Performance of parallel eigensolvers used for nuclear motion calculations on DiRAC CSD3 HPC platform

Phillip A. Coles, Jonathan Tennyson, Jeremy Yates

December 2, 2019

Department of Physics and Astronomy, University College London, London WC1E 6BT, UK

1 Introduction

The performance of numerical eigensolver library packages on various HPC systems has been the topic of numerous short-term investigations. Usually the related physical application is one of quantum chemistry, which often requires the diagonalisation of thousands of small or moderately sized $n < 100\ 000$ matrices (where n is the leading matrix dimension). In contrast, relatively little attention has been paid to the performance scalability of eigensolver packages for large $n > 100\ 000$ matrices. Such applications arise in theoretical molecular spectroscopy where the molecular energy levels can be calculated by constructing and diagonalising a series of quantum-mechanical molecular Hamiltonian matrices according to the variational principle. These matrices are dense, real, symmetric and require double precision numerics. For 4-5 atomic systems a sufficiently complete representation of the internal energy level spectrum of the molecule typically requires diagonalisation of several hundred matrices varying in size from 1000×1000 to $400\ 000 \times 400\ 000$. However, a current challenge within the field of theoretical spectroscopy is in applications for large polyatomic molecules, e.g, 8 or more atoms, especially hydro-carbons, which will require diagonalisation of even larger matrices. Treatment of these molecules is hindered by the 'curse of dimensionality', which refers to the exponential scaling of the Hamiltonian matrix as $n \sim A^{N_{\text{atom}}}$, where A is a constant and N_{atom} is the number of atoms in the molecule.

In the following report we investigate the performance scalability of two numerical eigensolver libraries that are utilised in the *ab initio* spectroscopic code TROVE [32], on the DiRAC HPC platforms CSD3 and Wilkes2. We begin with a brief overview of TROVE and the eigendecomposition challenges faced by TROVE during a typical study of small and large molecules. Next, standard approaches for solving the eigenvalue problem for dense symmetric matrices are described from a theoretical perspective and several software implementations of these algorithms are reviewed. Two leading direct eigensolver libraries, ELPA and ScaLAPACK, that are particularly relevant to the DiRAC HPC platforms that provide the main environment in which TROVE is run are then discussed. A series of benchmark calculations investigating the performance scalability of these eigensolvers are then performed on a number of large matrices. Finally, the results are discussed and interpreted within the context of TROVE, and their relevance for future TROVE calculations are stated.

2 TROVE

TROVE [32] is a variational nuclear motion program that has been extremely successful in computing synthetic rotational-vibrational spectra for a large number of small polyatomic molecules of astronomical and terrestrial interest. A typical production-quality investigation for a single 4-5 atomic molecule in TROVE requires the diagonalisation of around 200 matrices, with dimensions varying from 1000 to several hundred-thousand, for anywhere between 10% and 100% of all roots. If the matrix and requisite workspace can fit onto one compute node this is performed using LAPACK's DSYEVD [2]. Larger matrices require parallel implementations, in the case of TROVE, ScaLAPACK's PDSYEVD[26] is used as standard.

An overview of one series of matrices diagonalised during computation of the PH_3 line list [25] is presented in Figure 1. Each J value corresponds to a separate quantum rotational state that requires the diagonalisation of a separate Hamiltonian matrix. The number of roots required from the calculation (red) depends on the energy thresholds employed in TROVE, and the number of non-zero elements in a row (green) is included to illustrate the degree of sparsity that these matrices span. Clearly only at very high values of J does the use of iterative solvers become feasible. In fact, almost all matrices generated by TROVE fall within the regime of what, for the purposes of eigensolver selection, would be classified as dense. In the past TROVE has required 17% of roots from matrices as large as 600 000 in dimension, and the largest matrix diagonalised within TROVE was 750 000 in dimension [28].

In the near future TROVE is expected to begin the calculation of synthetic spectra for the hydrocarbon molecules C_2H_6 and C_3H_6. This work is already at the stage of model development [23] and will eventually require the diagonalisation of a series of matrices of unprecedented size within TROVE. It is therefore extremely important, in order to increase throughput whilst simultaneously minimising use of HPC resources, that these matrices can be diagonalised as efficiently as possible.

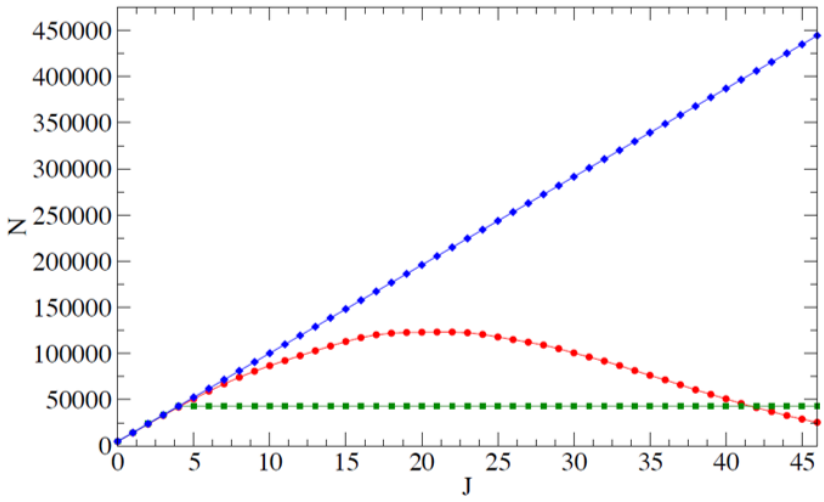

Figure 1: Series of matrices diagonalised during construction of the PH_3 line list [25]. The matrix leading dimension is shown in blue, the number of eigenpairs required is shown in red, the number of nonzero elements in a row is shown in green.

3 The eigenvalue problem

The standard eigenvalue problem for a symmetric matrix A is defined as

$$\mathrm{A}\mathbf{x} = \lambda\mathbf{x} \tag{1}$$

where $\mathbf{x}$ is the eigenvector corresponding to eigenvalue λ, which together are referred to as an eigenpair. For symmetric matrices all eigenvalues are real, and all eigenvectors form a complete, mutually orthogonal set.

To solve Eq. 1 one may choose to use either an *iterative* or *direct* approach. Direct solvers perform a set number of operations after which a solution is obtained. Iterative solvers generate a series of increasingly accurate approximations to the solution and terminate once the desired precision is obtained. For large, sparse matrices where few eigenpairs are required, i.e. $< 10\%$, iterative solvers are extremely effective. However, if more than 10% eigenvalues are required, they become extremely inefficient and are eschewed in favour of direct solvers. For almost all matrices generated by TROVE the use of direct solvers is more appropriate, therefore only direct solvers are considered henceforth.

The steps required in the direct approach to solving the standard eigenvector and eigenvalue problem are well known. They consist of the following:

1. Reduction of the matrix to tridiagonal form, typically by performing successive Household transformations.

2. Solution of the tridiagonal eigenproblem using one of the following methods: (i) QR iteration [12, 13]; (ii) Divide and Conquer [7, 14, 26]; (iii) Bisection for a selected subset of eigenvalues followed by inverse iteration for the eigenvectors [8, 9]; (iv) Multiple Relatively Robust Representations (MRRR) [11, 4, 31].

3. Back-transformation to determine the eigenvectors of the full problem from the eigenvectors of the tridiagonal problem.

Steps 1 and 3 are common to all algorithms discussed here, although the implementations vary. If Step 1 is performed using successive Householder transformations then the reduction takes $8/3n^3$ flops, which is reduced to $4/3n^3$ flops if only eigenvalues are required. The corresponding back-transformation in Step 3 then takes $2n^3$ flops.

Out of the various algorithms used for eigendecomposition of the tridiagonal matrix, QR iteration was historically the first to be developed. It computes all eigenvalues and optionally eigenvectors, and requires $\mathcal{O}(n^3)$ operations if eigenvectors are required and $\mathcal{O}(n^2)$ if only eigenvalues are required.

Bisection and inverse iteration (B&I) may be used to find a subset of eigenvalues and optionally eigenvectors in $\mathcal{O}(kn)$ flops, and so may be significantly faster than QR if only a subset of eigenpairs are required. However, for near-degenerate clusters of eigenvalues Gram-Schmidt orthogonalization is required at an additional cost of $\mathcal{O}(k^2n)$.

The MRRR algorithm can also compute a subset of eigenpairs, at a cost of $\mathcal{O}(kn)$ flops requiring only $\mathcal{O}(n)$ workspace. It is therefore generally preferred over QI and B&I. However, current implementations fail to achieve compute times reflective of this and, additionally, in certain cases the resulting eigenvectors may fail to achieve satisfying orthogonality.

Finally, the divide and conquer (D&C) algorithm, which computes the complete eigenspectrum, is arguably the most popular approach from the list above despite its larger $\mathcal{O}(n^2)$ workspace requirements. This is due to its robustness, and the fact that the $\mathcal{O}(n^3)$ cost is drastically reduced by a combination of deflation and the fact that most arithmetic operations correspond to matrix-matrix products which can be heavily optimised.

4 Parallel eigensolver libraries

A large number of numerical eigensolver libraries have been developed to solve the aforementioned eigenvalue problem on distributed memory machines. Arguably the most widely used is ScaLAPACK (Scalable Linear Algebra PACKage) [5], owing largely to its portability, efficiency, reliability and the scope of linear algebra routines it offers. Other notable libraries are PLAPACK [3] which was later superceded by Elemental [24], as well as ELPA [22, 20], Eigenexa [17], and MAGMA [27] for GPU architectures. From this list all are available as open source codes except for MAGMA. ELPA, in particular, has been shown to consistently outperform ScaLAPACK in both speed and scalability, and therefore presents a promising alternative to ScaLAPACK for implementation in TROVE. A Similar improvement has been noted for Elemental [24, 15], although in a recent study [16] on smaller matrices Elemental was generally outperformed by ELPA, particularly if only subsets of the spectrum were required.

In this work we decided to focus on ELPA as the most promising candidate for implementation in TROVE, and use MKL ScaLAPACK for benchmark comparisons. ELPA also provides a GPU implementation [19] which would align well with the infrastructure intended for the future HPC platforms of DiRAC 3. Although our study is limited to these two libraries due to man-hour and CPU-hour limitations, we hope that similar investigations using Elemental, Eigenexa and particularly MAGMA will be performed in future.

Finally, it is worth noting that a selection of shared memory parallel solvers have also been developed, such as LAPACK [2] and PLASMA [21]. Both of these have been utilised in previous TROVE calculations as an alternative to ScaLAPACK for large matrices, for example see Ref. [28]. However, as discussed in Section 2, a typical TROVE study requires the diagonalisation of 10's of large matrices, and therefore their repeated use becomes impractical owing to the rarity of individual compute nodes with sufficiently large memory (i.e. several Tb's).

4.1 ScaLapack

ScaLAPACK began development as a continuation of the LAPACK project, both of which are available in standard, open source implementations that can be compiled and run on any system, as well as highly optimised vendor specific implementations, e.g. Intel's Math Kernel Library (MKL) or IBM's ESSL. Low-level matrix and vector arithmetic operations within ScaLAPACK are handled by the BLAS (Basic Linear Algebra Subprograms) library, and a message passing interface is provided by the BLACS (Basic Linear Algebra Communication Subprograms) library. This limits the dependencies of ScaLAPACK to BLAS, BLACS and LAPACK.

ScaLAPACK utilises a 2D block-cyclic distribution of the matrix. Technically this means the processes of the parallel computer are first mapped onto a $\mathrm{P_{row}}\times\mathrm{P_{col}}$ rectangular grid, then the input matrix is decomposed into $\mathrm{m_b}\times\mathrm{n_b}$ blocks, where ideally $\mathrm{m_b}=\mathrm{n_b}$ and both should be a power of 2, and distributed amongst processes. Successful execution of a ScaLAPACK application thus involves 1) initialsing an MPI environment; 2) initialising and configuring the BLACS; 3) distributing the matrix; 4) calling the desired driver routine; 5) collecting the eigenvalues and distributed eigenvectors; 6) destroying the BLACS grid and MPI environment.

Four driver routines for solving the eigenproblem for dense symmetric matrices are included in ScaLAPACK. These are P?YEV, P?SYEVD, P?SYEVX and P?SYEVR, where the '?' character field indicates real (s)ingle or real (d)ouble precision. All four routines consist of the three classic steps outlined in Section 3. Aside from varying implementations of Steps 1 and 3, they all differ in their chosen method of eigen-decomposition of the tridiagonal matrix. The QR-based routine PDSYEV was the first to be developed and has since been superseded be the other three. It will therefore be discussed no further. A brief synopsis of Step 2 for the remaining three routines, along with other salient features, is presented below.

4.1.1 PDSYEVD

The simple driver PDSYEVD calculates all eigenvalues and optionally eigenvectors of a real symmetric matrix. Eigen-decomposition of the tridiagonal matrix uses Tisseur and Dongarra's parallel implementation [26] of Cuppen's divide and conquer method [7], and the back transformation is also developed by Tisseur and J Dongarra [26]. Orthogonality of eigenvectors is ensured using the approach by Gu and Eisenstat [14]. The total memory requirements of PDSYEVD are $4n^2/p + O(n)$ per process (n^2 for the distributed matrix, n^2 for the eigenvectors, and $2n^2$ workspace), where p is the number of processes.

4.1.2 PDSYEVX

The expert driver PDSYEVX computes either all, or a selected subset of eigenvectors and optionally eigenvalues of a real symmetric matrix. Eigenvalues of the tridiagonal matrix are computed by bisection [10], and the subsequent eigenvectors by inverse iteration [18]. The algorithms used for tridiagonal reduction and back transformation are the same as used in PDSYEVD. Memory requirements are $4n^2/p + O(n)$ per process, however, PDSYEVX does not guarantee a correct answer with $O(n^2/p)$ memory per processor. Depending on the number of eigenvalues within a degenerate cluster, reorthogonalisation of the corresponding eigenvectors can, in the worst case, require $O(n^2)$ memory on a single processor to guarantee the correctness of the computed eigenpairs in which case parallelism is lost.

4.1.3 PDSYEVR

The Multiple Relatively Robust Representations (MRRR) based driver PDSYEVR [30, 29] computes either all, or a selected subset of eigenvalues and optionally eigenvectors of a real symmetric matrix. The memory requirements of PDSYEVR for the full spectrum are $4n^2/p+O(n)$ per processer, which is reduced to $(2n^2+3kn)/p$ memory per processor for the computation of a subset of k eigenpairs.

4.2 ELPA

ELPA is a relatively new parallel direct eigensolver for dense symmetric matrices that utilises the same block-cyclic matrix distribution as ScaLAPACK, and can therefore act as a stand-in for ScaLAPACK. Communication between processors is handled by direct calls to a message-passing-interface (MPI) library and consequently the parallel linear algebra routines are completely independent from either BLACS or ScaLAPACK. Only auxilliary linear algebra subroutines that are serial are called from ELPA, and are typically taken from BLAS or LAPACK.

For the standard symmetric eigenvalue problem ELPA follows the same three steps as given in Section 1. Eigendecomposition of the tridiagonal matrix is performed using an efficient implementation of Cuppen's divide and conquer algorithm that has been modified to allow subsets of eigenpairs to be computed at a reduced time cost. Reduction to tridiagonal form, and the subsequent back-transformation to standard form, may be performed using either a one-step (ELPA1) or two-step (ELPA2) procedure. The two components of the two-step procedure may be far better optimised than the usual one-step procedure, leading to an overall saving in compute time, particularly for large matrices, despite the cost of having to perform an additional back-transformation.

Recent updates to the ELPA library to allow hybrid MPI + OMP threading and use of GPU accelerators have been reported in Ref. [20], along with other improvements such as an autotuning feature that provides a black-box method of optimising parameters such as block size and process grid setup. Comparisons of MPI + OMP threading with pure MPI have shown that ELPA currently works optimally in a pure MPI configuration [6]. Similarly, comparisons between CPU-only and CPU + GPU based calculations [19] have shown that use of GPU accelerators result in large speedups in time-to-solution providing the GPUs are saturated, the drawback being that GPUs generally have less RAM than CPUs. Exact memory requirements for ELPA are unknown but are expected to be $4n^2$ based on the requirements of the D&C algorithm.

5 HPC platform and testing environment

5.1 Architecture

All calculations were performed on the CSD3 petascale HPC platform, consisting of the Cumulus (CPU) and Wilkes2 (CPU + GPU hybrid) systems. The Cumulus cluster provides 2.27 petaflops of compute capability and contains 1152 Skylake nodes, each with 2 x Intel Xeon Skylake 6142 processors operating at 2.6 GHz, with either 192 Gb or 384 Gb of RAM. These were used for all the reported CPU calculations. The Wilkes2 cluster provides 1.19 petaflops of compute capability and consists of 360 NVIDIA Tesla P100 GPUs dispersed between 90 Dell EMC server nodes (4 GPUs per node), each node having 96 Gb RAM and connected by Mellanox EDR Infiniband.

5.2 Compilation/runtime environment

5.2.1 CSD3

Our tests on CSD3 invoked the Fortran versions of PDSYEVD, PDSYEVX, PDSYEVR through version 2019.3 of Intel's mpiifort compiler, which was linked to the 64-bit MKL library through

```
-lmkl_scalapack_lp64 -lmkl_blacs_intelmpi_lp64 -lmkl_intel_lp64 -lmkl_sequential -lmkl_core
-lpthread -lm
```

with compiler options

```
-O3 -ipo -xHost
```

ELPA version 2019.05.001 was configured... built... Phil requested this info from Arjen

5.2.2 Wilkes2

ELPA version 2019.05.001 was configured... built... Phil requested this info from Arjen

6 Performance testing

Our experiments were performed on a series of pseudo-randomly generated matrices ranging in size from 100 000 × 100 000 to 500 000 × 500 000. The general matrix structure is illustrated for a small ($n = 4000$) matrix in Fig. 2. It is not meant to closely resemble the structure of the Hamiltonian matrices generated by TROVE; for a pictoral representation of these

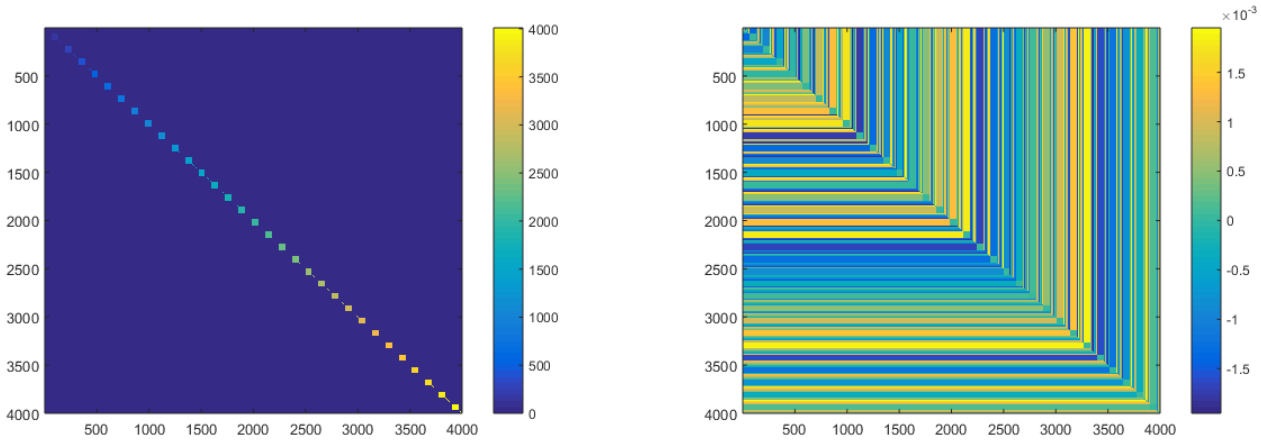

Figure 2: Structure of the pseudo-random matrices used for our tests. Both plots represent the same $n = 4000$ matrix, except on the right plot elements with a value greater than 10.0 have been set to zero.

matrices the reader is directed to Ref. [1]. All matrices generated were double-precision, real, symmetric and shared the same sequence of eigenvalues.

A block size of 64 was used in all calculations. Optimising this value corresponds to finding a compromise between data access contiguity and load-balancing, but we did not attempt this optimisation by hand. Nor did we test the autotuning feature in ELPA, as this procedure may take several iterations to optimise a single parameter, which would be far too expensive given the size of matrices considered. All calculations were performed in a pure MPI configuration, i.e. one process per core, and we did not investigate the effect of OpenMP threading.

From our selection of ScaLAPACK solvers, PDSYEVD was the most reliable and did not crash once providing sufficient workspace was provided. PDSYEVX is known to be unstable, and we had problems with the orthogonality of eigenvectors it produced despite trialling various combinations of values for the `orfac`, `abstol`, and `clustersize` parameters, and providing a workspace array several times larger than the recommended minimum. For this reason results for PDSYEVX are not presented in the following sections. Similar problems were experienced with PDSYEVR if the full spectrum of eigenvalues was requested, although if only the smallest 10% of the spectrum was requested it produced correct results. We did not investigate the relation between the success of the solver and the matrix size, number of processes, number of roots etc.

The ELPA 2-stage solver (ELPA2) was fairly reliable, although we encountered memory errors at a matrix dimension of 600 000 × 600 000 running on 1280 12Gb CPU cores, which occurred irrespective of the number of roots requested. Assuming a workspace requirement of $2n^2$ for the D&C algorithm, plus an additional n^2 for the distributed matrix and n^2 for the eigenvectors, this should be well within the memory specifications of the CSD3 Skylake nodes. Following the stacktrace it appears the crash occurred during an MPI send and receive, when a process received data and tried to copy onto the local memory. We speculate that either the temporary cache was not valid memory, or the region being copied into was not valid.

Problems were also experienced when using the ELPA 1-stage solver (ELPA1), which produced diverging eigenvalues at both ends of the eigenspectrum under certain circumstances. For our 200 000 × 200 000 test matrices, this occurred when 640 MPI processes were utilized, but not for less than this number (i.e. 256, 384, or 512 processes). This issue was reported to the authors of ELPA who could not reproduce the error. Given that the ELPA 2-stage solver is known to be faster than the 1-stage solver, we did not advance our testing of the 1-stage solver to matrices larger than 200 000×200 000.

6.1 Cumulus results

Three series' of benchmark calculations were performed on the Cumulus cluster. The first and second aimed to investigate the performance scaling of each solver for matrices of size 200 000 × 200 000 and 400 000 × 400 000 respectively as the number of MPI processes is increased. The third aimed to push the boundaries of ELPA regarding matrix size given a resource limit of 1280 CPU cores.

The performance scalability of the three solvers for the 200 000 × 200 000 matrix is presented in Fig. 3, and the corresponding maximum memory used per core, as reported by the Slurm workload manager, is shown in Fig. 4. For a matrix this size all routines show scalability up to the maximum of 640 cores. The ELPA 2-stage solver (ELPA2) execution times, when the full spectrum is requested, are roughly half those of PDSYEVD and two-thirds those of PDSYEVR for only 10% of the spectrum (PDSYEVR 10pc). Furthermore, ELPA2 yields a threefold speed increase relative to the full spectrum if only 10% of the spectrum is calculated. Memory requirements for the two D&C based routines were mostly comparable, and often slightly exceeded $4n^2/p$ per core owing to Phil: add explanation. Reducing the number of eigenpairs calculated by ELPA resulted in little, if any, reduction in maximal memory useage as Phil: check D&C memory bottleneck. The MRRR based algorithm PDSYEVR, as expected, required substantially less memory than D&C.

Equivalent measurements for the 400 000×400 000 matrix are presented in Figs. 5 and 6. It was decided not to include PDSYEVR in this set of benchmark measurements due to its four-fold increase in compute times relative to ELPA2 and

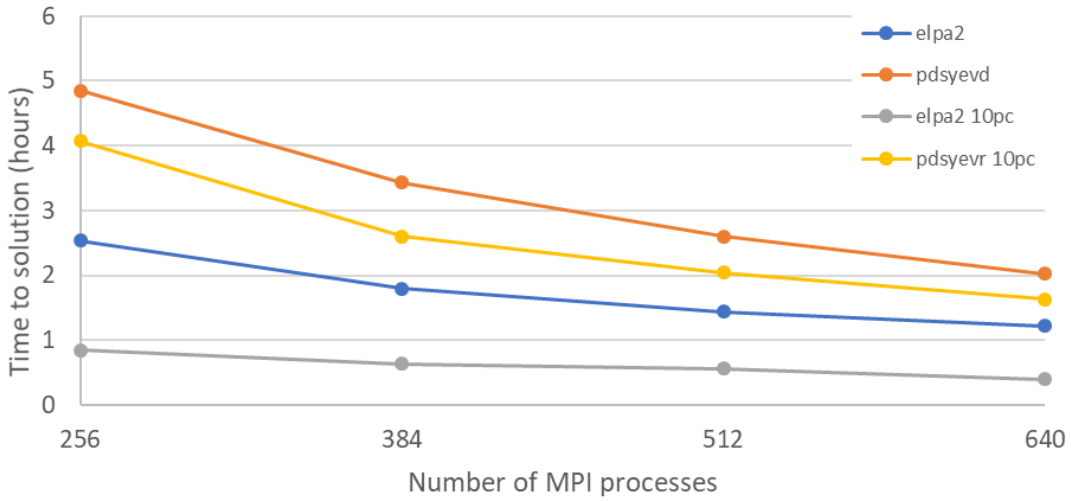

Figure 3: Execution times of each solver for a 200 000 × 200 000 matrix. The full eigenspectrum was computed using ELPA2 (blue) and PDSYEVR (orange), and 10% of the eigenspectrum was computed using ELPA2 (grey) and PDSYEVR (yellow).

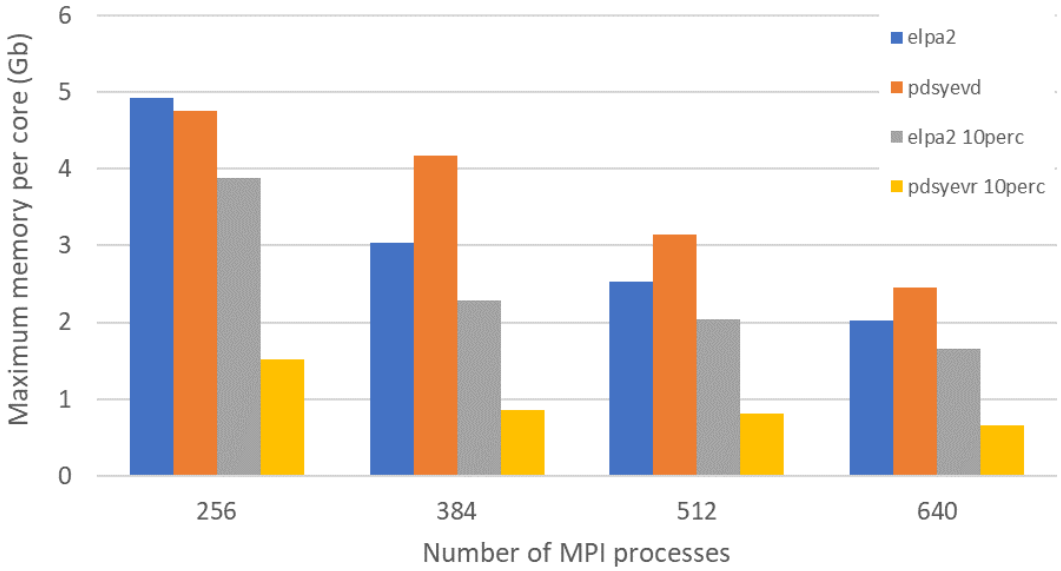

Figure 4: Maximum memory useage of each solver, as reported by Slurm, for a 200 000 × 200 000 matrix. The full eigenspectrum was computed using ELPA2 (blue) and PDSYEVR (orange), and 10% of the eigenspectrum was computed using ELPA2 (grey) and PDSYEVR (yellow).

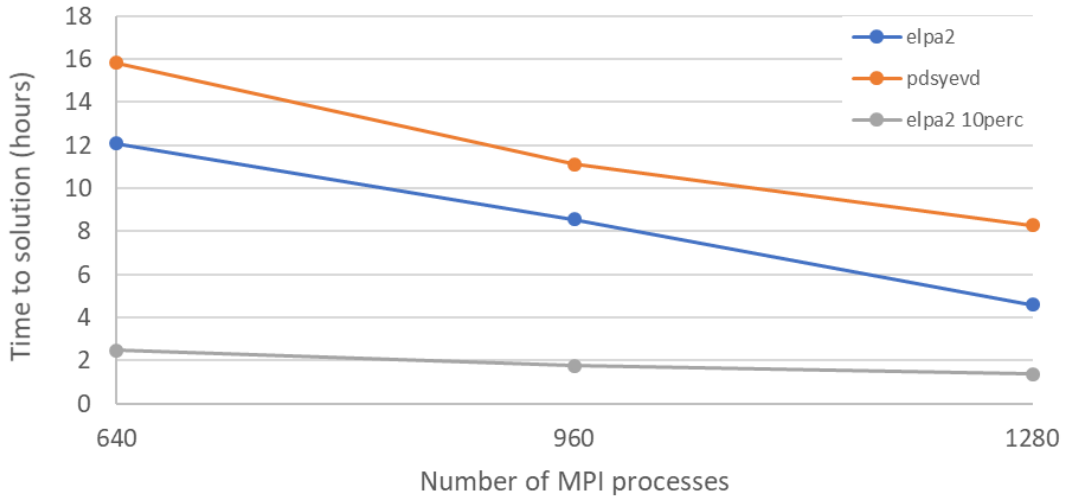

Figure 5: Execution times of each solver for a 400 000 × 400 000 matrix. The full eigenspectrum was computed using ELPA2 (blue) and PDSYEVR (orange), and 10% of the eigenspectrum was computed using ELPA2 (grey) and PDSYEVR (yellow).

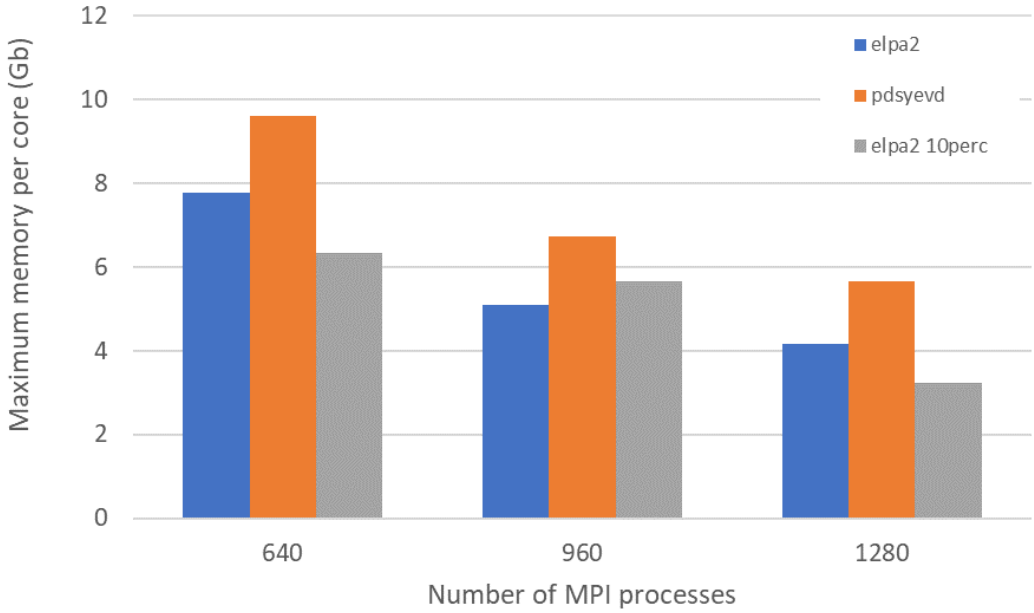

Figure 6: Maximum memory useage of each solver, as reported by Slurm, for a 400 000 × 400 000 matrix. The full eigenspectrum was computed using ELPA2 (blue) and PDSYEVR (orange), and 10% of the eigenspectrum was computed using ELPA2 (grey) and PDSYEVR (yellow).

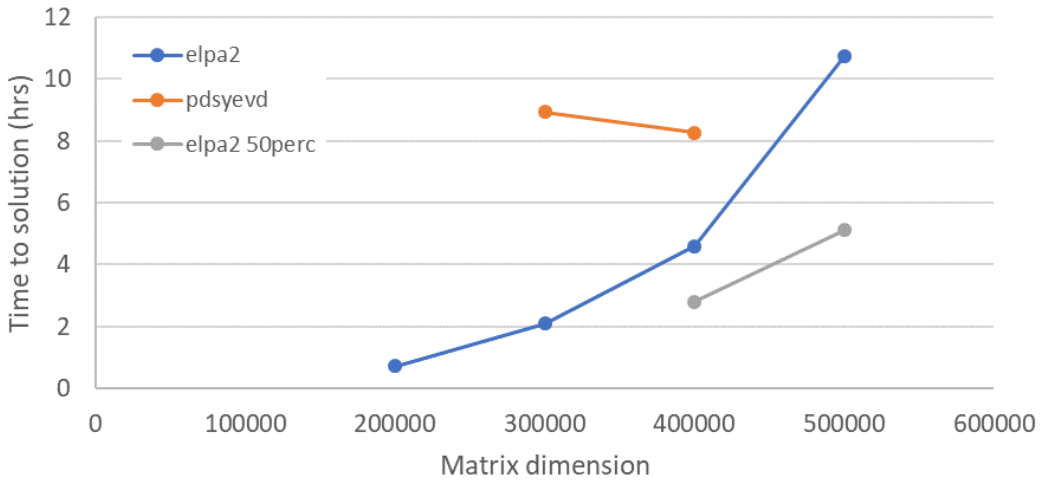

Figure 7: Execution times of each solver using a constant 1280 MPI processes (CPU-cores). ELPA2 (blue) and PDSYEVD (orange) were used to compute the full eigenspectrum. ELPA2 (grey) was used to compute 50% of the eigenspectrum.

the convergence problems observed earlier; we felt that this would be an inefficient use of our limited CPU resources given the success of each calculation was uncertain. All data points shown are the result of only one measurement, except for the ELPA2 execution time using 960 processes which was averaged of two measurements (8.08 and 9.02 real hours). Once again both solvers are scalable within framework tested. ELPA2 was consistently faster than PDSYEVD when computing the entire spectrum, providing most benefit when used with 1280 CPU-cores for which it was roughly a factor of 2× faster than PDSYEVD. This is also the most efficient number of processes in terms of overall CPU-hours required. For calculating only 10% of the eigenspectrum ELPA2 is 6× faster than PDSYEVD is for calculating all roots, although this advantage seems to decrease marginally as the number of cores is increased. Finally, the memory requirements of ELPA2 for this set of benchmark calculations is slightly less than those of PDSYEVD and, as with the Figure 4 comparisons, computation of only a subset of the eigenspectrum yields little, if any, reduction in maximum memory use.

The performance of ELPA2 and PDSYEVD for a series of matrices ranging in size from 200 000 × 200 000 to 500 000 × 500 000, using a constant 1280 CPU-cores (1280 MPI processes), is presented in Fig. 7. Each data point represents one measurement. As mentioned before, we could not succeed in diagonalising a 600 000 × 600 000 matrix using ELPA, even for a subset of the spectrum. Two prior attempts at diagonalising the 300 000 × 300 000 matrix using PDSYEVD failed owing to us allocating insufficient wall clock time (12 hours total for construction and diagonalisation). For this reason we are confident that the PDSYEVD measurements are a true reflection of the solver's performance, and that the 9 hour execution time for 300 000 × 300 000 is due to saturation of the BLACS communications, as ELPA does not rely on BLACS. This behaviour has been previously observed in Refs..., although additional profiling tests would be required to confirm the reason. Once again, for calculating a subset of the eigenspectrum, in this case 50%, the ELPA2 execution times are substantially less than those for the full spectrum. Phil: Can't find how the flops scale for a subset of eigenpairs using the ELPA2 approach anywhere in the literature.

6.2 Wilkes2 results

Phil: Waiting for Arjen to complete arranged runs.

Table 1: Run times of CPU-only and CPU + GPU versions ELPA for a matrix of size 100 000.

	Cumulus	Wilkes2	
	10×Skylake node	10×(Server nodes + 4 Nvidia P100 GPUs)	
	CPU	CPU + GPU	
Solver	Time (s)	Time (s)	Speed-up
ELPA1	1154	372	**3.10**×

6.3 Tesseract results

Phil: Waiting for Arjen to complete arranged runs.

7 Concluding remarks and outcomes

The performance scalability of the Scalapack routines PDSYEVD and PDSYEVR was compared to the ELPA 2-stage solver for matrices ranging from $n = 200\,000$ to $n = 500\,000$, and for varying numbers of CPU-cores. In all measurements ELPA2 outperformed the ScaLAPACK routines. PDSYEVR failed to compute the complete eigenspectrum for any of the test cases, and when used to compute a 10% subset of the eigenspectrum it was slightly faster than PDSYEVD but still slower than the ELPA2 solver calculating the complete spectrum. The speedup obtained by using ELPA2 instead of PDSYEVD, as is currently standard in TROVE, ranged between 1.3× and 6× depending on the matrix size, number of roots required, and number of MPI processes. However, when larger numbers of cores are used (relative to the matrix dimension) than we have tested here, the speedup from using ELPA2 may be substantially more owing to saturation of the BLACS communications.

From this investigation we conclude that implementation of ELPA would be extremely advantageous to the TROVE project, which would benefit from the significant reduction in overall CPU resource consumption. We therefore extended the current suite of applications used to externally read and diagonalise matrices generated by TROVE to include the option of using ELPA. This was relatively easy to do considering ELPA uses the same block-cyclic distribution as ScaLAPACK and requires the same input parameters. An MPI version of TROVE is currently under development and we expect it to eventually include ELPA. Until then, the TROVE procedure for diagonalising the Hamiltonian matrix for large n is as follows: i) use TROVE to save the matrix to disk, ii) use the external routine to read and distribute the matrix between MPI processes, iii) call the desired solver in the external routine, iv) use the external routine to write the eigenvectors to disk, along with some additional information. The eigenvectors and additional information, used by TROVE to determine the quantum numbers of each eigenstate, can then be read back into TROVE in order to continue calculations.

To check the aforementioned procedure results in correct eigenvalues and eigenvectors it was used to reproduce part of the previously published spectrum of AsH_3, which was also computed by TROVE using the LAPACK routine DSYEVD. The intensity of an absorption line depends on the eigenvectors (wavefunctions) of the corresponding upper and lower molecular quantum states, so providing ELPA, PDSYEVD and DSYEVD are set up and functioning correctly the line intensity values should be independent of the choice of solver, as should the quantum numbers that describe the transition. This was indeed the case in our testing, and the line intensities and quantum numbers generated in TROVE using eigenvectors from ELPA2 and PDSYEVD were identical to the published results which used the DSYEVD solver.

The external routine developed during this project that was used to initialise the MPI environment, read/generate and distribute the matrix, set up and call the ELPA/ScaLAPACK, and collect the resulting eigenvectors is available at `https://github.com/PhillipAColes/Diagonalisers`.

References

[1] A. Al-Refaie. *Efficient Production of Hot Molecular Line Lists*. PhD thesis, University College London, 2016. An optional note.

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

[3] G. Baker, J. Gunnels, G. Morrow, B. Riviere, and R. van de Geijn. PLAPACK: high performance through high-level abstraction. In *Proceedings. 1998 International Conference on Parallel Processing (Cat. No.98EX205)*. IEEE Comput. Soc.

[4] P. Bientinesi, I. S. Dhillon, and R. A. van de Geijn. A parallel eigensolver for dense symmetric matrices based on multiple relatively robust representations. *SIAM Journal on Scientific Computing*, 27(1):43–66, 2005.

[5] L. 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. Whaley. Scalapack users' guide. Technical report.

[6] B. Cook, T. Kurth, J. Deslippe, P. Carrier, N. Hill, and N. Wichmann. Eigensolver performance comparison on cray XC systems. *Concurrency and Computation: Practice and Experience*, 2018.

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

[8] J. Demmel and K. Stanley. The performance of finding eigenvalues and eigenvectors of dense symmetric matrices on distributed memory computers. In *In proceedings of the seventh SIAM conference on parallel processing for scientific computing. SIAM*, pages 528–533. SIAM, 1994.

[9] J. W. Demmel. *Applied Numerical Linear Algebra*. Society for Industrial and Applied Mathematics, 1997.

[10] J. W. Demmel and I. Dhillon. On the correctness of some bisection-like parallel eigenvalue algorithms in floating point arithmetic. *Electronic Trans. Num. Anal*, 3:116–149, 1995.

[11] I. S. Dhillon. *A New O(n^2) Algorithm for the Symmetric Tridiagonal Eigenvalue/Eigenvector Problem*. PhD thesis, EECS Department, University of California, Berkeley, 1997.

[12] J. G. F. Francis. The QR transformation a unitary analogue to the LR transformation–part 1. *The Computer Journal*, 4(3):265–271, 1961.

[13] J. G. F. Francis. The QR transformation–part 2. *The Computer Journal*, 4(4):332–345, 1962.

[14] M. Gu and S. C. Eisenstat. A divide-and-conquer algorithm for the symmetric tridiagonal eigenproblem. *SIAM Journal on Matrix Analysis and Applications*, 16(1):172–191, 1995.

[15] I. Gutheil, T. Berg, and J. Grotendorst. Performance analysis of parallel eigensolvers of two libraries on bluegene/p. *Journal of Mathematics and Systems Science*, 2(4):231 – 236, 2012.

[16] I. Gutheil, J. F. Münchhalfen, and J. Grotendorst. Performance of dense eigensolvers on BlueGene/q. In *Parallel Processing and Applied Mathematics*, pages 26–35. Springer Berlin Heidelberg, 2014.

[17] T. Imamura, S. Yamada, and M. Machida. Development of a high-performance eigensolver on a peta-scale next-generation supercomputer system. *Progress in Nuclear Science and Technology*, 2(0):643–650, 2011.

[18] I. C. F. Ipsen. Computing an eigenvector with inverse iteration. *SIAM Review*, 39(2):254–291, 1997.

[19] P. Kůs, H. Lederer, and A. Marek. GPU optimization of large-scale eigenvalue solver. In *Lecture Notes in Computational Science and Engineering*, pages 123–131. Springer International Publishing, 2019.

[20] P. Kůs, A. Marek, S. Köcher, H.-H. Kowalski, C. Carbogno, C. Scheurer, K. Reuter, M. Scheffler, and H. Lederer. Optimizations of the eigensolvers in the ELPA library. *Parallel Computing*, 85:167–177, 2019.

[21] C. Liao. Optimizing PLASMA eigensolver on large shared memory systems. In *2016 7th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA)*. IEEE, 2016.

[22] 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. *Journal of Physics: Condensed Matter*, 26(21):213201, 2014.

[23] T. Mellor, S. Yurchenko, B. Mant, and P. Jensen. Transformation properties under the operations of the molecular symmetry groups g36 and g36(EM) of ethane h_3cch_3. *Symmetry*, 11(7):862, 2019.

[24] J. Poulson, B. Marker, R. A. van de Geijn, J. R. Hammond, and N. A. Romero. Elemental: A new framework for distributed memory dense matrix computations. *ACM Transactions on Mathematical Software*, 39(2):1–24, 2013.

[25] C. Sousa-Silva, A. F. Al-Refaie, J. Tennyson, and S. N. Yurchenko. ExoMol line lists – VII. the rotation–vibration spectrum of phosphine up to 1500 k. *Monthly Notices of the Royal Astronomical Society*, 446(3):2337–2347, 2014.

[26] F. Tisseur and J. Dongarra. A parallel divide and conquer algorithm for the symmetric eigenvalue problem on distributed memory architectures. *SIAM Journal on Scientific Computing*, 20(6):2223–2236, 1999.

[27] S. Tomov, J. Dongarra, and M. Baboulin. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Computing*, 36(5-6):232–240, 2010.

[28] D. S. Underwood, S. N. Yurchenko, J. Tennyson, A. F. Al-Refaie, S. Clausen, and A. Fateev. ExoMol molecular line lists – XVII. the rotation–vibration spectrum of hot SO3. *Monthly Notices of the Royal Astronomical Society*, 462(4):4300–4313, 2016.

[29] C. Vömel. LAPACK working note 195: ScLAPACK's MRRR algorithm. Technical report, Institute of Computational Science, ETH Zürich, 2008.

[30] C. Vömel and D. Antonelli. LAPACK working note 168: ScaLAPACK's parallel MRRR algorithm for the symmetric eigenvalue problem. Technical report, Computer Science Division, University of California, 2005.

[31] P. R. Willems and B. Lang. A framework for the MR^3 algorithm: Theory and implementation. *SIAM Journal on Scientific Computing*, 35(2):A740–A766, 2013.

[32] S. N. Yurchenko, W. Thiel, and P. Jensen. Theoretical ROVibrational energies (TROVE): A robust numerical approach to the calculation of rovibrational energies for polyatomic molecules. *Journal of Molecular Spectroscopy*, 245(2):126–140, 2007.