Efficient implementation of quantum materials simulations on distributed CPU-GPU systems

Raffaele Solcà
Institute for Theoretical Physics, ETH
Zurich, Switzerland
rasolca@itp.phys.ethz.ch

Anton Kozhevnikov
Swiss National Supercomputer Center, CSCS
Lugano, Switzerland
kozhevnikov@cscs.ch

Azzam Haidar
Computer Science Dept
University of Tennessee
Knoxville, USA
haidar@eecs.utk.edu

Stanimire Tomov
Computer Science Dept
University of Tennessee
Knoxville, USA
tomov@eecs.utk.edu

Thomas C. Schulthess
Institute for Theoretical Physics, ETH
Swiss National Supercomputer Center
Zurich, Switzerland
Oak Ridge Nat. Lab., USA
schulthess@cscs.ch

Jack Dongarra
University of Tennessee, USA
Oak Ridge Nat. Lab., USA
University of Manchester, UK
dongarra@eecs.utk.edu

ABSTRACT

We present a scalable implementation of the Linearized Augmented Plane Wave method for distributed memory systems, which relies on an efficient distributed, block-cyclic setup of the Hamiltonian and overlap matrices and allows us to turn around highly accurate 1000+ atom all-electron quantum materials simulations on clusters with a few hundred nodes. The implementation runs efficiently on standard multi-core CPU nodes, as well as hybrid CPU-GPU nodes. The key for the latter is a novel algorithm to solve the generalized eigenvalue problem for dense, complex Hermitian matrices on distributed hybrid CPU-GPU systems. Performance tests for Li-intercalated CoO_2 supercells containing 1501 atoms demonstrate that high-accuracy, transferable quantum simulations can now be used in throughput materials search problems. While our application can benefit and get scalable performance through CPU-only libraries like ScaLAPACK or ELPA2, our new hybrid solver enables the efficient use of GPUs and shows that a hybrid CPU-GPU architecture scales to a desired performance using substantially fewer cluster nodes, and notably, is considerably more energy efficient than the traditional multi-core CPU only systems for such complex applications.

1. INTRODUCTION

Quantum simulations have reached a level of maturity and predictiveness that makes them useful as inexpensive screening tools in a materials design process [6, 35]. In such a computation, ab initio electronic structure methods are applied to thousands or tens of thousands of inorganic compounds, in order to compute materials properties targeted by the design process. The present approach is to run the ab initio code on all known inorganic materials compounds and store away computed properties in a database for later use in screening.

A much more flexible approach would be to use a materials database in conjunction with a powerful suite of first principles electronic structure codes that can be run dynamically during the screening process in order to compute, on the fly, observable quantities that are specific to the design goals of a particular project. Such an idea would have been unrealistic a few years ago. But with continued exponential performance improvements of supercomputers and computational methods, this will be feasible in the near future as computers reach exascale performance. Therefore, it is worth preparing implementations of electronic structure methods for such a proposition of materials design.

The more accurate and canonically applicable an electronic structure method is across many different classes of materials, the more appropriate it will be for materials design. Evidently, since the computation must be repeated thousands or tens of thousands of times, the methods must be efficient and individual calculations must be executed on the smallest possible resource in a reasonable amount of time. Furthermore, to be a flexible design tool, the methods must be applicable to large enough systems. Due to the nearsightedness property of electronic matter [28, 38], the size of systems treated in accurate electronic structure computation will have to be "only" on the order of 1000 atoms – we will refer to this here as the ∼1000 atom problem.

Modern electronic structure methods rely on the Kohn-Sham approach [29] to density functional theory (DFT) [24], in which the exponential complexity of the many-body Schrödinger equation is reduced to a computational traceable problem with polynomial complexity. They require the solution of the Kohn-Sham equation:

$$\left[\frac{1}{2}\nabla^2 + V_s(\mathbf{r})\right]\psi_i(\mathbf{r}) = \epsilon_i\psi_i(\mathbf{r}), \qquad (1)$$

where the complex valued solutions $\psi_i(\mathbf{r})$ describe auxiliary single electron orbitals. The effective potential:

$$V_s(\mathbf{r}) = V_{\text{ext}}(\mathbf{r}) + \int \frac{\rho_s(\mathbf{r}')}{|\mathbf{r}-\mathbf{r}'|}d^3\mathbf{r}' + V_{\text{XC}}\left[\rho_s(\mathbf{r})\right](\mathbf{r}), \qquad (2)$$

consists of an external potential V_{ext} that includes the ionic potential of the nuclei, the classical coulomb contribution that is also called the Hartree potential $V_{\text{H}}(\mathbf{r}) = \int \frac{\rho_s(\mathbf{r}')}{|\mathbf{r}-\mathbf{r}'|}d^3\mathbf{r}'$, and the (non-classical) exchange and correlation contributions of all other electrons in the systems V_{XC}. $\rho_s = \sum_{i\in\text{occ.}}|\psi_i|^2$ is the electron density determined from the modulus square of all occupied orbitals. The charge density and the electron potential must be iterated to convergence, thus the Kohn-Sham equations must be solved many times. This is called the self-consistent field (SCF) approach.

Accuracy and computational complexity in quantum simulations is determined primarily by the levels of approximations used for the $V_{\text{ext}} + V_{\text{H}}$ and V_{XC}, respectively. A hierarchy of approximations, called Jacob's ladder [37], exists for the latter. The simplest, and by now most established, approximations to V_{XC} depend on the electron density (Local Density Approximation – LDA) and its gradient (Generalized Gradient Approximation – GGA). More accurate models, such as meta-GGA, hybrid functionals that use Hartree-Fock exchange or self-interaction corrections, and methods based on perturbation theory such as RPA, MP2, and GW, although highly promising, are still in a research state, and their usability across many different compounds is not yet well established. In contrast, LDA and GGA based simulations can be applied to all known inorganic compounds, their limitations are well understood, and are thus applicable for high-throughput screening of materials.

Reliable solutions to the Kohn-Sham equations within the LDA and GGA class of approximations rely on dense representations of the Kohn-Sham Hamiltonian operator $H_s = -\frac{1}{2}\nabla^2 + V_s$ that turns the solution of the Kohn-Sham equation into a generalized eigenvalue problem. Approximations to $V_{\text{ext}} + V_{\text{H}}$ are often used to reduce the size of the subspace over which the eigenvalue problem must be solved. Today's most widely used electronic structure packages use element dependent pseudo potentials [43, 10] that adsorb the core electrons, leaving only the valence electrons and a "soft" potential in the Kohn-Sham equations that can be efficiently treated with a simple plane wave basis set. The gain in computational efficiency of such an approximation is tremendous: the relevant subspace that has to be considered in a 1000-atom calculation has dimensions of only 5,000 to 10,000, and a variety of techniques have been developed to solve this problem efficiently.

However, despite a vast amount of effort spent on developing transferable pseudo potentials, their use and validation remains an art that requires experience and a generally accepted standard still has yet to emerge. Thus, all-electron methods that do not approximate $V_{\text{ext}} + V_{\text{H}}$ remain the only alternative for truly canonical simulations. But since these methods operate on a much larger Hilbert space, they are computationally expensive and thus have so far not been used for materials screening.

In the present contribution we give a novel implementation of the Linearized Augmented Plane Wave (LAPW) method [3, 40, 18] for distributed hybrid multi-core architectures. LAPW is an all-electron method that does not approximate $V_{\text{ext}} + V_{\text{H}}$, and provides the most accurate and robust solution to the Kohn-Sham equations that is applicable to all classes of materials known today – it is used as a gold standard against which other electronic structure methods are calibrated [31]. The implementation we present in section 3 allows us to solve the $\sim$1000 atom scale problem with good turnaround time on a few hundred nodes with commodity multicore CPU and hybrid CPU-GPU architectures alike. Unlike previous implementations, it does not rely solely on k-point parallelization, but uses an efficient distributed setup of Hamiltonian and overlap matrices, and relies on effective distributed memory eigensolvers. For standard multicore CPU systems, the implementation uses ScaLAPACK and the ELPA libraries, where the best performance is achieved with the latter. For hybrid CPU-GPU architectures we have implemented a new distributed memory two-stage solver for the generalized eigenvalue problem of dense, complex Hermitian matrices, which we will discuss in section 4. Performance results are given for both, distributed multi-core and hybrid CPU-GPU architectures in section 5.

2. RELATED WORK

The LAPW implementation is available in several packages - WIEN2k (www.wien2k.at), FLEUR (www.flapw.de), Exciting (exciting-code.org), and Elk (elk.sourceforge.net) -- that are typically applied to smaller systems with a few tens of atoms due to their computational cost. At this scale the method can be parallelized over the k-points in the Brillouin zone integration, and the eigenvalue problems at each k-point are solved with a shared memory model. The present work represents a paradigm shift in scale that requires a fundamentally different implementation strategy. At 1000 atoms scale, the Brillouin zone integral is reduced to an individual point and the memory requirements for the eigenvalue problem cannot be economically accommodated on individual nodes – hence our focus is on a distributed memory implementation. The implementation we give here is open source and can be incorporated into the above mentioned packages – work along these lines is already under way to provide support for $\sim$1000 atom scale LAPW calculations in a future release of the Exciting code.

A distributed eigensolver for the type of eigenvalue problems considered here has long been available within the ScaLAPACK package [8]. A more efficient implementation has been given by [5, 34] and relies on a similar two-stage algorithm that we use here. Both ScaLAPACK and ELPA libraries were used for the CPU-only benchmarks of our implementation of the LAPW method on distributed multicore architecture.

An implementation for the eigensolvers used in electronic structure codes on a hybrid CPU-GPU system [23] is available within the MAGMA library. This has been generalized to systems that have multiple GPUs on a node [22] with large shared random access memory. The present 1,000 atom scale computations would require nodes with a terabyte of RAM or more. Hence, the present contribution will complement this previous work, enabling large quantum simulations on a distributed cluster of commodity CPU-GPU nodes with typical memory size of 32GB.

3. LAPW METHOD WITH DISTRIBUTED MEMORY

The gist of the LAPW method consists of basis functions that follow a quasi analytic construction, similar to the plane waves in the pseudopotential methods, and that are efficient in reproducing the strong variations of the wave-functions near the nuclei. This is achieved by partitioning the space into non-overlapping spheres centered around the atoms and realizing the strongly varying potential is nearly spherically symmetric near the origin. Between the spheres the potential varies slowly. Thus, the Kohn-Sham orbitals can be expanded in plane waves in the interstitial regions and in atomic-like functions $u_{\ell\nu}(r)Y_{\ell m}(\hat{\mathbf{r}})$ inside the spheres, where the radial components $u_{\ell\nu}(r)$ are orthogonalized n-th order (with zero-order being a function itself) energy derivatives of the radial Schrödinger equation solutions[3]:

$$u_{\ell\nu}(r) \equiv \frac{\partial^{n_\nu}}{\partial^{n_\nu}E} u_\ell(r,E)\Big|_{E=E_\nu}, \tag{3}$$

$$\int_0^{R_{MT}} u_{\ell\nu}(r)u_{\ell'\nu'}(r)r^2dr = \delta_{\ell\ell'}\delta_{\nu\nu'}. \tag{4}$$

Hence, the LAPW basis functions are given by:

$$\varphi_{\mathbf{G}+\mathbf{k}}(\mathbf{r}) = \begin{cases} \sum\limits_L \sum\limits_{\nu=1}^{O_\ell^\alpha} A^{\mathbf{k}}_{\alpha L\nu}(\mathbf{G})u^\alpha_{\ell\nu}(r)Y_L(\hat{\mathbf{r}}) & \mathbf{r}\in \mathrm{MT}\alpha \\ \frac{1}{\sqrt{\Omega}}e^{i(\mathbf{G}+\mathbf{k})\mathbf{r}} & \mathbf{r}\in \mathrm{I}, \end{cases} \tag{5}$$

where $L \equiv \{\ell, m\}$ denotes the angular momentum and azimuthal quantum numbers and $\sum_L \equiv \sum_{\ell=0}^{\ell_{max}}\sum_{m=-\ell}^{\ell}$. The matching coefficients $A^{\mathbf{k}}_{\alpha L\nu}(\mathbf{G})$ are chosen to ensure continuity of the basis functions (and if possible of their derivatives) on the boundaries of the sphere α. The overlap matrix is given by:

$$\begin{aligned} O^{\mathbf{k}}_{\mathbf{G}\mathbf{G}'} &= \langle\varphi_{\mathbf{G}+\mathbf{k}}|\varphi_{\mathbf{G}'+\mathbf{k}}\rangle \\ &= \sum_{\alpha L\nu} A^{\mathbf{k}*}_{\alpha L\nu}(\mathbf{G})A^{\mathbf{k}}_{\alpha L\nu}(\mathbf{G}') + \Theta(\mathbf{G}-\mathbf{G}'), \end{aligned} \tag{6}$$

where $\Theta(\mathbf{G})$ is a Fourier transform of the unit step function[1].

For an efficient high-performance implementation of these methods, it is important to note that the contribution of the overlap matrix inside the spherical regions is nothing but a multiplication of two matching coefficient arrays with the summation over a composite index $\{\alpha, L, \nu\}$. Similarly, the Hamiltonian matrix can be written in a form that involves matrix-matrix multiplications:

$$\begin{aligned} H^{\mathbf{k}}_{\mathbf{G}\mathbf{G}'} &= \langle\varphi_{\mathbf{G}+\mathbf{k}}|\hat{H}|\varphi_{\mathbf{G}'+\mathbf{k}}\rangle \\ &= \sum_{\alpha L\nu} A^{\mathbf{k}*}_{\alpha L\nu}(\mathbf{G})B^{\mathbf{k}}_{\alpha L\nu}(\mathbf{G}') \\ &+ \frac{1}{2}(\mathbf{G}+\mathbf{k})(\mathbf{G}'+\mathbf{k})\Theta(\mathbf{G}-\mathbf{G}') + \tilde{V}_s(\mathbf{G}-\mathbf{G}'), \end{aligned} \tag{7}$$

where $\tilde{V}_s(\mathbf{G})$ is a Fourier transform of the effective Kohn-Sham potential multiplied by the unit step function, and array $B^{\mathbf{k}}_{\alpha L\nu}(\mathbf{G})$ can be considered as a result of the application of the muffin-tin Hamiltonian $\sum_L h^\alpha_L(r)R_L(\hat{\mathbf{r}})$ to the array of matching coefficients:

$$\begin{aligned} B^{\mathbf{k}}_{\alpha L\nu}(\mathbf{G}) &= \sum_{\substack{L_3\\ L_2\nu_2}} A^{\mathbf{k}}_{\alpha L_2\nu_2}(\mathbf{G})h^{\alpha\ell\nu}_{L_3\ell_2\nu_2}\langle Y_L|R_{L_3}|Y_{L_2}\rangle \\ &+ \frac{1}{2}\sum_{\nu_2} A^{\mathbf{k}}_{\alpha L\nu_2}(\mathbf{G})u^\alpha_{\ell\nu}(R_\alpha)u'^\alpha_{\ell\nu_2}(R_\alpha)R^2_\alpha. \end{aligned} \tag{8}$$

The second part of Eq. (8) is a surface contribution to kinetic energy[2] and

$$h^{\alpha\ell\nu}_{L_3\ell_2\nu_2} = \int_0^{R^\alpha_{MT}} u^\alpha_{\ell\nu}(r)h^\alpha_{L_3}(r)u^\alpha_{\ell_2\nu_2}(r)r^2dr \tag{9}$$

$$\langle Y_L|R_{L_3}|Y_{L_2}\rangle = \iint Y^*_L(\theta,\phi)R_{L_3}(\theta,\phi)Y_{L_2}(\theta,\phi)\sin\theta d\phi d\theta \tag{10}$$

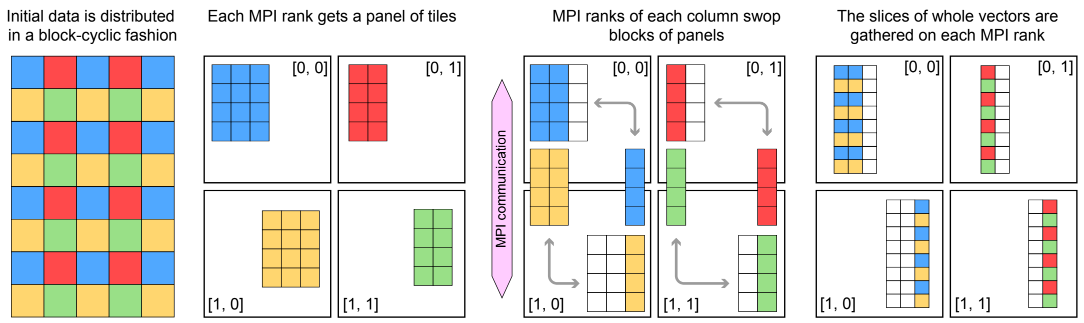
are, respectively, the radial Hamiltonian integrals and complex Gaunt coefficients.

For ∼1000 atom problems, the resulting generalized eigenvalue problem must be solved for a dense, complex Hermitian matrix with dimension of order 10^5. Since in a materials design problem these simulations will have to run primarily on large parallel supercomputers that cannot hold these matrices on individual nodes, the implementation must be designed for distributed memory architectures. Thus the underlying arrays must be partitioned in such a way that the above construction can be executed with minimum communication and results in Hamiltonian and overlap matrices that have the desired block-cyclic data distribution of the distributed eigensolver.

The matrix multiplies in equations (6) and (7) imply a block-cyclic distribution for the array $A^{\mathbf{k}}_{\alpha L\nu}(\mathbf{G})$ of matching coefficients, where $\mathbf{G}$-vector and composite $\{\alpha, L, \nu\}$ indices are distributed respectively, over the columns and rows of a 2D MPI grid. This distribution, however, is very inefficient for the computation of the auxiliary array $B^{\mathbf{k}}_{\alpha L\nu}(\mathbf{G})$ (Eq. 8); the reason being, that in order to compute a local panel of B-coefficients, the sum over $\{L_2, \nu_2\}$ indices is needed which may run out of scope of the current MPI rank. This is a well known problem when, for some operations (e.g., FFT used in pseudopotential methods), it would be better to have the entire array on a node, whereas for others the data needs to be distributed. We solve this problem, as illustrated in Fig. (1), by replicating A-coefficients as slices of whole vectors created on the row ranks of the MPI grid. The memory overhead that we have to pay for such data replication is comparable to the size of the local panel (typically ∼1.5 Gb for a 10K×10K complex matrix panel). The slices of A are defined for the entire composite index $\{\alpha, L, \nu\}$ and for only $\sim 1/N^{\mathrm{MPI}}_{\mathrm{row}}$ fraction of $\mathbf{G}$-vectors assigned to a column of MPI ranks. The corresponding slice of B-coefficients is computed locally using Eq. (8) before scattering B back to the panels of the block-cyclic distribution. The second sliced panel data storage, which we have introduced in the LAPW formalism here, may be useful in iterative subspace diagonalization methods as well, where

[1]unit step function $\Theta(\mathbf{r})$ is defined to be 0 in the muffin-tin region and 1 in the interstitial

[2]surface contribution to kinetic energy can be derived form the Green's identity $\int_S f(\nabla g)d\vec{S} = \int_V \Big(f(\nabla^2 g) + (\nabla f)(\nabla g)\Big)dV$



Initial data is distributed in a block-cyclic fashion
Each MPI rank gets a panel of tiles
MPI communication
MPI ranks of each column swop blocks of panels
The slices of whole vectors are gathered on each MPI rank
[0, 0]
[0, 1]
[1, 0]
[1, 1]

Figure 1: (color online) 'Panel' and 'slice' storage of the data. For parallel linear algebra operations, arrays must be distributed in a block-cyclic fashion over a 2D grid of MPI ranks. In order to perform a local operation on a whole vector, the slices of vectors are gathered from panels or created locally on the corresponding row ranks of the MPI grid. To perform a distributed operation with PBLAS or ScaLAPACK the vectors are shuffled to the 'panel' storage.

current hybrid CPU-GPU implementations use a data layout that constantly requires data exchange between "band" and "G-vector" partitioning [26] and rely on expensive MPI all-to-all communications. In our proposed scheme $N_{\rm col}^{\rm MPI}$ chunks of the data are redistributed between $N_{\rm row}^{\rm MPI}$ ranks which reduces the communication cost by a factor of $N_{\rm col}^{\rm MPI}$.

Algorithm 1 Distributed Hamiltonian and overlap matrix setup

```
1: precompute Hamiltonian radial integrals and plane-wave coefficients of the interstitial potential
2: create local panels of matching coefficients $A^{\mathbf{k}}_{\alpha L\nu}(\mathbf{G})$ with $\{\alpha, L, \nu\}$ index distributed over rows and $\mathbf{G}$-vector index distributed over columns of the MPI grid.
3: create local slices of matching coefficients $A^{\mathbf{k}}_{\alpha L\nu}(\mathbf{G})$ with a full $\{\alpha, L, \nu\}$ index and a $\mathbf{G}$-vector index which is distributed in a block-cyclic manner over the columns and additionally split over the rows of the MPI grid.
4: create slice of matrix $B^{\mathbf{k}}_{\alpha L\nu}(\mathbf{G})$ using the slice of $A^{\mathbf{k}}_{\alpha L\nu}(\mathbf{G})$ matrix
5: change to a 'panel' storage for $B$-matrix
6: compute muffin-tin contribution to the overlap matrix using **pzgemm**: $\mathbf{A}^* \times \mathbf{A} \rightarrow O^{\mathbf{k}}_{\mathbf{GG}'}$
7: compute muffin-tin contribution to the Hamiltonian matrix using **pzgemm**: $\mathbf{A}^* \times \mathbf{B} \rightarrow H^{\mathbf{k}}_{\mathbf{GG}'}$
8: add interstitial contribution to the overlap and Hamiltonian matrices
```

The general procedure of setting up the Hamiltonian and overlap matrices is described in Algorithm 1. When this is done the generalized eigen-value problem:

$$\sum_{\mathbf{G}'} H^{\mathbf{k}}_{\mathbf{GG}'} C^{i\mathbf{k}}_{\mathbf{G}'} = \epsilon_{i\mathbf{k}} \sum_{\mathbf{G}} O^{\mathbf{k}}_{\mathbf{GG}'} C^{i\mathbf{k}}_{\mathbf{G}'} \ , \tag{11}$$

is solved and the lowest eigenvectors are found. It is possible to use the iterative diagonalization technique and reduce the computational cost [9], however the 'safe default' of major LAPW community codes is to solve the generalized eigen-value problem with the brute force dense matrix diagonalization.

After computing the eigenvectors, Kohn-Sham wave functions are created. We use the following wave function representation:

$$\psi_i(\mathbf{r}) = \begin{cases} \sum\limits_{L\nu} F^{i\mathbf{k}}_{\alpha L\nu} u^{\alpha}_{\ell\nu}(r) Y_L(\hat{\mathbf{r}}) & \mathbf{r} \in \mathrm{MT}\alpha \\ \sum\limits_{\mathbf{G}} \frac{1}{\sqrt{\Omega}} e^{i(\mathbf{G}+\mathbf{k})\mathbf{r}} C^{i\mathbf{k}}_{\mathbf{G}} & \mathbf{r} \in \mathrm{I}, \end{cases} \tag{12}$$

where $C^{i\mathbf{k}}_{\mathbf{G}}$ are the eigenvectors of the generalized eigenvalue problem and the muffin-tin expansion coefficients $F^{i\mathbf{k}}_{\alpha L\nu}$ are obtained from the matrix-matrix multiplication between the eigenvectors and matching coefficients:

$$F^{i\mathbf{k}}_{\alpha L\nu} = \sum_{\mathbf{G}} A^{\mathbf{k}}_{\alpha L\nu}(\mathbf{G}) C^{i\mathbf{k}}_{\mathbf{G}}. \tag{13}$$

The wave functions (represented by matrices **F** and **C**) are initially distributed in the block-cyclic manner which is not convenient for the charge density construction. Great simplification is achieved by switching to the 'slice' storage for the wave functions. In that case each MPI rank in the grid gets the subset of the whole wave function and computes its contribution to the charge density. At the end, the charge density is reduced over all MPI ranks and the full charge density is obtained.

The last step of the DFT self-consistency loop consists of generating new Hartree and exchange-correlation potentials. These operations are trivially parallelizable and will not be discussed in the context of the present work.

4. SOLUTION OF THE EIGENVALUE PROBLEM ON DISTRIBUTED HYBRID CPU-GPU ARCHITECTURES

Solving the generalized Hermitian-definite eigenvalue problem is one of the main bottlenecks that dominates many electronic structure computations [27, 4, 40], and in particular prevents from scaling LAPW-based methods to larger numbers of atoms.

Stated more generically, the need of many applications motivates the development of modern distributed eigensolvers for generalized Hermitian-definite problems of the form:

$$Ax = \lambda Bx, \tag{14}$$

where A is a Hermitian matrix and B is Hermitian positive definite. Solving (14) requires four phases:

1. **Generalized to standard eigenvalue transformation phase**: the matrix B is decomposed using a Cholesky factorization into $B = LL^H$, where ^{H} denotes conjugate-transpose. The resulting L factors are used to transform (14) to a standard Hermitian eigenproblem is $A_s z = \lambda z$, where $A_s = L^{-1}AL^{-H}$. After solving the standard Hermitian eigenproblem, the eigenvectors X of the generalized problem (14) are computed by back-solving with the Cholesky factor, $X = L^{-H}Z$. The technique to solve the standard Hermitian (symmetric) eigenproblem $A_s z = \lambda z$, i.e., finding its eigenvalues Λ and eigenvectors Z so that $A_s = Z\Lambda Z^H$, and follows the three phases described below [17, 2, 36];

2. **Reduction phase**: orthogonal matrices Q are applied on both the left and the right side of A_s to reduce it to a tridiagonal form matrix $T = Q^T A_s Q$ – hence, these are called "two-sided factorizations." Note that the use of two-sided orthogonal transformations guarantees that A_s has the same eigenvalues as the reduced matrix T, and the eigenvectors of A_s can be easily derived from those of the reduced matrix (step 4);

3. **Solution phase**: an eigenvalue solver such as the divide and conquer (D&C), the multiple relatively robust representations (MRRR), the bisection algorithm, or the QR iteration method computes the eigenvalues Λ and eigenvectors E of the tridiagonal matrix T, so that $T = E\Lambda E^H$, yielding Λ to be the eigenvalues of A_s;

4. **Back transformation phase**: if required, the eigenvectors Z of A_s are computed by multiplying E by the orthogonal matrices Q used in the reduction phase $Z = QE$.

The classical approach, implemented in LAPACK, follows the procedure above. The most time consuming is the reduction phase [17]. The reduction algorithm is blocking – a current block of columns (panel) is factored and the transformations used in the panel factorization are accumulated and applied at once to the trailing matrix as Level 3 BLAS. The panel factorization requires computing Level 2 BLAS symmetric matrix-vector products with the entire trailing matrix, and thus is a slow, memory bound computation. Furthermore, the approach features data dependencies and artificial synchronization points between the panel factorization and the trailing submatrix update steps that prevent the use of standard techniques to increase the intensity of the computation (e.g., look-ahead), where the slow panel factorizations are computed on the CPUs and overlapped with trailing matrix updates on the GPUs (used extensively in the one-sided LU, QR, and Cholesky factorizations). As a result, the algorithm follows an expensive fork-and-join parallel model, preventing overlap between the CPU and the GPU computation, as well as hiding CPU-GPU communication costs by overlapping them with GPU computations.
Peak performance model: The reduction to tridiagonal proceeds by computing a Hermitian matrix-vector product (zhemv: $8l^2$ flop, where l is the size of the Householder reflector at step i) and an update every n_b steps (zher2k: $8n_b k^2$ flop, where k is the size of the trailing matrix at a step i). For all the steps ($\frac{n}{n_b}$ steps), the total flop count is:

$$\text{flop} = 8\sum_{l=1}^{n-1} l^2 + 8n_b \sum_{k=0}^{(n-n_b)/n_b} k^2 \approx \frac{8}{3}n^3 + \frac{8}{3}n^3, \qquad (15)$$

where the first $8/3\ n^3$ flops are in zhemv and the second $8/3\ n^3$ are in zher2k. The peak performance P_{peak} can be expressed as:

$$\begin{aligned} P_{\text{peak}} &= \frac{\text{flop}}{t} = \frac{16/3\ n^3}{t_{\text{zhemv}} + t_{\text{zher2k}}} \\ &= \frac{2}{P_{\text{zhemv}}^{-1} + P_{\text{zher2k}}^{-1}} = \frac{2P_{\text{zher2k}}P_{\text{zhemv}}}{P_{\text{zher2k}} + P_{\text{zhemv}}} \\ &\le \frac{2P_{\text{zher2k}}P_{\text{zhemv}}}{P_{\text{zher2k}}} = 2P_{\text{zhemv}}, \end{aligned} \qquad (16)$$

where t is the time for the tridiagonalization routine, and t_{zhemv} and t_{zher2k} are the times spent in zhemv and zher2k, respectively. The zhemv routine (pzhemv for distributed) is memory bounded, hence its performance P_{zhemv} is limited by the memory bandwidth. Thus, Eq. (16) guarantees a low performance behavior of the classical tridiagonal reduction algorithm.

Recent research has been concentrated on the development of new algorithms, known as "two-stage" algorithms [30, 7, 20, 21, 33, 19], where a first stage uses Level 3 BLAS operations to reduce A_s to band form, and a second stage further reduces the matrix to the proper tridiagonal form. We developed our distributed eigensolver based on the "two-stage" techniques that allow us to exploit more parallelism, and use accelerator hardware more efficiently. The bases of the implementation of the distributed hybrid eigensolver presented in this paper is a GPU-enabled hybrid implementation of PBLAS. This implementation allows the user to call the PBLAS routines not only with matrices located in the host memory, but also with matrices located on the GPU memory. It is provided by CRAY through the libsci_acc library. This library is an extension to LibSci, the CRAY implementation of CPU-only BLAS, LAPACK, PBLAS and ScaLAPACK. In the following section we describe the details of our implementation.

4.1 Transformation from generalized to standard eigenvalue

The Cholesky decomposition is the first step of the transformation from generalized to standard eigenvalue problem. Matrix B is distributed among the compute nodes in a 2D block cyclic fashion with block size n_d. Fig. 2a shows the notations used. The diagonal block B_{ii} is always located on one node, while the panel B_i is owned by one column of nodes. We use the right-looking standard algorithm, that proceeds panel by panel, performing a single-node Cholesky decomposition on the diagonal block $B_{ii} = L_{ii}L_{ii}^H$ (L_{ii} is stored in B_{ii}). For this operation the single node hybrid Cholesky distribution provided by libsci_acc (zpotrf) is used. Then, to update the rest of the panel $L_i = B_i L_{ii}^{-H}$, pztrsm is used. In the next step the trailing matrix must be updated in the following way: $B_T = B_T - L_i L_i^H$. This operation is performed with the rank-k update pzherk. The width of the panel n_b can be chosen such that n_b is a divisor of the block size n_d. To avoid unnecessary copies between the hosts and the GPUs, matrix B resides in the GPU memory that also contributes to keeping the GPU busy as much as possible.

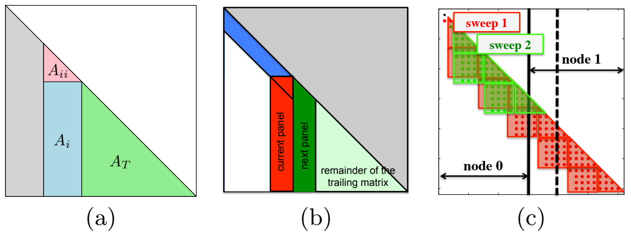


Figure 2: a) Notation used in the description of the transformation from generalized to standard eigenproblem. A_{ii} is the diagonal block, A_i is the panel, and A_T is the trailing matrix. b) Description of the reduction to band form (stage 1), panel factorization, and trailing matrix update. c) An example of the bulge chasing that shows the shared data region between two nodes.

In the next step we need to compute $A_s = L^{-1}AL^{-H}$. This corresponds to the ScaLAPACK pzhegst routine. The matrices A and L are distributed in a 2D block cyclic way with block size n_d as in the Cholesky decomposition (notation in Fig. 2a). Each step of this algorithm, which proceeds panel by panel, can be split in four phases (the full algorithm can be found in [23]): (1) compute the diagonal block A_{ii}, (2) partially update the panel A_i, (3) update the trailing matrix A_T, and (4) finish the update of the panel A_i. To increase the parallelism of this routine we use the same approach we used in [22], where phase (4) is delayed at the end of the routine since phases (1-3) of the next steps do not depend on it. This allows the final update of all the panels to be done in parallel, increasing the parallel efficiency.

Similarly to the Cholesky decomposition, phase (1) requires a single node hybrid implementation. Phases (2-4) require only PBLAS routines, in particular pztrsm, pzhemm, pzher2k, and pzgemm. Similarly to the Cholesky decomposition, the width of the panel n_b can be chosen such that n_b is a divisor of the block size used in the 2D block-cyclic distribution n_d. The matrices A and L are stored in the GPU memory to avoid unnecessary copies, and to keep the GPU as busy as possible.

In the last step of the generalized eigensolver the eigenvectors must be back-solved with the Cholesky factor L, $X = L^{-H}Z$. This operation is performed using the pztrsm routine.

4.2 Distributed Hybrid CPU-GPU Two-stage Tridiagonal Reduction

The two-stage reduction first reduces the dense Hermitian Matrix A_s to band form (band reduction). The band reduction is compute efficient since it replaces the memory-bound matrix-vector product, present in the classical one-stage tridiagonal reduction, with compute-intensive matrix-matrix product. However, the resulting matrix is banded, instead of tridiagonal, and needs an additional step ("bulge chasing") to be reduced to tridiagonal form.

4.2.1 *First Stage: Hybrid CPU-GPU Band Reduction*

The first stage applies a sequence of block Householder transformations to reduce a Hermitian dense matrix to Hermitian band form. This stage has been shown to have a good data access pattern and large portion of Level 3 BLAS operations [13, 15, 23]. It also enables the efficient use of GPUs by minimizing communication and allowing overlap of computation and communication. The Hermitian dense matrix A_s is distributed in a 2D block cyclic way with block size n_d. The algorithm proceeds panel by panel, performing a distributed QR decomposition for each panel to generate the transformation defined by the Householder reflectors V (i.e., an orthogonal transformation) required to zero out elements below the n_b-th subdiagonal. Then, the generated Householder transformation is applied from the left and the right to the trailing Hermitian matrix, according to

$$A_T = A_T - WV^H - VW^H, \tag{17}$$

where V and T define the blocked Householder transformation and W is computed as

$$\begin{aligned} W &= X - \tfrac{1}{2}VT^HV^HX, \text{ where} \\ X &= AVT. \end{aligned} \tag{18}$$

Since the panel factorization consists of a QR factorization performed on a panel shifted by n_b rows below the diagonal, the panel factorization by itself does not require any operation on the data of the trailing matrix, making it an independent task. This allows the algorithm to start the factorization of the panel of step $i+1$ after its update, hence, the computation of the panel factorization can be overlapped with the computation of the rest of the trailing matrix.

The hybrid CPU-GPU implementation is described below. First, on the CPU, we compute the QR decomposition (pzgeqrf) of the distributed panel at step i (red panel of Fig. (2b)). Once the panel factorization of step i is finished, we compute W on the GPUs, as defined by equation (18), using Level 3 parallel BLAS. Once W is computed, the trailing matrix update defined by equation (17) can be performed using pzher2k. In order to allow overlap between CPU and GPU computation, the trailing matrix update is split into two pieces. First, the panel of the next step $(i+1)$ (dark green panel of Fig. (2b)) is updated on the GPUs. Then, the remainder of the trailing submatrix is updated on the GPUs using pzher2k, which overlaps with the factorization of the panel of step $i+1$ on the CPUs. In this way, a part of the panel factorization and the associated communication are hidden by overlapping with GPU computation. This procedure is similar to the look-ahead technique typically used in the one-sided dense matrix factorizations.

4.2.2 *Second Stage: Cache-Friendly Computational Kernels*

The band matrix A_b is further reduced to the tridiagonal form T using the bulge chasing technique. This procedure annihilates the extra off-diagonal elements by chasing the created fill-in elements down to the bottom right side of the matrix using successive orthogonal similarity transformations. Each annihilation of the n_b non-zero element below the off-diagonal of the band matrix is called a *sweep*. This stage involves memory-bound operations and requires the band matrix to be accessed from multiple disjoint locations. In other words, there is an accumulation of substantial latency overhead each time different portions of the matrix are loaded into cache memory, which is not compensated for by the low execution rate of the actual computations (the so-called surface-to-volume effect). To overcome these critical limitations, we used a bulge chasing algorithm to use cache friendly kernels combined with fine grained memory aware tasks in an out-of-order scheduling technique, which considerably enhances data locality. We refer the reader to [23, 20] for a detailed description of the technique.

This stage is, in our opinion, one of the main challenges for the hybrid distributed algorithm, as it is difficult to track the data dependencies and move data between nodes. We describe the technique using the simple example illustrated in Fig. (2c). In order to minimize the amount of data that must be communicated, we define a *ghost region* which corresponds to the region between the solid and the dashed vertical black line in Fig. (2c). The region contains the data that moves back and forth between two nodes. In order to analyze the data movement in this region, denote the tasks that correspond to the elimination of the first two sweeps by the red color in Fig. (2c). The computational tasks generated by sweep 2 partially overlap with the tasks generated by sweep 1. In particular, the data used by the first five tasks of sweep 2, which are represented by the green color in Fig. (2c, overlap with those of sweep 1. The tasks that affect data on the border between two nodes require special attention (for example the fifth green task of sweep 2) to track the data dependencies between different tasks of different sweeps. For example, it can be observed that the data used by the fifth green task of sweep 2 overlap the fifth one from sweep 1 by one extra column to the right. Note that since the fifth task of sweep 1 has been allocated to node 0, it will be beneficial to only get one column from node 1 and let node 0 compute the fifth task of sweep 2. As a result, for every sweep, one column of the ghost region is sent to the previous rank processor. This is repeated for n_b sweeps, then at the level, all the data of the ghost region becomes owned by processor 0, hence it must be sent back to node 1.

We implemented the second stage to execute entirely on the CPUs of the different nodes. The main motivations are the large communication requirements of the algorithm and the fact that the accelerators perform poorly when dealing with memory-bound fine-grained computational tasks (such as bulge chasing). We dedicated a specific thread per node to manage the communication. The computational tasks are distributed among the remaining threads. This allows for hiding the communication needed by the algorithm.

Just as we established with Eq. (15) for the classic one-stage approach, we can do the same for our two-stage implementation:

$$\begin{aligned}\text{flop} &= \underbrace{\tfrac{4}{3}n^3(\text{compute bounded})}_{\text{first stage}} + \underbrace{6n_bn^2(\text{memory bounded})}_{\text{second stage}}\\ &= \frac{4}{3}n^3 + \Theta(n^2),\end{aligned} \tag{19}$$

where n is the matrix size and n_b is the bandwidth of the band matrix after the first stage. The cubic-order (first stage) operation is performed with Level 3 BLAS. The second stage only performs a small (and decreasing with n) percentage of the flops by using custom kernels for the bulge-chasing [20]. Clearly, all of the cubic-order flops are performed using the Level 3 BLAS. Unfortunately, since the band matrix can be distributed only in a 1D block cyclic way, the scaling of the second stage is limited.

4.2.3 *Tridiagonal eigenvalue solver*

The divide and conquer (D&C) algorithm, introduced by Cuppen [11] computes all the eigenvalues and eigenvectors of the real tridiagonal matrix T. Many serial and parallel implementations of the Cuppen eigensolver have been proposed in the past [12, 16, 25, 39, 41, 42]. The overall D&C algorithm approach splits the problem into two subproblems (the son nodes). They represent a rank-one modification of the parent node. Each of the son subproblems can then be solved independently. The split process can be repeated recursively, and a binary tree that represents all the subproblems can be built. In the end, starting from the bottom row of the tree, the suproblems are merged to get the final solution. The merge phase proceeds in the following way. The size n matrix T is split into two subproblems: T_1 of size n_1, and T_2 of size $n_2 = n - n_1$ (see (20)). Let the son subproblems already be solved and their solutions be $T_1 = \tilde{E_1}\tilde{\Lambda_1}\tilde{E_1}^T$ and $T_2 = \tilde{E_2}\tilde{\Lambda_2}\tilde{E_2}^T$, where $(\tilde{\Lambda_j}, \tilde{E_j})$, $j = 1, 2$ are the eigenpairs of the matrix T_j. The rank-one modification eigenproblem is then solved finding the eigenpairs $(\tilde{\Lambda_0}, \tilde{E_0})$. The eigenvalues and eigenvectors of T are then computed using:

$$\begin{aligned} T &= \begin{pmatrix} T_1 & 0 \\ 0 & T_2 \end{pmatrix} + \rho \mathbf{v}\mathbf{v}^T \\ &= \begin{pmatrix} \tilde{E_1} & 0 \\ 0 & \tilde{E_2} \end{pmatrix} \left\{ \begin{pmatrix} \tilde{\Lambda_1} & 0 \\ 0 & \tilde{\Lambda_2} \end{pmatrix} + \rho \mathbf{u}\mathbf{u}^T \right\} \begin{pmatrix} \tilde{E_1} & 0 \\ 0 & \tilde{E_2} \end{pmatrix}^T \\ &= \begin{pmatrix} \tilde{E_1} & 0 \\ 0 & \tilde{E_2} \end{pmatrix} \left(\tilde{E_0}\tilde{\Lambda_0}\tilde{E_0}^T \right) \begin{pmatrix} \tilde{E_1} & 0 \\ 0 & \tilde{E_2} \end{pmatrix}^T = E\Lambda E^T. \end{aligned} \tag{20}$$

The D&C implementation we present in this work is based on the ScaLAPACK implementation, but has been modified in the following way: (1) the eigenvector matrices E, $\tilde{E_1}$, and $\tilde{E_2}$ are located in the GPU memory, (2) the eigenvectors of the rank-1 modification $\tilde{E_0}$ are generated by the CPUs and copied to the GPUs, allowing us to perform the matrix-matrix multiplication directly on the GPUs, and (3) the last merge step is modified such that only the requested percentage of the eigenvectors are computed, reducing the total amount of computation.

4.3 Back Transform the Eigenvectors of the Two Stage Technique

In the context of the two-stage approach, the first stage reduces the original dense matrix A_s to a band matrix $A_b = Q_1^H A_s Q_1$, and the second, bulge-chasing stage reduces the band matrix A_b to the tridiagonal form $T = Q_2^H A_b Q_2$. Thus, when the eigenvectors matrix Z of A_s are requested, the eigenvectors matrix E resulting from the tridiagonal eigensolver needs to be back transformed by the Householder reflectors generated during the reduction phase, according to

$$Z = Q_1 Q_2 \ E, \tag{21}$$

where Q_1 and Q_2 are defined by the Householder reflectors (V_1, τ_1) and (V_2, τ_2) generated during the two-stage reduction.

From a practical standpoint, the back transformation Q_2 is not as straightforward as the one of Q_1. In particular, because of the complications of the bulge-chasing mechanism, the order in which the Householder reflectors need to be applied, and the overlap of the data regions they modify, makes this task more complex. In order to achieve very good scalability and performance, the main focus of the imple-

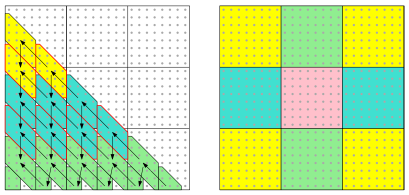


Figure 3: Left: The distribution and the order of application of the Householder reflector blocks V_2. Right: The distribution of the eigenvectors matrix.

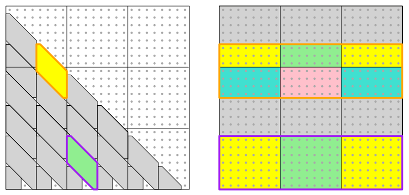


Figure 4: The data regions affected by the diamond shape block. The green diamond with purple border belong to the first category while the yellow diamond with yellow border belong to the second category.

mentation is to create compute intensive operations to take advantage of the efficiency of Level 3 BLAS. To this end, we implemented a technique that accumulates and groups the Householder reflectors. Our technique is represented by the diamond-shaped region in Fig. (3), where each diamond is considered one block, and the arrows represent the dependency order that their application needs to follow.

In a distributed environment, the data is distributed among the nodes and, a difficulty of applying Q_2, described above, needs to be carefully handled. Fig. (3) illustrates the distribution of both the eigenvectors matrix and the Householder reflectors V_2, where each color represents a node. We can observe that the diamond blocks of V_2 are divided into two categories. The first category consists of the diamond blocks that modify (touch) the data of only one row of the node grid (for example the green diamond with purple border modifies the data owned by the third row of processors highlighted by the purple border of Fig. (4)), while the second category is composed of those that affect the data owned by two rows of the node grid (the yellow diamond with yellow border modifies the data owned by the first and the second row of processors highlighted by the yellow border of Fig. (4)). Each Householder reflector group of the first category must be broadcast among its corresponding row of the node grid, then the application of the Householder transformation defined by the block is independent among the different nodes. On the other hand, the portion of the eigenvectors, modified by the application of the transformation defined by each Householder reflector group of the second category, is shared between two rows of the node grid. Therefore the block must be broadcast between the two rows of nodes and the application of the transformation requires a sum reduce between the two rows. In our implementation, the CPUs manage the communication and the GPUs apply the Householder transformations, hence, there is an overlap between communication and computation, since during the application of the Householder transformation defined by the current block, the CPUs can broadcast the next block.

The back transformation of Q_1 is similar to the classical back transformation for the one-stage algorithm that corresponds to the pzunmqr routine of ScaLAPACK. It involves efficient Level 3 parallel BLAS kernels and therefore is performed efficiently on the GPUs.

Note that when the eigenvectors are required, the two-stage approach has the extra cost of the back transformation of Q_2, i.e., $8n^2k$ flops, where k is the number of the computed eigenvectors. However, experiments show that even with this extra cost the overall performance of the generalized eigensolver using the two-stage approach can be faster than the solvers using the one-stage approach, especially when only a fraction of the eigenvectors is required, since the extra operations depend linearly on k.

5. RESULTS

We now turn to characterizing the performance of a complete $\sim$1000 atom electronic benchmark run in terms of time and energy to solution on distributed CPU and hybrid CPU-GPU architectures, implementing the algorithms discussed in the previous sections. The system used to run our benchmark is a 28-cabinet Cray XC30 with a fully populated third dimension of the dragonfly network. All compute nodes of the system have an 8-core Intel Xeon E5-2670 multi-core CPU and one NVIDIA K20X GPU. In a performance comparison between a distributed multi-core and a hybrid CPU-GPU implementation, we keep the total number of sockets involved in the computation constant. That is, we compare the performance of the multi-core implementation running on the CPUs of $2n$ nodes with corresponding hybrid CPU-GPU implementation running on n nodes.

The Cray XC30 platform is equipped with advanced software and hardware features for monitoring energy consumption [14]. Since in the present work we are interested in comparing energy to solution between a CPU-only and a hybrid CPU-GPU implementation, we have used only the power sensors on the blade that measure the consumption of the nodes. Off-node components such as the Aires network and blowers are ignored. Power consumption of the idle GPU has been subtracted from the energy measurements we report for CPU-only runs.

5.1 Full LAPW application benchmark

The algorithms described in section 3 have been implemented in a new LAPW library SIRIUS[1], which was created within the work package eight (WP8) of the PRACE second implementation phase (PRACE-2IP) project with the main goal of finding major performance bottlenecks in the ground-state calculations in both Exciting (exciting-code.org) and Elk (elk.sourceforge.net) codes. SIRIUS invokes architecture dependent backends and the appropriate distributed eigensolvers. A performance analysis of the latter will be given in the next sub-section.

For the test case we choose a full-potential DFT ground states simulation of a Li-ion battery cathode. A unit cell with a Li-intercalated CoO_2 supercell containing 432 formula units of CoO_2 and 205 atoms of lithium (1501 atoms in total) was created. The Li sites were randomly populated to produce a $\sim$50% intercalation. Initial unit cell parameters were taken from Ref. [32]. This example represents the broad class of problems such as diluted magnetic semiconductors, energy formation of vacancies, impurity levels in band insulators, alloys, etc., where the large supercell setup is required. Below, we demonstrate that the full-potential simulations of large unit cells are feasible in a acceptable time ($\sim$17 minutes per SCF iteration) with a reasonable amount of resources in terms of hardware and energy. We setup[3] a single Γ-point calculation with $\sim$115000 basis functions and $\sim$7900 lowest bands to compute, and we measured the execution time of major parts of the DFT self-consistency cycle. The results are collected in Table 5.1, where we report, in columns, the time for the eigenvalue problem setup, eigenvalue problem solution, total DFT iteration time, and the rest of the DFT cycle, which incorporates construction of the wave-functions, construction of the new charge density, and calculation of the new effective potential. At the moment, only the Hamiltonian and overlap matrix setup and eigenvalue solvers are GPU-aware and the rest of the code is executed on the CPU. We consider 196-node hybrid CPU-GPU run with one MPI rank and eight OpenMP threads per node ('14×14 (1R:8T) hybrid' in the table) as a reference point because it corresponds to the minimum amount of nodes on which the calculation fits and at the same time it gives the best $\sim$54 node-hours per iteration measure. The hardware footprint of this run is 392 sockets. For the CPU-only run we setup two alternative configurations: 28×28 MPI grid with 2 MPI ranks per node and 4 OpenMP threads per MPI rank (with ScaLAPACK and ELPA2 solvers), as well as a 20×20 MPI grid with a single MPI rank and 8 OpenMP threads per node (ELPA2 solver). The hardware footprint of the former is 392 active sockets, and 400 active sockets for the latter, which is comparable to the 392-socket hybrid reference configuration. As we will see in the next section, the ELPA2 eigenvalue solver is more efficient with more MPI ranks per socket and fewer OpenMP threads per rank. However, a memory limitation in the entire application forces us to use the configurations with fewer MPI ranks per socket and more OpenMP threads.

The results in Table 5.1 show that the CPU-only runs are much more efficient when using the ELPA2 library for the eigensolver rather than ScaLAPACK. In terms of time and energy to solution, the 28×28 and the 20×20 MPI grid configurations are approximately the same, which is not surprising since they use about the same number of active sockets. The hybrid reference run with 14×14 nodes is about 15% faster than CPU-only runs with ELPA2, and a factor of 2 more energy efficient. We have also performed a 20×20 MPI grid hybrid run on an equivalent number of CPU-GPU nodes, i.e. with 800 active sockets or two times as many as the other runs. This larger hybrid run is about 30% faster and more energy efficient than the CPU-only runs, and has the fastest time to solution of all configurations. However, the smaller of the two hybrid runs is more efficient in terms of both throughput and energy to solution. These results will have to be considered in large materials design simulations.

5.2 Eigensolver benchmark

Given the dominance of the generalized eigenvalue problem in the full LAPW runs presented in the previous section,

	set $\mathbf{H}, \mathbf{O}$	$\mathbf{H}C = \epsilon \mathbf{O}C$	the rest	total	energy
28×28 (2R:4T) ScaLAPACK	382.5	3166.8	69.2	3618.5	39.46
28×28 (2R:4T) ELPA2	383.2	705.3	63.6	1152.1	17.40
20×20 (1R:8T) ELPA2	374.0	720.5	61.1	1155.6	16.9
14×14 (1R:8T) hybrid	159.9	741.8	84.8	986.5	8.27
20×20 (1R:8T) hybrid	96.9	652.1	58.9	807.9	12.49

Table 1: Execution time (in seconds) and energy consumption (kWh) per iteration of major parts of the CPU-only and hybrid CPU-GPU versions of the LAPW code

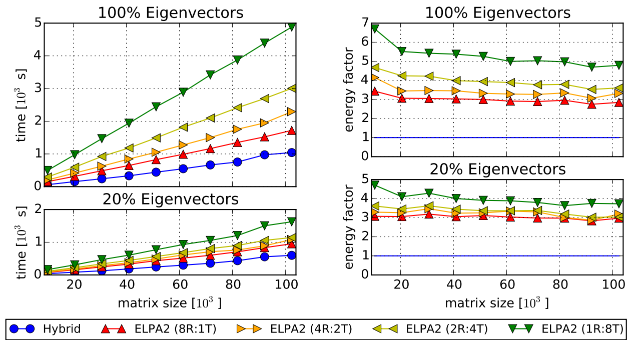


Figure 5: Weak scaling of time and energy of ELPA2, being run with different configurations, compared to the hybrid generalized eigensolver. The hybrid solver is executed on $(n/10240)^2$ nodes, while ELPA2 is on $2\,(n/10240)^2$ nodes, where n is the matrix size.

and the similarity in performance between the ELPA2 CPU only and the hybrid CPU-GPU implementations stipulated by the results in table 5.1, we will now give a more thorough performance study of the eigensolvers on the two architectures. In order to properly gauge the hybrid implementation of our new, distributed hybrid CPU-GPU eigensolver, we compare the performance on n hybrid nodes to $2n$ cpu-only nodes running the distributed multi-core version of ELPA.

Fig. (5) presents the weak scaling of time and energy to solution of the generalized eigenvalue solvers, when different percentages of the eigenvectors are required. The application runs on $(n/10240)^2$ nodes, where n is the matrix size. This quantity of nodes represents the minimum amount of resources needed by the hybrid solver to execute. In particular, the solver is bounded by the quantity of GPU memory in the NVIDIA K20X. ELPA2 is executed on twice as many nodes ($2\,(n/10240)^2$) using only the CPUs, to satisfy the socket to socket comparison model. We run ELPA2 with four different configurations. The configurations are denoted by (N_rR:N_tT), where N_r is the number of MPI ranks per socket and N_t is the number of threads per MPI rank. Since each configuration holds $N_r N_t = 8$, this means all cores of each socket are always used. The ELPA2 solver runs more efficiently in the configurations that have more MPI ranks.

[3]the following LAPW parameters were used: $R_{MT} = 1.72$ a.u., $\ell_{max}^{\mathrm{APW}} = \ell_{max}^{V} = \ell_{max}^{\rho} = 8$, $|\mathbf{G}|_{max} = 20$ a.u.$^{-1}$, $R_{MT} \times |\mathbf{G}+\mathbf{k}|_{max} = 7$

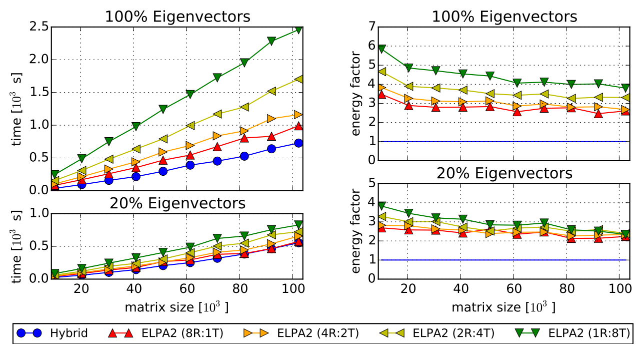


Figure 6: Weak scaling of time end energy of ELPA2, being run with different configuration, compared to the hybrid generalized eigensolver. The hybrid solver is executed on $2\,(n/10240)^2$ nodes, while ELPA2 on $4\,(n/10240)^2$ nodes, where n is the matrix size.

However, as we have seen in the previous section, memory limitations in the full application may force us to use fewer MPI ranks per socket and more OpenMP threads. The hybrid (CPU-GPU) implementation is a factor 1.5× to 2× faster than the most efficient configuration of the ELPA2 (CPU-only) implementation. The hybrid architecture is 2 times more energy efficient than the distributed multi-core architecture.

Fig. (6) shows the results with double of the nodes used in Fig. (5), i.e., $2\,(n/10240)^2$ for the hybrid solver, and $4\,(n/10240)^2$ for ELPA2. In this case the hybrid implementation and the fastest ELPA2 configuration require comparable time to solve the same problem. However, the hybrid architecture is between 2 and 2.5 times more energy efficient than the CPU-only architecture.

6. SUMMARY AND CONCLUSIONS

We have presented a high-performance implementation of the LAPW methods and demonstrated the feasibility of ∼1000 atom ground state calculations with good turnaround time on clusters with a few hundred nodes. Since the two major time-consuming parts of the LAPW methods – the setup and solution of the generalized eigenvalue problem – both have algorithmic complexity $O(N_{atom}^3)$, each must be implemented in a scalable way. One key ingredient for this new implementation is thus a distributed setup of the LAPW Hamiltonian and overlap matrix that runs on both multicore CPU and hybrid CPU-GPU nodes alike, and results in the same block-cyclic data distribution used by common linear algebra libraries such as ScaLAPACK. Furthermore, we have introduced a dual 'panel-slice' storage of the relevant arrays, which allows performing local and distributed operations on the data efficiently at the expense of a small increase in memory size.

On standard multi-core nodes, the distributed eigenvalue problem for dense Hermitian matrices has been solved with the corresponding routines of the ScaLAPACK and ELPA2 libraries, where clearly the best performance and scalability is achieved with the latter. For hybrid CPU-GPU systems we have discussed in detail a novel algorithm that implements a two-stage solver on computer architectures with heterogeneous, GPU accelerated nodes.

We have presented performance benchmarks with realistic calculations that use a Li-intercalated CoO_2 super cell with 1501 atoms. We execute a full SCF iteration step in less than 20 minutes on 196 hybrid CPU-CPU nodes and about 30 minutes on an equivalent number of 400 CPU sockets. These results show that highly accurate and transferable quantum simulations are now usable for high-throughput materials search problems, given the necessary computing capabilities. All our benchmark runs show that energy efficiency significantly favors the hybrid CPU-GPU architecture over a traditional multi-core architecture.

Furthermore, the implementation and results we presented demonstrate how complex codes and algorithms can be implemented in a performance portable way for such diverse architectures as multi-core and hybrid CPU-GPU systems. Key to this is the separation of concerns, where the complexity of hardware-specific programming models can be hidden into libraries, be it general linear algebra packages such as ELPA and MAGMA, or domain specific libraries such as the SIRIUS library introduced here for the LAPW and similar methods to solve the electronic structure problem in materials.

7. ACKNOWLEDGMENTS

The authors would like to thank the NSF grant #ACI-1339822 and NVIDIA for supporting this research effort.

8. REFERENCES

[1]

[2] J. Aasen. On the reduction of a symmetric matrix to tridiagonal form. *BIT Numerical Mathematics*, 11(3):233–242, 1971.

[3] O. K. Andersen. Linear methods in band theory. *Phys. Rev. B*, 12(8):3060–3083, Oct 1975.

[4] 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.

[5] 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 Comput.*, 37(12):783–794, Dec. 2011.

[6] S. R. Bahn and K. W. Jacobsen. An object-oriented scripting interface to a legacy electronic structure code. *Comput. Sci. Eng.*, 4(3):56–66, MAY-JUN 2002.

[7] P. Bientinesi, F. D. Igual, D. Kressner, and E. S. Quintana-Ortí. Reduction to condensed forms for symmetric eigenvalue problems on multi-core architectures. In *Proceedings of the 8th international conference on Parallel processing and applied mathematics: Part I*, PPAM'09, pages 387–395, Berlin, Heidelberg, 2010. Springer-Verlag.

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

[9] P. Blaha, H. Hofstätter, O. Koch, R. Laskowski, and K. Schwarz. Iterative diagonalization in augmented plane wave based methods in electronic structure calculations. *Journal of Computational Physics*, 229(2):453–460, Jan. 2010.

[10] P. E. Blöchl. Projector augmented-wave method. *Phys. Rev. B*, 50:17953–17979, Dec 1994.

[11] J. J. M. Cuppen. A divide and conquer method for the symmetric tridiagonal eigenproblem. *Numerische Mathematik*, 36(2):177–195, 1980.

[12] J. J. Dongarra and D. C. Sorensen. A fully parallel algorithm for the symmetric eigenvalue problem. *SIAM J. Sci. Stat. Comput.*, 8(2):139–154, Mar. 1987.

[13] J. J. Dongarra, D. C. Sorensen, and S. J. Hammarling. Block reduction of matrices to condensed forms for eigenvalue computations. *Journal of Computational and Applied Mathematics*, 27(1-2):215 – 227, 1989.

[14] G. Fourestey, B. Cumming, L. Gilly, and T. C. Schulthess. First experiences with validating and using the Cray power management database tool. *Proceedings of CUG Meeting*, 2014. (reprint available on arxiv.org under arXiv:1408.2657).

[15] W. Gansterer, D. Kvasnicka, and C. Ueberhuber. Multi-sweep algorithms for the symmetric eigenproblem. In *Vector and Parallel Processing - VECPAR'98*, volume 1573 of *Lecture Notes in Computer Science*, pages 20–28. Springer, 1999.

[16] K. Gates and P. Arbenz. Parallel divide and conquer algorithms for the symmetric tridiagonal eigenproblem, 1994.

[17] G. H. Golub and C. F. Van Loan. *Matrix Computations (3rd Ed.)*. Johns Hopkins University Press, Baltimore, MD, USA, 1996.

[18] J. Grotendorst, S. Blügel, and D. Marx, editors. *Computational Nanoscience: Do It Yourself!*, volume 31 of *NIC serie*. Forschungszentrum Jülich, 2006.

[19] A. Haidar, J. Kurzak, and P. Luszczek. An improved parallel singular value algorithm and its implementation for multicore hardware. In *SC13, The International Conference for High Performance Computing, Networking, Storage and Analysis*, Denver, Colorado, USA, November 17-22 2013.

[20] A. Haidar, H. Ltaief, and J. Dongarra. Parallel reduction to condensed forms for symmetric eigenvalue problems using aggregated fine-grained and memory-aware kernels. In *Proceedings of SC '11*, pages 8:1–8:11, New York, NY, USA, 2011. ACM.

[21] A. Haidar, H. Ltaief, P. Luszczek, and J. Dongarra. A comprehensive study of task coalescing for selecting parallelism granularity in a two-stage bidiagonal reduction. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, Shanghai, China, May 21-25 2012. ISBN 978-1-4673-0975-2.

[22] A. Haidar, R. Solcà, M. Gates, S. Tomov, T. C. Schulthess, and J. Dongarra. Leading Edge Hybrid Multi-GPU Algorithms for Generalized Eigenproblems in Electronic Structure Calculations. In *Supercomputing*, pages 67–80. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[23] A. Haidar, S. Tomov, J. Dongarra, R. Solcà, and T. Schulthess. A novel hybrid CPU-GPU generalized eigensolver for electronic structure calculations based on fine grained memory aware tasks. *International Journal of High Performance Computing Applications*, August 2013.

[24] P. Hohenberg and W. Kohn. Inhomogeneous electron gas. *Phys. Rev.*, 136(3B):B864–B871, Nov 1964.

[25] L. C. F. Ipsen and E. R. Jessup. Solving the symmetric tridiagonal eigenvalues problem on the hypercube. *SIAM J. Sci. Stat. Comput.*, 11:203–229, March 1990.

[26] W. Jia, J. Fu, Z. Cao, L. Wang, X. Chi, W. Gao, and L.-W. Wang. Fast plane wave density functional theory molecular dynamics calculations on multi-gpu machines. *Journal of Computational Physics*, 251(0):102 – 115, 2013.

[27] P. R. C. Kent. Computational challenges of large-scale, long-time, first-principles molecular dynamics. *Journal of Physics: Conference Series*, 125(1):012058, 2008.

[28] W. Kohn. Density functional and density matrix method scaling linearly with the number of atoms. *Phys. Rev. Lett.*, 76:3168–3171, Apr 1996.

[29] W. Kohn and L. J. Sham. Self-consistent equations including exchange and correlation effects. *Phys. Rev.*, 140(4A):A1133–A1138, Nov 1965.

[30] B. Lang. Efficient eigenvalue and singular value computations on shared memory machines. *Parallel Computing*, 25(7):845–860, 1999.

[31] K. Lejaeghere, V. Van Speybroeck, G. Van Oost, and S. Cottenier. Error Estimates for Solid-State Density-Functional Theory Predictions: An Overview by Means of the Ground-State Elemental Crystals. *Critical Reviews in Solid State & Materials Sciences*, 39:1–24, Jan. 2014.

[32] Q. Lin, Q. Li, K. E. Gray, and J. F. Mitchell. Vapor growth and chemical delithiation of stoichiometric $LiCoO_2$ crystals. *Crystal Growth and Design*, 12(3):1232–1238, 2012.

[33] H. Ltaief, P. Luszczek, A. Haidar, and J. Dongarra. Enhancing parallelism of tile bidiagonal transformation on multicore architectures using tree reduction. In R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Wasniewski, editors, *Proceedings of 9th International Conference, PPAM 2011*, volume 7203, pages 661–670, Torun, Poland, 2012.

[34] A. Marek, V. Blum, R. Johanni, V. Havu, B. Lang, T. Auckenthaler, A. Heinecke, H.-J. Bungartz, and H. Lederer. The elpa library - scalable parallel eigenvalue solutions for electronic structure theory and computational science. *Psi-K Research Highlight*, 2014(1), Jan. 2014.

[35] S. P. Ong, W. D. Richards, A. Jain, G. Hautier, M. Kocher, S. Cholia, D. Gunter, V. L. Chevrier, K. A. Persson, and G. Ceder. Python materials genomics (pymatgen): A robust, open-source python library for materials analysis. *Computational Materials Science*, 68(0):314 – 319, 2013.

[36] B. N. Parlett. *The Symmetric Eigenvalue Problem*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1998.

[37] J. P. Perdew, A. Ruzsinszky, J. Tao, V. N. Staroverov, G. E. Scuseria, and G. I. Csonka. Prescription for the

design and selection of density functional approximations: more constraint satisfaction with fewer fits. *The Journal of Chemical Physics*, 123(6):62201–62201, Aug. 2005.

[38] E. Prodan and W. Kohn. Nearsightedness of electronic matter. *Proceedings of the National Academy of Sciences of the United States of America*, 102(33):11635–11638, 2005.

[39] J. Rutter and J. D. Rutter. A serial implementation of cuppen's divide and conquer algorithm for the symmetric eigenvalue problem, 1994.

[40] D. J. Singh. *Planewaves, pseudopotentials and the LAPW method*. Kluwer, Boston, MA, 1994.

[41] D. C. Sorensen and P. T. P. Tang. On the orthogonality of eigenvectors computed by divide-and-conquer techniques. *SIAM J. Numer. Anal.*, 28(6):1752–1775, 1991.

[42] F. Tisseur and J. Dongarra. Parallelizing the divide and conquer algorithm for the symmetric tridiagonal eigenvalue problem on distributed memory architectures. *SIAM J. SCI. COMPUT*, 20:2223–2236, 1998.

[43] D. Vanderbilt. Soft self-consistent pseudopotentials in a generalized eigenvalue formalism. *Phys. Rev. B*, 41:7892–7895, Apr 1990.