a_{l,\alpha}^{\boldsymbol{k},\boldsymbol{G},\sigma}\right)^* \sum_{\sigma'} t_{l,l}^{\alpha,\sigma,\sigma'} a_{l,\alpha}^{\boldsymbol{k},\boldsymbol{G}',\sigma'}\right] \end{aligned} \tag{11}$$

in which P_l denotes the Legendre polynomial of degree l. An analogous expression is obtained for the MT contributions to the overlap matrix $S_{\boldsymbol{G},\boldsymbol{G}'}^{\boldsymbol{k},\alpha}$ which is computed as a byproduct. Note that the analytic m summation reduces the computational demands for these matrix elements by a factor of about 10.

The remaining Hamiltonian matrix contributions due to the non-spherical part of the potential are

$$H_{\boldsymbol{G},\boldsymbol{G}'}^{\boldsymbol{k},\alpha,\mathrm{nsph}} = \sum_{L} \sum_{\sigma} \left(a_{L,\alpha}^{\boldsymbol{k},\boldsymbol{G},\sigma}\right)^* \left(\sum_{L' \neq L} \sum_{\sigma'} t_{L,L'}^{\alpha,\sigma,\sigma'} a_{L',\alpha}^{\boldsymbol{k},\boldsymbol{G}',\sigma'}\right) . \tag{12}$$

The last measure to reduce the required computational effort is based on the realization that in comparison to Eq. (11), Eq. (12) has lower demands with respect to the cutoff of the L sums. Therefore in practice one uses a new cutoff $l_{\mathrm{nsph}}^{\alpha} \approx \min(8, l_{\mathrm{max}}^{\alpha} - 2)$ for the L and L' sums in this equation. This provides another reduction of the time requirements for these calculations by 30 to 50%. However, calculating the non-spherical contributions remains the most time-consuming step in the setup of the matrices.

2.3 Scaling and Time Requirements

Of course, the computational demands of the different steps of an FLAPW calculation feature different scaling behaviors with respect to the system size defined by the number of atoms N_{at}. Tab. 1 shows these different behaviors depending on N_{at} but also more explicitely on the number of LAPW basis functions $N_{\boldsymbol{G}}$, the angular momentum cutoff $l_{\mathrm{max}}^{\alpha}$, the number of $\boldsymbol{k}$-points $N_{\boldsymbol{k}}$, and the number of occupied eigenstates N_{occ} (see sum over ν in Eq. (2)).

Table 1: Scaling of the most time-consuming parts of an FLAPW self-consistency iteration

Computational task	Scaling vs. numerical parameters	Scaling vs. system size
Potential generation	$\mathcal{O}\left(\sum_\alpha (l^\alpha_{\max}+1)^2 N_{\boldsymbol{G}} + N_{\boldsymbol{G}}\log(N_{\boldsymbol{G}})\right)$	$\mathcal{O}\left(N_{\text{at}}^2\right)$
Matrix setup	$\mathcal{O}\left(N_{\boldsymbol{k}} \sum_\alpha (l^\alpha_{\max}+1)^2 {N_{\boldsymbol{G}}}^2\right)$	$\mathcal{O}\left(N_{\text{at}}^3\right)$
Diagonalization	$\mathcal{O}\left(N_{\boldsymbol{k}} {N_{\boldsymbol{G}}}^3\right)$	$\mathcal{O}\left(N_{\text{at}}^3\right)$
Charge density generation	$\mathcal{O}\left(N_{\boldsymbol{k}} \sum_\alpha (l^\alpha_{\max}+1)^2 N_{\boldsymbol{G}} N_{\text{occ}}\right)$	$\mathcal{O}\left(N_{\text{at}}^3\right)$

All of these parameters are system-dependent but only $N_{\boldsymbol{G}}$ and N_{occ} are proportional to the number of atoms, while $l^\alpha_{\max}$ is independent of N_{at} and $N_{\boldsymbol{k}}$ is reciprocal to N_{at} in each direction but at least 1. Overall this implies a cubical scaling of the time requirements with respect to the number of atoms.

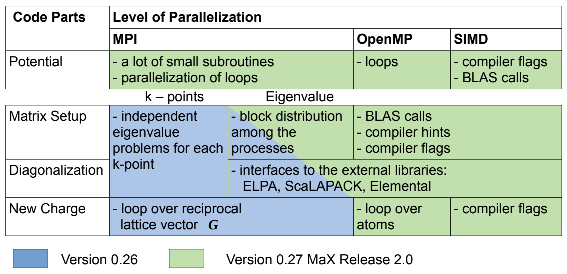
Typical time requirements for the different steps in a single iteration of the self-consistency loop are shown in Tab. 2. The run time dominance of the matrix setup and the diagonalization step are clearly visible for all problem sizes. For larger numbers of atoms this dominance becomes even more pronounced.

Table 2: Run time measurements of the FLEUR code (MaX Release 2.0) for three test unit cells: NaCl (64 atoms), AuAg (108 atoms) and CuAg(256 atoms). All simulations are performed on the CLAIX computing cluster with one $\boldsymbol{k}$-point, for one self-consistency iteration. The measurements are provided in seconds (left side) as well as relative percentage values (right side).

Test system	NaCl		AuAg		CuAg	
Number of atoms	64		108		256	
Potential generation	3.5	12.5 %	12.4	3.9 %	47.2	4.8 %
Matrix setup	8.1	29.0 %	127.7	40.4 %	455.2	46.3 %
Diagonalization	10.6	38.2 %	145.5	46.0 %	384.2	39.1 %
New charge density generation	2.9	10.4 %	22.5	7.1 %	78.6	8.0 %
Total time	27.8	100 %	316.3	100 %	982.4	100 %

3 Parallelization and Optimization

Since different parts of the code have different algorithms and scaling behaviour, there is no single parallelization strategy which is applicable to the whole code. Fig. 1(left) summarizes how the computational load is distributed for each section of the code on every parallelization level. There are two layers of MPI parallelization for the most time-consuming parts, matrix setup, the diagonalization and for the new charge density generation part. To make the code suitable for modern HPC architectures with their hierarchical structure of parallelism, it has been extended with multi-threading and SIMD parallelization schemes.



Code Parts	Level of Parallelization			
	MPI		OpenMP	SIMD
Potential	- a lot of small subroutines - parallelization of loops		- loops	- compiler flags - BLAS calls
	k – points	Eigenvalue		
Matrix Setup	- independent eigenvalue problems for each k-point	- block distribution among the processes	- BLAS calls - compiler hints - compiler flags	
Diagonalization		- interfaces to the external libraries: ELPA, ScaLAPACK, Elemental		
New Charge	- loop over reciprocal lattice vector $\boldsymbol{G}$		- loop over atoms	- compiler flags

🟦 Version 0.26 🟩 Version 0.27 MaX Release 2.0

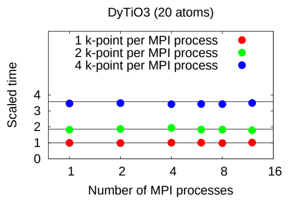


Fig. 1: *Left Side:* The schematic summary of parallelization strategies used for different parts of the code. *Right Side:* Week scaling over $\boldsymbol{k}$-points for test unit cell DyTiO3. The number of $\boldsymbol{k}$-points is proportional to the number of MPI processes. The red points show the run times for calculations with 1,2,4,6,8, and 12 $\boldsymbol{k}$-points distributed over 1,2,4,6,8, and 12 MPI processes correspondingly. The green and blue points show the run times for test cases with 2 and 4 $\boldsymbol{k}$-points per MPI process. Run time is scaled to the run time of the test case with 1 $\boldsymbol{k}$-point on 1 node (94 seconds for one self-consistency iteration). The horizontal lines are theoretical predictions. Simulations are done on the RWTH Bull Cluster, one MPI process per node.

3.1 MPI Parallelization

The MPI parallelization relies on two levels of parallelism. On the first level, the different k-points for which the Kohn-Sham Eqs. (3) have to be solved are distributed. As these are independent problems only the final results of the diagonalization has to be communicated and hence this parallelization is extremely efficient with nearly ideal scaling. Fig. 1(right) demonstrates and confirms this perfect weak scaling. While this level of parallelization is very efficient in terms of distributing the computational load, it has two shortcomings. First, in large systems the number of k-points to be considered is small and hence this parallelization is very limited. Second, as the diagonalization part of the code corresponds to peak memory usage, the k-point parallelization does not reduce memory requirements per node.

The second level of MPI parallelization implements the distribution of the matrices and hence additionally distributes the computation of the matrix setup, the diagonalization and some critical parts of the charge generation routines. We will discuss details of the distributed matrix setup of the new version in the next section. The distributed memory parallelization was very performant at the time of its implementation [8], it worked excellent for machines like the CRAY T3E (512 CPUs).

The new code version (FLEUR version 0.27 MaX Release 2.0) reported in this work extends the existing MPI parallelization into further code parts and hence pushes the scalability limit as set by Amdahl's law. The old optimization (FLEUR version 0.26) for a small memory footprint also affected the quadratically scaling storage of the eigenvectors and the linearly scaling storage of the potential and the density. To reduce the memory consumption these were sequentially written to Fortran direct access files on disc whenever they were not needed and later read from disc again. However for large scale parallelization this approach becomes a bottleneck that was overcome by additional alternative storage schemes for the eigenvectors. On the one hand it is now possible to keep them entirely in working memory and communicate them by one-sided MPI communication and on the other hand if memory consumption still is a problem they can be stored on disc in terms of HDF5 files with parallel IO. The potential and density are now always kept in memory and communicated via MPI broadcasts. Overall the reduction of disc IO measurably increases the parallelization scalability.

3.2 Hybrid Parallelization and Optimized Matrix Setup

One of the main optimization targets was the matrix setup. In the old version, it was heavily optimized to reduce memory footprint. For example, several matrix-matrix multiplications were unrolled to enable the calculation of matrix elements on the fly without storing the whole matrix. In all of the matrix setup routines the second level of MPI parallelization utilizes a cyclic row distribution [1] of the matrices. This ensures good load-balancing and effective re-use of calculated quantities. The interstitial contribution can be easily calculated, does not take much time and allows for a straightforward MPI and an additional OpenMP parallelization over the matrix rows. It scales almost perfectly due to the independence of the computations and the absence of communication.

As discussed above, the matrix setup in the MT spheres is the most computationally relevant part of the matrix setup. In the old (version 0.26) implementation of FLEUR, spherical contributions to the H and S matrices and non-spherical contribution to the H matrix were calculated in a single subroutine which contained more than 1500 lines of code. This coarse-grained modularity of the code is beneficial if the heavy reduction of the memory footprint is aspired. Nowadays modularity in routines in which the main computational effort is performed by the lowest kernels is more advantageous. It is less error-prone and improves readability and maintainability of the code. Besides that, in case these low kernels perform some common mathematical operation such as linear algebra operations or Fourier transforms, external libraries can be used which are usually highly optimized for a given hardware. Hence, the first step was to increase the modularity of the code. The huge initial subroutine was split to several smaller ones.

The most important code split reflected the separation of the spherical and non-spherical contributions. In the routines for the spherical MT contribution the parallelization over the basis vectors on the MPI-level shows close to ideal scaling. To further distribute the computations in this code section, a layer of OpenMP parallelization over the atoms of the system has been added.

The non-spherical contributions to the Hamiltonian are now calculated by first explicitly constructing the matrices $A_\alpha^{\boldsymbol{k}} = [a_{L,\alpha}^{\boldsymbol{k},\boldsymbol{G},\sigma}]$ and $T_\alpha = [t_{L,L'}^{\alpha,\sigma,\sigma'}]$ such that the sums over L, L', σ, σ' can now be performed as matrix multiplications. Hence the algorithm in this part basically consists of the construction of the A-matrices, a first matrix-matrix multiplication

$$C_\alpha^{\boldsymbol{k}} = T_\alpha * A_\alpha^{\boldsymbol{k}} \tag{13}$$

and a second multiplication

$$H_\alpha^{\boldsymbol{k},\mathrm{nsph}} = \left(A_\alpha^{\boldsymbol{k}}\right)^H * C_\alpha^{\boldsymbol{k}} \ . \tag{14}$$

These two different matrix computations scale significantly different with system size. The first is an $\mathcal{O}\left((l_{\max}^\alpha + 1)^4 N_{\boldsymbol{G}}\right)$ operation, the second scales as $\mathcal{O}\left((l_{\max}^\alpha + 1)^2 {N_{\boldsymbol{G}}}^2\right)$ and hence is most relevant for large systems. As the first of these matrix multiplications has to be performed on all MPI-ranks, it is simply mapped onto a standard matrix-matrix multiplication that enables us to exploit highly optimized BLAS-3 libraries for this operation.

For the second matrix multiplication, the MPI-distribution over rows and the property of the Hamiltonian should be considered. The algorithm we implemented here is a trade-off of the two contradictory conditions. On the one hand, it is determined by the fact that the final matrix is Hermitian and only one half of it has to be calculated and stored. On the other hand, we wish to use again optimized, vendor supplied BLAS3 (matrix-matrix multiply) routines to increase the efficiency. $H_\alpha^{\boldsymbol{k},\mathrm{nsph}}$ is distributed between MPI processes in cyclic row distribution: if there are M processes, the line i of the matrix can be found on the process with the number $mod(i, M)$. That means, each MPI process possess data from a rectangular matrix with size $(N_G/M) \times N_G$. Note that line i only has i elements. The matrix is stored as a packed storage vector. To be able to use BLAS3 routines, the matrix $H_\alpha^{\boldsymbol{k},\mathrm{nsph}}$ is divided into blocks (Fig. 2). Each block is calculated as matrix-matrix multiplication, then the values from the block are copied to the packed storage vector. Here we had to find a trade-off between a small block-size that exploits the fact that the final result is Hermitian most effectively, and a larger block-size that leads to better performance of the matrix-matrix multiplication. We found a value of about 64-128 most suitable on the machines we considered. As a final point we should stress, that our scheme has the important advantage that all operations performed in the matrix setup are local for each MPI-process. No communication is required as the MPI-distributed matrix elements are obtained independently for each process.

Besides the matrix-setup the second time consuming part is the diagonalization of the matrices. Here we rely on standard libraries. The old code imple-

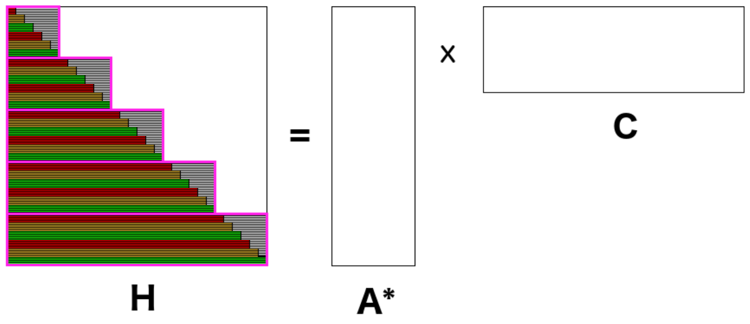


Fig. 2: Example of parallel data layout distributed between 3 MPI processes (red, yellow and green). Matrix $\mathbf{H}$ is distributed among MPI processes in line-wise fashion, so that each MPI process has data from a rectangular matrix with size $(N_G/M) \times N_G$. To be able to use BLAS3 routines, the matrix $H_{\alpha}^{\boldsymbol{k},\mathrm{nsph}}$ is divided into blocks (pink). Each block is calculated as matrix-matrix multiplication, then the values from the block are copied to the packed storage vector.

mented an interface to the ScaLAPACK [1] for this purpose. To obtain reasonable performance this requires a redistribution of the matrix from the simple row cyclic scheme used in the matrix setup to a two-dimensional block-cyclic scheme. While this imposes a communication overhead in theory, such a redistribution turns out to be fast enough that it does not impose a relevant restriction in practice. We furthermore implemented additional interfaces to external hybrid-parallel libraries (ELPA [13], Elemental [15]). It turns out that the ELPA library outperforms ScaLAPACK significantly and also has the additional benefit of delivering much more consistent performance for different levels of MPI and OpenMP parallelism resulting in different processor grids.

With substantial parallelization, also other parts of the code start to play substantial roles: for example, the potential generation could not be left sequential any more. In the other parts of the code either the usage of multi-threaded libraries or the explicit implementation of OpenMP pragmas provided the needed scaling on top of the existing MPI parallelization.

4 Benchmarks

We demonstrate the performance and scalability of the code by showing some exemplary cases. As we have already shown that the additional $\boldsymbol{k}$-point parallelization leads to ideal scaling behaviour we restrict the presentation to calculations using a single $\boldsymbol{k}$-point, in realistic simulations one would have an additional parallelization allowing to use a factor $3 - 20$ (depending on system size) more computational cores effectively. In addition we only consider a single iteration. As the code usually has to perform approximately 20-50 iterations sequentially, the total runtime would increase accordingly.

4.1 Computational Environment

We have parallelized and optimized the FLEUR code for typical architectures found in HPC today: compute clusters with several levels of parallelism: inter-node with distributed memory, intra-node with shared memory and SIMD inside the core. The concrete specifications of the compute clusters used for the performance evaluations in this work are given in Tab. 3.

Table 3: Hardware systems used to perform the benchmark calculations.

	CPU	cores per node	node performance	memory	mem. bandwidth
RWTH Bull Cluster	Intel X5675	12	147 GFlops	24 GB	40 GB/s
CLAIX	Intel E5-2650v4	24	840 GFlops	128 GB	120 GB/s

4.2 Efficient Usage of a Single Node

To investigate the behaviour of FLEUR on a single node we use a small test case: NaCl with 64 atoms. The intranode scaling of the whole code and its main parts are shown in Fig. 3. Only parts of the code whose running time is more than 1% of the total time are considered. We see that the most time-consuming parts are the matrix setup and the diagonalization. The potential generation and the new charge generation do not contribute much to the run time on one core, but as we try to distribute the workload among all cores on this node, their negative influence on the overall efficiency becomes more important.

Most significant in these plots is the limited scalability of the matrix setup in the old, MPI only version. Here we can see that the MPI parallelization shows scalability limits as soon as the workload per MPI process becomes too small. This is not a communication based bottleneck as the matrix setup is local, but a limitation induced by the underlying algorithm with its complex loop structure being heavy on memory access tasks. The new version shows significant improvements not only on the scaling but also on the sequential run-time. This leads to a difference in wall-clock time for the utilization of a full node between the old version requiring 198 seconds versus 97 seconds for the new version. Tests on Intel Knights Landing processors (Xeon Phi 7210) showed comparable results indicating performance portability of the new implementation.

4.3 Internode Hybrid Scaling

To investigate the full scaling of the hybrid code we studied two setups (Fig. 4): A smaller system on the RWTH Bull Cluster and a larger system on the more modern CLAIX machine. Here we not only compared to the pure MPI parallelization of the older code version but we also studied the effect of shifting resources between the MPI- and the multithreaded parallelization. In all cases we utilized

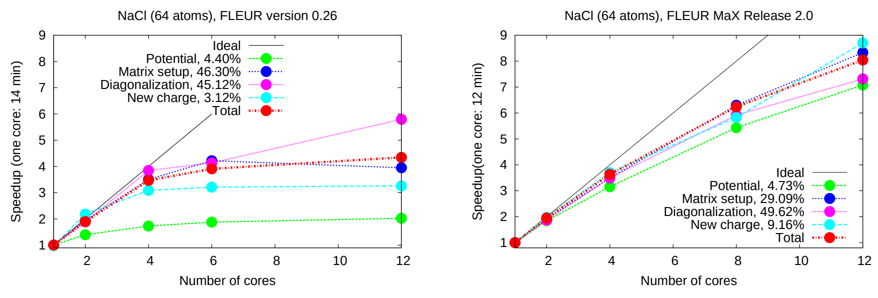


Fig. 3: Intranode scaling of the FLEUR code in total (red) as well as of its most relevant parts (time requirements are given as percentage of the total runtime for a single core (14min/12min)), before (MPI-only version 0.26, left side) and after (hybrid version MaX Release 2.0, right side) optimization. For the optimized version: up to 4 cores - only MPI processes, on 8 und 12 cores - hybrid parallelization. The simulations were performed on the RWTH Bull Cluster.

all cores of the node, but with a different number of MPI ranks per node and a resulting change in mutlithreading. Both systems on both machines show similar behavior. While the pure MPI parallelization is still efficient for small numbers of MPI-ranks, it becomes less favorable with increasing parallelization. Notably, in an intermediate range of parallelization there is little difference between the two approaches demonstrating that both implementations have a similar efficiency.

As a final test we also show (Fig. 5) that the new version of the code enables the simulation of significantly larger setups utilizing stronger parallelization. Here we stress that the hybrid approach also is required as a pure MPI parallelization over 24 ranks per node will fail for larger setups due to memory constraints.

5 Conclusions

We demonstrated that the hybrid MPI+OpenMP parallelization of the large legacy DFT code FLEUR enables the efficient use of modern computer architectures to perform simulations of large unit-cells. The two most performance relevant parts, the matrix setup and the matrix diagonalization show improved scaling and performance by implementing interfaces to optimized standard libraries and by implementing an additional layer of OpenMP parallelization on top of the MPI parallelization. The possibility to shift computational resources between these different parallelization approaches not only shows the effectiveness of the hybrid scheme but also enables the adaptation to different hardware.

The new FLEUR version is able to treat setups with more than 1000 atoms. While this limit imposes an important milestone in itself, this also paves the

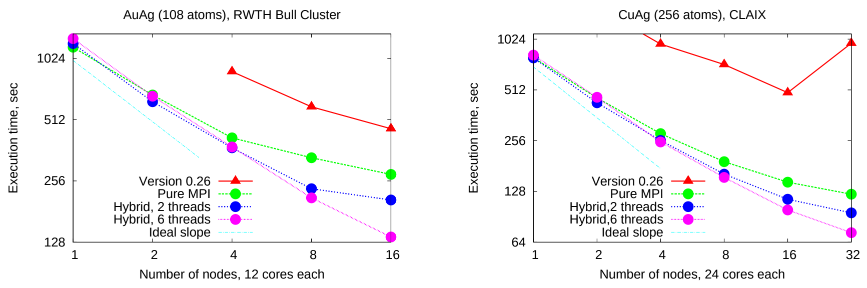


Fig. 4: Internode scaling of the FLEUR code, before (MPI-only version 0.26) and after (hybrid version MaX Release 2.0) optimization. For the hybrid version different hybrid setups are shown: pure MPI, i.e. 1 thread per MPI process (green), 2 threads per MPI process (blue) and 6 threads per MPI process (magenta).

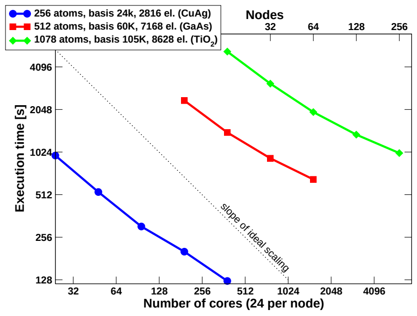


Fig. 5: Comparison of the scaling of the FLEUR code for three systems with different number of atoms, basis functions and electrons. The smallest system is the one discussed in Fig. 4, the largest system contains more than 1000 atoms. Due to the higher computational demand, the scaling for the larger systems extends to more nodes. (MaX Release 2.0, CLAIX compute cluster)

way for the investigation of effects in heterogeneous multilayer structures, reconstructed surfaces, adsorbates on surfaces, defects and extended defects in solids, complex magnetic superstructures or simply large bulk superstructures.

6 Acknowledgments

This work has been supported by a JARA-HPC seed-fund project and by the MaX Center of Excellence [3] funded by the EU through the H2020-EINFRA-

2015-1 project: GA 676598. The authors gratefully acknowledge the computing time granted by the JARA-HPC Vergabegremium on the RWTH supercomputer.

References

1. ScaLAPACK users' guide (1997), http://www.netlib.org/scalapack/slug
2. The Jülich FLEUR project (2018), http://www.flapw.de
3. MaX Centre of Excellence - MATERIALS DESIGN AT THE EXASCALE (2018), http://www.max-centre.eu
4. Psi-k: software codes (2018), http://psi-k.net/software
5. Andersen, O.K.: Linear methods in band theory. Phys. Rev. B **12**, 3060–3083 (1975)
6. Arfken, G.: The Addition Theorem for Spherical Harmonics, pp. 693–695. Academic Press, Orlando (1985)
7. Ashcroft, N.W., Mermin, N.D.: Solid State Physics. Holt, Rinehart and Winston, New-York (1976)
8. Blügel, S., Bihlmayer, G.: Full-potential linearized augmented planewave metho. In: J. Grotendorst, S. Bluğel, D.M. (ed.) Computational Nanoscience: Do It Yourself! NIC Series. vol. 31, pp. 85–129. John von Neumann Institute for Computing, Jülich (2006)
9. Hohenberg, P., Kohn, W.: Inhomogeneous electron gas. Phys. Rev. **136**, B864–B871 (1964)
10. Jones, R.O.: Density functional theory: Its origins, rise to prominence, and future. Rev. of Mod. Phys. **8**, 897–923 (2015)
11. Koelling, D.D., Arbman, G.O.: Use of energy derivative of the radial solution in an augmented plane wave method: application to copper. J. Phys. F: Metal Phys. **5**, 2041–2054 (1975)
12. Kohn, W., Sham, L.: Self-consistent equations including exchange and correlation effects. Phys. Rev. **140**, A1133–1138 (1965)
13. Marek, A., et al.: The elpa library: scalable parallel eigenvalue solutions for electronic structure theory and computational science. J. of Phys.: Condensed Matter **26**, 213201 (2014)
14. Napoli, E.D., et al.: High-performance generation of the hamiltonian and overlap matrices in flapw methods. Comput. Phys. Commun. **211**, 61–72 (2017)
15. Poulson, J., et al.: Elemental: A new framework for distributed memory dense matrix computations. ACM Trans. Math. Soft. **39**, 1–24 (2013)
16. Solca, R., et al.: Efficient implementation of quantum materials simulations on distributed cpu-gpu systems. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 1–12. Texas (2015)
17. Supalov, A., Semin, A., Klemm, M., Dahnken, C.: Optimizing HPC Applications with Intel Cluster Tools. Apress Media (2014)
18. Weinert, M., Wimmer, E., Freeman, A.: Total-energy all-electron density functional method for bulk solids and surfaces. Phys. Rev. B **26**, 4571–4578 (1982)
19. Wimmer, E., Krakauer, H., Weinert, M., Freeman, A.J.: Full-potential self-consistent linearized-augmented-plane-wave method for calculating the electronic-structure of molecules and surfaces - o2 molecule. Physical Review B **24**, 864–875 (1981)