Eigensolver Performance Comparison on Cray XC Systems

Brandon Cook, Thorsten Kurth, Jack Deslippe
NERSC
Lawrence Berkeley National Laboratory
Berkeley, CA 94720, USA
Email: (bgcook,tkruth,jrdeslippe)@lbl.gov

Pierre Carrier, Nick Hill, Nathan Wichmann
Cray Inc.
Bloomington, MN 55425, USA
Email: (nhill,pcarrier,wichmann)@cray.com

Abstract—**Hermitian (symmetric) eigenvalue solvers are the core constituents of electronic structure, quantum-chemistry and other HPC applications, such as Quantum ESPRESSO, VASP, CP2K, NWChem, to name a few. Our understanding of the performance of symmetric eigenvalue algorithms on various hardware is clearly important to the quantum chemistry or condensed matter physics community, but in fact goes beyond that community. For instance, big data analytics is increasingly utilizing eigenvalues solvers, in the study of randomized singular value decomposition (SVD), or principal component analysis (PCA). Noise, vibration, and harshness (NVH) is another field where fast and efficient eigenvalue solvers are required. Most eigenvalue solver packages feature numerous different parameters which can be tuned for performance, e.g. the number of nodes, number of total ranks, the decomposition of the matrix, etc.. In this paper, we investigate the performance of different packages as well as the influence of these knobs on the solver performance.**

***Keywords*-eigensolver; cray; ELPA; ScaLAPACK; performance;**

I. Introduction

Dense hermitian (symmetric) eigensolvers are critical for performance in a number of common HPC workloads. In quantum chemistry and condensed matter physics, electronic structure applications such as Quantum ESPRESSO, VASP, CP2K, and NWChem repeatedly compute eigenvalues and eigenvectors in order to determine a self-consistent electron densities. In material-science applications focused on excited states (e.g. ABINIT[1], Yambo[2], BerkeleyGW[3]) dense eigenvalue problems can reach large matrix dimensions on the order of many 100's of thousands because both the occupied and unoccupied electron orbitals and energies must be computed. In addition, big data analytics is increasingly utilizing eigenvalue solvers in the study of randomized singular value decomposition (SVD), or principal component analysis (PCA). Noise, vibration, and harshness (NVH) is another field where fast and efficient eigenvalue solvers are required.

While iterative approaches are often employed when a small faction of the eigenvalues or eigenvectors are required, direct solvers (as implemented in standard libraries) typically out perform iterative approaches when the entire space of eigenvectors/eigenvalues is required - as is the case in the applications listed above. However, there is significant diversity of implementations even within the direct eigensolver space. Eigenvalue solver packages feature numerous different parameters, especially with respect to matrix decomposition, which can have significant performance impacts.

In addition, the emergence of energy-efficient computer architectures in Cray systems like the Cori (Xeon-Phi powered) supercomputer at NERSC motivates questions on how different distributed eigensolvers perform on these novel architectures, if the optimal library parameters need to be adjusted and how these libraries perform when running in a Hybrid MPI-OpenMP mode. On Cori for example, applications are often motivated to use a significant number of OpenMP threads per MPI rank because of memory constraints of performance considerations - it is important to know if the distributed eigensolvers used in some of these applications perform adequately in this scenario.

In this paper we investigate, empirically, the performance of both the main incumbent library in this space, ScaLAPACK [4], as well as the newer ELPA (Eigenvalue SoLvers for Petaflop Applications) library [5]-[6] on a range of Cray systems with different node-architectures. We show performance scales with matrix size, node-count as well as MPI-OpenMP tradeoff. The use of both libraries additionally requires a number of parameter choices relative to the distribution of the matrix across MPI ranks. We investigate the impact of suboptimal choice of parameters and describe efficient methods for choosing optimal parameters. Finally we extract guidelines for when trade-offs must be made due to other application constraints such as MPI vs OpenMP in hybrid codes.

II. Background

A. Dense symmetric eigen-problems

Numerically the task is to find λ and $\mathbf{c}$ such that

$$\mathbf{A}\mathbf{c} = \mathbf{B}\mathbf{c}\lambda$$

where $\mathbf{A}$ and $\mathbf{B}$ are $N \times N$ matrices, $\mathbf{c}$ is an $N \times N$ matrix of eigenvectors and λ is a diagonal matrix containing the eigenvalues.

In the case where only a few eigenpairs are needed an iterative method is typically very efficient. However in many cases the full spectrum is desired [6]. The direct solution for all eigenvector, eigenvalue pairs scales as $O(N^3)$ which in addition to memory requirements motivates research for efficient parallel solvers.

B. Parallel Block cyclic distributions

Both ELPA and ScaLAPACK utilize block cyclic decompositions for dense linear algebra routines. Arrays are split into many small blocks of $(n_b \times n_b)$ and processes are arranged in a two-dimensional grid such that $n = n_x \times n_y$ where n is the number of MPI ranks. The small blocks are then distributed cyclically on the process grid. The size of the blocks and process grid then become important parameters in a solver using such a decomposition. More detailed discussion of the block cyclic distribution used by both solvers is available [4].

III. Methodology

For the purposes of examining the performance of both `PZHEEVD` (SCALAPACK) and `ELPA`, we developed a standalone benchmark capable of generating and diagonalizing Hermitian matrices of arbitrary dimension across different MPI and OpenMP configurations and block-cyclic layouts. The standalone benchmark supports diagonalization with both `PZHEEVD` and `ELPA`.

A. Input matrix

The input matrix is built in both PZHEEVD and ELPA codes based on the same random seeds that generates a common matrix, using the `zlarnv` routine. ELPA transposes the generated upper triangle matrix, while SCALAPACK does not store the lower triangle of the matrix:

```
numEle = ml*nl
call zlarnv(1, iseed, numEle, a_ref)
call transpose_matrix(”L”, ...)
```

We only consider matrices larger than $N = 10000$ as node local solvers such as LAPACK are more appropriate for smaller problems, and as is commonly required in many applications we utilize complex numbers for the matrix elements.

Source codes and compilation instructions for the ELPA and PZHEEVD benchmark drivers, which provide identical input matrices, are shown in Appendices A, B and C.

B. ELPA

Eigenvalue SoLvers for Petaflop-Applications (ELPA) offers a variety of options in terms of specific algorithms used. In our case we are using an *ELPA 2stage* solver. The specific mathematics of these algorithms is not the focus of this paper and details can be found elsewhere [5], [6].

Table I
CPU/ System Architectures

Architecture	System	Sockets per node	Cores per node
HSW	Cori	2	32
KNL	Cori	1	68
BDW	crystal	2	36
SKX	horizon	2	40

C. ScaLAPACK

The ScaLAPACK routine `PZHEEVD` uses a divide and conquer algorithm. The algorithm has natural parallelism as the initial problem is partitioned into several subproblems that can be solved independently [7]. This algorithm is widely used since it offers a very stable and efficient parallel solution to Hermitian matrices used in a wide variety of applications. The `cray-libsci` library, as well as the Intel MKL library, include and optimize this ScaLAPACK routine. For the purposes of this paper, we consider the `cray-libsci` implementation.

D. System Architecture

The Cray XC series (XC40 or XC50) offers the combined advantages of the Aries interconnect [8] and Dragonfly network topology, Intel Xeon multi-core and Intel© Xeon Phi™ many-core processors, delivering up to 100 PF sustained system performance. It is designed for production supercomputing and user productivity.

Haswell and Xeon Phi (KNL) results collected on Cori a Cray XC40 sited at NERSC at Lawrence Berkeley National Laboratory. Broadwell-18 and Skylake-20 results were collected on crystal (a Cray XC40) and horizon (a Cray XC50). Those 2 systems are a mixture of various Intel processors, and offer up to 926 and 378 compute nodes, respectively on crystal and horizon, sited locally at Cray Inc. The evaluated architectures are summarized in Table I.

IV. Parameters

The 2-dimensional block decomposition of data in both solvers in controlled primarily by two parameters: n_b, the blocksize, and (n_x, n_y) which is the processes grid subject to the constraint $n_x \times n_y = n$, where n is the total number of MPI ranks.

One may also choose the number of nodes and number of MPI ranks per node (or number of OpenMP threads per rank). These parameters may have additional constraints in real applications which may rely on specific numbers of MPI ranks or OpenMP threads, or require a minimum number of nodes due to memory requirements. Additionally one may have access to a variety of compute platforms with different computational characteristics. For example: Cori at NERSC contains both Haswell and Xeon Phi nodes. Given the variety and range of valid parameters one may have to choose from several thousand valid combinations of (M, r, n_x, n_b), where M is the number of nodes, r is MPI ranks per node, n_x

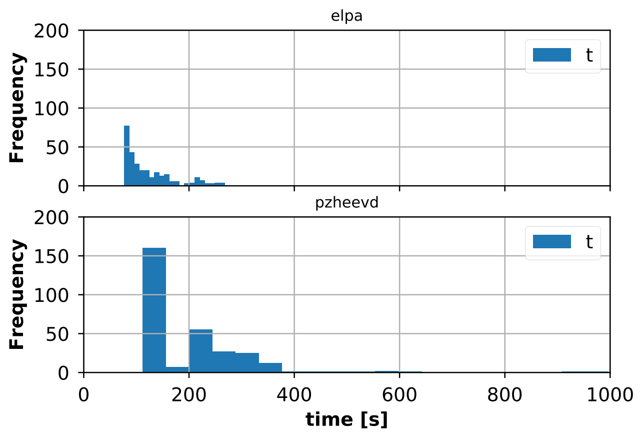

Figure 1. Histogram of solution times for $N = 20000$ on 4 Xeon Phi nodes of Cori for range valid combinations of (n_x, n_y), ranks per node, $n_b \in [1, 100]$ for ELPA and ScaLAPACK

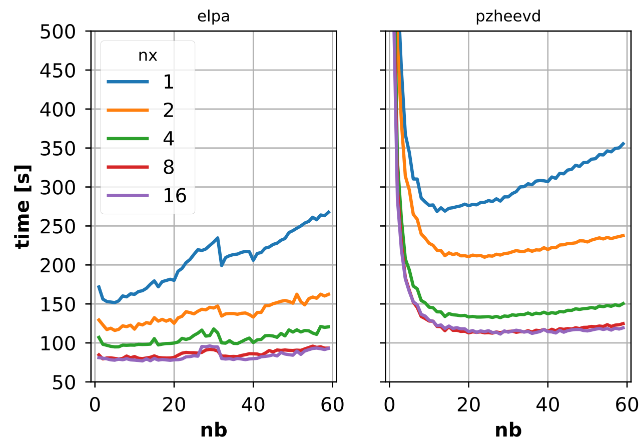

Figure 3. Time to solution for $N = 20000$ matrix on 4 KNL nodes with 64 ranks per node.

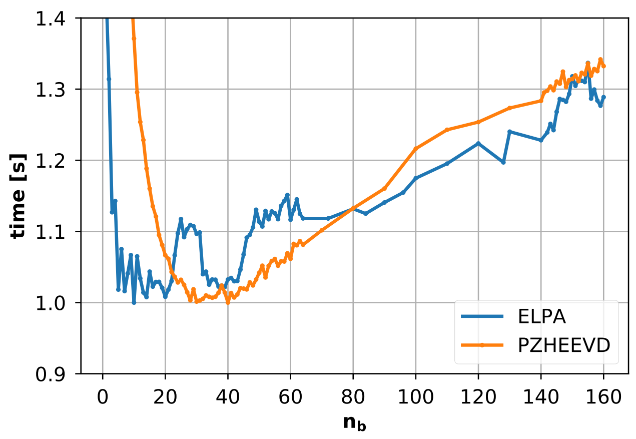

Figure 2. ELPA time as a function of n_b, for $(n_x, n_y) = (10, 32)$ on SKX with $N = 40000$.

determines the process grid and n_b is the block size. Figure 1 illustrates the potential range of performance by showing a histogram of solution times for a full sweep of the valid parameter space for a $N = 20000$ matrix on 4 Xeon Phi nodes. Note that for some choices the performance could be 10x worse than the optimal case.

A. Block size

The block size of the decomposition is one of the most straightforward parameters to vary. Typical advice for this parameter is either anecdotal or "it depends" [4]. However, in the case of ELPA an internal blocksize is used for operations, but the n_b parameter is still important for parallel load balancing and efficiency.

Figure 2 shows the n_b landscape for a matrix of size $N = 40,000$, as a function of the parameter n_b. This graph is provided as a reminder that the time-to-solution minimization landscape is not a smooth function of any of the parameters- n_b in this example. Generally the choice of n_b is a tradeoff between good load balancing with very small n_b and higher level BLAS for local operations with larger n_b.

Figure 3 shows that ELPA and ScaLAPACK have a very different dependence on n_b. Generally the optimal block size is between 1 and 200, with 1 almost never being a good solution. For ELPA there may be multiple minima, but they are generally similar in performance. For ScaLAPACK small block sizes $n_b <= 4$ are not desirable and result in significantly worse performance while in ELPA there is a smaller penalty for a suboptimal choice of n_b, likely due to the additional internal blocking which ELPA performs.

B. Process Grid

The process grid should be chosen such that $n_x \times n_y = n$, where n is the total number of MPI ranks.

Figure 4 shows that the main MPI communication patterns are a function of the n_x and n_y parameters, as one departs from the optimum solution. The optimum that corresponds to the lowest time is on the left, for $n_x = 6$ (i.e., not $n_x = 12$). The goal of optimization process is essentially to eliminate the `MPI_Wait` (brown blocks) and `MPI_Allreduce` (orange blocks) times. As those MPI times increase, the ratio of the main computation `hh_trafo_complex_kernel_12_AVX_1hv_double`, to the total time diminishes. Noticeably, the `MPI_Bcast` improves as the total compute time increases. Therefore, the Perftools profile seems to indicate that the $n_x \times n_y$ is an interplay between `MPI_Bcast` and `MPI_Allreduce` + `MPI_Wait`. We observe that n_b correlates to `MPI_Bcast` to a lesser extent than (n_x, n_y).

C. Optimization

For a given number of OpenMP threads per rank, number of nodes, and matrix size there are essentially only 2 param-

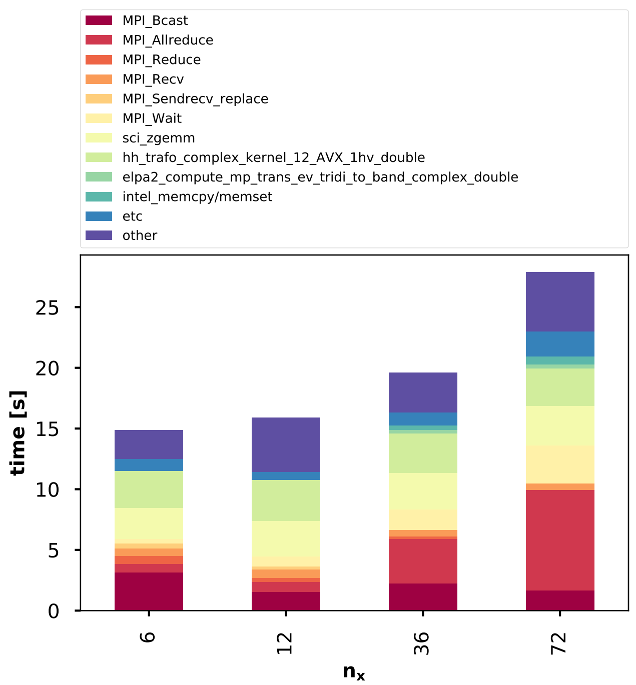

Figure 4. Influence of n_x and n_y on the MPI communication. The fastest time is on the left, with $n_x = 6$. ELPA times vary as 14.85, 15.92, 19.6, and 27.9, with n_x that varies as 6, 12, 36, and 72, in the graph.

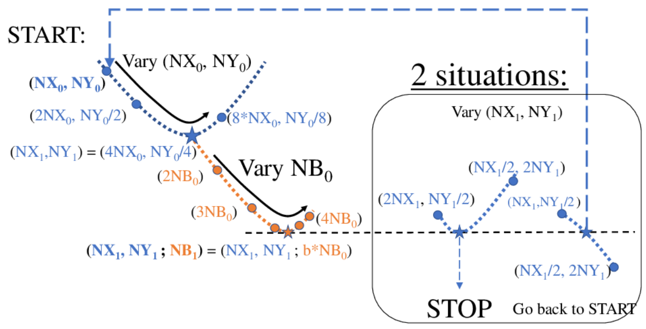

Figure 5. Manual optimization

eters that need to be minimized for a given number of MPI ranks: the matrix grid that needs to satisfy MPI_comm_size = $n_x \times n_y$, and the block size, n_b.

Figure 2 shows the actual landscape for a matrix of size 40,000 $\times$ 40,000, as a function of the parameter n_b. The minimization is obviously not as clean and perfect as shown in Fig. 5.

1) Manual: Optimization can be performed manually, in just few steps. Figure 5 shows the general approach for finding the fastest ELPA (or PZHEEVD) compute time, which is essentially a minimization procedure where the variable that needs to be minimized is the total compute time. It is a simple and efficient approach for finding the minimum of 2 sets of variables (n_x depends on n_y, through MPI_comm_size), corresponding to the fastest solution. The iteration starts at the upper left part of the figure, where one chooses 2 guess values for n_x and n_y that satisfy MPI_comm_size = $n_x \times n_y$. An initial guess is also chosen for n_b (e.g., the number of cores per node on the hardware is a good choice).Then the pair (n_x, n_y) is multiplied/divided by 2, or any factors that gives an integer value of MPI_comm_size. A faster runtime combination of (n_x, n_y) corresponds to the local optimum value. The pair (n_x^1, n_y^1) is then fixed, while n_b now is set to vary. The same process is repeated for n_b, until a faster runtime is found, with n_b^1. Two runs are done at this n_b^1, surrounding the local minimum, using $(2n_x, n_y/2)$ and $(n_x/2, 2n_y)$, as shown in the box at the bottom-right. Two possibilities can occur, either the minimum was found, or that there is a new value for the pair (n_x, n_y) that can lead to a faster time. If that is the case, then the entire procedure is repeated, until no more variations are observed. In practice that process is rarely repeated.

The method is sequential, since the choice of the next parameters depend on the outcome of the current run. On the other hand, in practice, a global minimum can be found in most case, in less then 10 steps of Fig. 5. However, the procedure does not guarantee that the global minimum is found, especially when multiple minima are shown to exist, as shown in Table II, or shown in the landscape of the parameter n_b Figure 2. This is true even when using random searches, although random search might help jump out of a local minimum (which is of crucial importance for the physics of energy minimization problems, but less crucial here, since we are only trying to find the fastest time-to-solution).

2) Bayesian Parameter Optimization: There is a vast amount of literature on Bayesian (hyper-)parameter estimation. Most algorithms were developed to efficiently perform high dimensional parameter space searches in order to optimize a cost (or maximize an objective) function. For our studies we will use the spearmint software package [9] which was developed to find good hyper parameters of neural networks. However, the package can be used to perform arbitrary parameter searches as long as a good objective function is defined. The idea behind spearmint is that it uses Gaussian Process Regression on a sequence of pairs of parameter guesses and objective function values to estimate the next step in this sequence by maximizing the expected improvement over the current best value for the objective function. Spearmint can perform concurrent searches in which it will estimate an expected improvement based on observed as well as queued data points. It achieves this by interpreting the outstanding expected best improvements as a multivariate Gaussian distribution from which it can sample in order to generate a next best guess (cf. [10] for details).

For our experiments we use the execution time of the diagonalization algorithm as the objective function. We

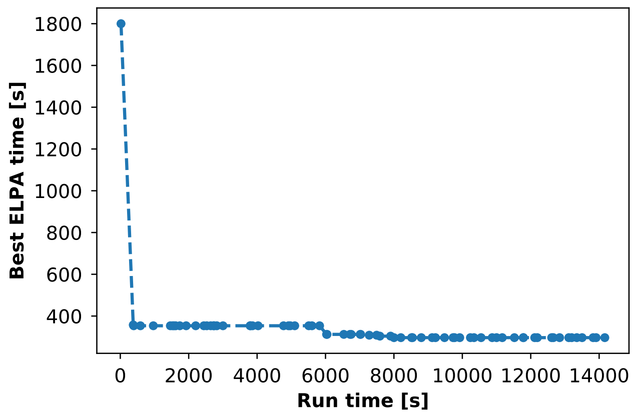

Figure 6. Optimization with Spearmint. As spearmint is designed to run multiple experiments in parallel we show the current best benchmark time as a function of wall clock time for the optimization job

fix the number of nodes and use spearmint to perform a parameter search within these bounds, i.e. the unconstrained parameters are the number of ranks per node, the number of row-ranks (n_x) and the number of blocks. The number of column ranks (n_y) will be computed from the total number of ranks and the number of row-ranks and the block size (n_b) is varied. The efficiency of spearmint for finding good solutions is show in Figure 6.

V. Discussion

Table III shows the general trend that for a given matrix size there are essentially no correlation between (n_x, n_y) and n_b as the size of the matrix is increased. For example, Figure 7 shows the scaling of the optimum values of n_x, n_y, and n_b, for various sizes of matrices. Table III shows the optimum values used in the Fig. 7. Considering the 8 node case, and varying the dimension of the matrix, one finds that as N increases from 10000 to 60000, in increments of 10000, the optimum values of n_x, n_y, and n_b do not follow any precise trends. For example, as N=10000, 20000, 30000, 40000, 50000, 60000; n_x=10, 8, 20, 10, 8, and 20; n_y=32, 40, 16, 32, 40, 16; n_b=20, 20, 10, 5, 20, 10. The conclusion is that often simulations are started using a smaller problem size, with the goal of increasing the accuracy of the total energy, say, by increasing the cutoff energy of the plane wave basis, or doubling the number of atoms, etc., a procedure typical in VASP or quantum espresso FFT-based code simulations. The set of optimum (n_x, n_y) and n_b values for that smaller problem can certainly be taken as an initial guess for the larger problem, however, in practice, that larger size simulation will require a new set of n_x, n_y and n_b in order to be optimum.

In comparison to ELPA, the PZHEEVD parameters seem to show a more regular variation with the matrix size (n_x is more or less constant as the matrix size increases). Therefore, PZHEEVD has the advantage to not require that

Table II
Choice of parameters n_x, n_y, for $N = 40000$, $n_b = 2$ (ELPA) and $n_b = 70$ (PZHEEVD), using 36 cores/node on 4 Broadwell nodes ($n_b = 2$ and $n_b = 70$ correspond respectively to the ELPA and PZHEEVD optimum values for $n_x = 12$, $n_y = 12$)

n_x	n_y	ELPA time	PZHEEVD time
1	144	739.10	1503.82
2	72	625.05	1109.87
3	48	600.12	989.70
4	36	579.62	914.85
6	24	**544.87**	887.56
8	18	552.19	839.23
9	16	579.21	**819.93**
12	12	**546.52**	888.01
16	9	600.16	**783.72**
18	8	**581.08**	866.30
24	6	599.62	979.10
36	4	602.15	1075.34
48	3	850.71	1168.19
72	2	1019.38	1369.65
144	1	930.02	1798.52

Table III
Data corresponding to Figure 7, for ELPA. We also have extra dimensions of the matrix N included in the table. All runs are using `OMP_NUM_THREADS=1`. Runs are on Skylake using AVX512

N	nodes	MPI	n_x	n_y	n_b	ELPA time
10000	1	40	5	8	10	20.43
10000	2	80	10	8	20	11.51
10000	4	160	10	16	20	7.18
10000	8	320	10	32	20	5.31
20000	1	40	5	8	10	145.58
20000	2	80	8	10	10	77.32
20000	4	160	8	20	20	43.60
20000	8	320	8	40	20	29.09
20000	16	640	8	80	10	21.22
30000	1	40	5	8	20	468.49
30000	2	80	10	8	10	245.28
30000	4	160	10	16	20	132.19
30000	8	320	20	16	10	76.12
30000	16	640	20	32	10	47.44
30000	32	1280	20	64	5	35.74
40000	1	40	5	8	20	1090.04
40000	2	80	10	8	20	566.55
40000	4	160	10	16	20	299.41
40000	8	320	10	32	5	175.48
40000	16	640	20	32	10	100.30
40000	32	1280	40	32	10	69.33
50000	1	40	5	8	10	2142.55
50000	2	80	4	20	20	1113.33
50000	4	160	8	20	10	570.83
50000	8	320	8	40	20	324.59
50000	16	640	8	80	20	200.34
50000	32	1280	32	40	10	144.74
50000	64	2560	128	20	5	92.68
60000	2	80	5	16	20	1852.40
60000	4	160	10	16	10	963.72
60000	8	320	20	16	10	522.03
60000	16	640	20	32	5	299.50
60000	32	1280	40	32	10	187.67
60000	64	2560	80	32	10	136.36

Table IV
DATA CORRESPONDING TO FIGURE 7, FOR PZHEEVD. ALL RUNS ARE USING OMP_NUM_THREADS=1. RUNS ARE ON SKYLAKE-20 USING AVX512

N	nodes	MPI	n_x	n_y	n_b	PZHEEVD time
10000	1	40	8	5	40	31.59
10000	2	80	5	16	40	17.25
10000	4	160	10	16	40	9.79
10000	8	320	10	32	40	6.88
20000	1	40	8	5	40	264.11
20000	2	80	16	5	40	131.92
20000	4	160	16	10	40	69.22
20000	8	320	16	20	50	40.02
20000	16	640	8	80	30	24.10
30000	1	40	8	5	70	878.59
30000	2	80	8	10	60	457.92
30000	4	160	32	5	40	223.03
30000	8	320	32	10	40	122.62
30000	16	640	32	20	40	74.56
30000	32	1280	16	80	40	46.60
40000	1	40	8	5	60	2085.62
40000	2	80	16	5	40	1035.70
40000	4	160	32	5	40	534.76
40000	8	320	32	10	40	286.01
40000	16	640	32	20	50	166.77
40000	32	1280	16	80	40	97.38

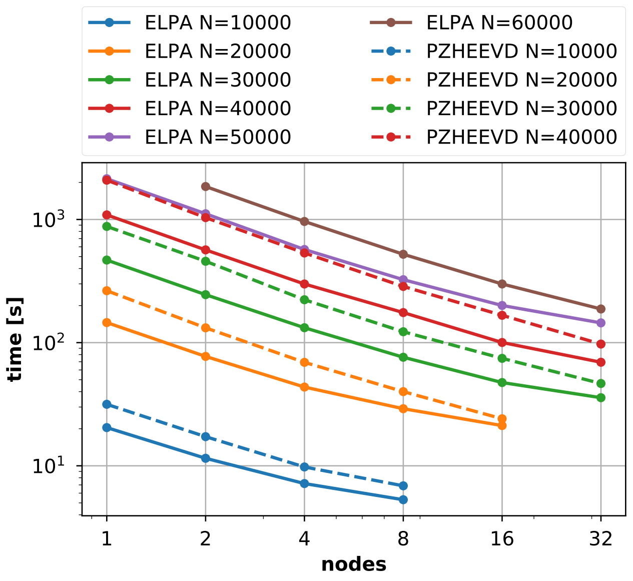

Figure 7. Comparison of scaling between PZHEEVD and ELPA, for N=10000, 20000, 30000, 40000 on SKX. The (n_x, n_y) and n_b values were optimized manually.

much optimization as the problem size (or when increasing the energy cutoff in plane wave codes) is increased.

The optimum block size, NB, is in general much larger in PZHEEVD than in ELPA. For instance, the average value that NB takes in ELPA is 10, while in PZHEEVD, it is 40.

Table II shows an interesting case on broadwell-18 cores, using 4 nodes, which corresponds to 144 MPI ranks, easily divisible by $2^4 3^2$. We show all the possible combinations of n_x for ELPA and PZHEEVD. Note that n_x and n_y are not symmetric. In other words, the runtime completed using the pair (n_x, n_y) does not equates that using the reversed pair (n_y, n_x). The important observation from this set of computations is that having $n_x \times n_y = 12^2$, is essentially a good option, it is however not necessarily the only best option in both algorithms. For instance, the pair $(n_x, n_y) = (6, 24)$ gives a slightly better performance than $(n_x, n_y) = (12, 12)$ with ELPA, while $(n_x, n_y) = (16, 9)$ is clearly the optimum for PZHEEVD. Note that there are 3 local minima in ELPA, shown in bold in the Table [excluding the extreme case of $(n_x, n_y) = (144, 1)$], and 2 local minima in PHZEEVD, although one is much lower than the other. Table II also indicates a feature less apparent with PZHEEVD, that there may be multiple optima to consider. This feature was also observed in the case of the parameter n_b, shown in Figure 2.

In summary, the optimum values that ELPA requires is based on these two general considerations: (a) The optimum value that n_x and n_y should take are not too far from each others. (b) The pair (n_x, n_y) is not symmetric. For example, if (n_x, n_y) is fast, consider swapping the number, (n_y, n_x), and verify that it is still an optimum. (c) If the optimum parameter, n_b, from SCALAPACK is known, consider reducing its value by a factor of 10, when running with ELPA. (d) The n_x parameter increases more or less with the number of nodes. There are no exact rules that can dictate exactly how to match the number of nodes to n_x, but that general trend is still valid.

A. Strong Scaling

Figure 7 shows graphically the scaling data from Tables III and IV, in a log-log scale. The parameters in this figure were obtained via manual optimization show in Figure 5. Overall, the two methods scale with equivalent ratio. The ELPA method is approximately 1.5 times faster than than PZHEEVD, although this ratio is not constant and depends on the size of the problem. In our testing ELPA was always faster than PZHEEVD.

B. OpenMP vs MPI

Figure 9 shows the performance of openMP threading for the 2 codes, ELPA and PZHEEVD from ScaLAPACK. One advantage of ELPA is that threading performs better than SCALAPACK, at least up to 4 threads. This is very important for codes such as quantum espresso, where threading has been optimized (for example, within the computation of the exact-exchange).

C. Architecture comparison

To compare architectures we focus on a $N = 40000$ problem size on 4 nodes of a given platform. Table V shows the best combination of parameters for each solver on Skylake, Broadwell, Haswell and Xeon Phi. Xeon Phi

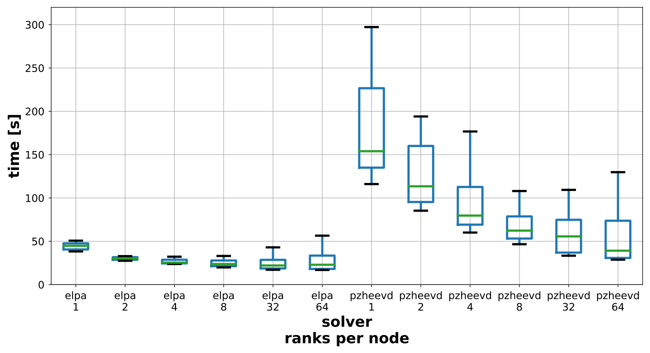

Figure 8. Range of solution time for ELPA and PZHEEVD on 4 KNL nodes for different ranks per node. The matrix is $N = 10000$. All valid combinations of (n_x, n_y) were run for each (solver, ranks per node) combination. The blocksize was varied in the range of $n_b \in [1, 60]$.

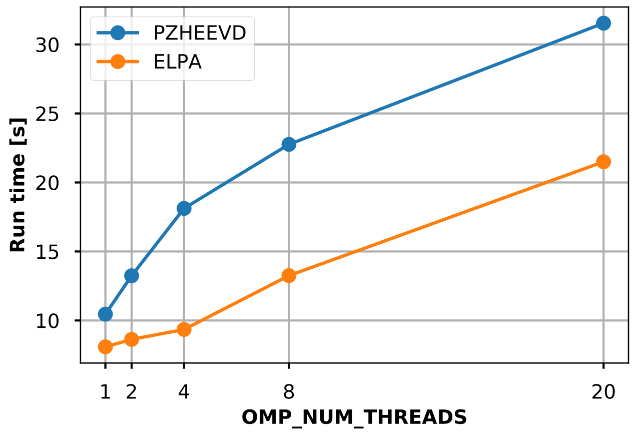

Figure 9. Performance variation for a matrix N=10000 on 4 SKX nodes.

outperforms both Haswell and Broadwell with optimal parameters with ELPA, but is slower than Skylake. The same trend is continued for ScaLAPACK. Though one interesting thing to note is that the performance gain of ELPA over ScaLAPACK is smallest on Xeon Phi. The choice of block size on each (platform, solver) combination fits with the previously discussed guidelines. One interesting finding was that with this specific problem and node count the hybrid OpenMP/MPI version with 2 OpenMP threads on Haswell outperformed the pure MPI version. This may be due to cache effects and the better parallel load balancing with $n_x = n_y$.

VI. Conclusion

We have examined the performance of ELPA and ScaLAPACK for complex eigenvalue problems on a range of Cray XC platforms with different architectures. We provide methods for choosing optimal parameters and discuss the tradeoffs involved. We have found that the choice of non-optimal parameters can result in a 2x or worse slowdown

Table V
Choice of parameters n_x, n_y, for $N = 40000$ on different architectures)

Architecture	solver	MPI	(n_x, n_y)	n_b	t[s]
HSW	ELPA	128	(8,16)	16	611.84
HSW	PZHEEVD	64	(8,8)	50	929.18
KNL	ELPA	256	(8,32)	36	501.79
KNL	PZHEEVD	256	(8,32)	50	633.58
BDW	ELPA	144	(6,24)	2	544.87
SKX	ELPA	160	(10,16)	20	299.41
SKX	PZHEEVD	160	(32,5)	40	534.76

when compared to optimal parameters. ELPA is particularly sensitive to parameters like the block-size in the block-cyclic distribution, where the performance of the diagonalization varies in a non-smooth way with this parameter choice. We also note that the optimal parameter choice is architecture dependent with Intel's Xeon-Phi processors generally outperforming dual-socket Haswell and Broadwell nodes. Dual socket Skylake nodes generally showed the best performance, with ELPA providing close to a 2x performance advantage over SCALAPACK.

Despite the extra sensitivity, we found that ELPA outperforms SCALAPACK on all architectures tested and all matrix sizes and concurrencies. In addition, ELPA performs significantly better than SCALAPACK in scenarios where multiple OpenMP threads are used for each MPI rank. The performance gap between ELPA and SCALAPACK widens as function of threads used. While, ELPA performance is still generally optimal in a pure MPI configuration, improved OpenMP scaling is important when using these libraries in applications that implement a hybrid MPI-OpenMP approach because of other memory of performance considerations.

Given the significant use of dense diagonalization in a number of important HPC applications and the potential penalty of poor parameter choice, the strategy presented can help make more efficient use of systems and show the comparison of architectures for parallel dense linear algebra.

Acknowledgment

The authors would like to thank Cray Inc for generous allocation of computing time on the Cray internal machines, crystal (XC40), and horizon (XC50). This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

References

[1] X. Gonze, B. Amadon, P.-M. Anglade, J.-M. Beuken, F. Bottin, P. Boulanger, F. Bruneval, D. Caliste, R. Caracas, M. Ct, T. Deutsch, L. Genovese, P. Ghosez, M. Giantomassi, S. Goedecker, D. Hamann, P. Hermet, F. Jollet, G. Jomard, S. Leroux, M. Mancini, S. Mazevet, M. Oliveira, G. Onida, Y. Pouillon, T. Rangel, G.-M. Rignanese,

D. Sangalli, R. Shaltaf, M. Torrent, M. Verstraete, G. Zerah, and J. Zwanziger, "Abinit: First-principles approach to material and nanosystem properties," *Computer Physics Communications*, vol. 180, no. 12, pp. 2582 – 2615, 2009, 40 {YEARS} {OF} CPC: A celebratory issue focused on quality software for high performance, grid and novel computing architectures. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0010465509002276

[2] A. Marini, C. Hogan, M. Grüning, and D. Varsano, "Yambo: an ab initio tool for excited state calculations," *Computer Physics Communications*, vol. 180, no. 8, pp. 1392–1403, 2009.

[3] J. Deslippe, G. Samsonidze, D. A. Strubbe, M. Jain, M. L. Cohen, and S. G. Louie, "Berkeleygw: A massively parallel computer package for the calculation of the quasiparticle and optical properties of materials and nanostructures," *Computer Physics Communications*, vol. 183, no. 6, pp. 1269–1289, 2012.

[4] 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*. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1997.

[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 Computing*, vol. 37, no. 12, pp. 783–794, 2011.

[6] 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*, vol. 26, no. 21, p. 213201, 2014.

[7] F. Tisseur and J. Dongarra, "A Parallel Divide and Conquer Algorithm for the Symmetric Eigenvalue Problem on Distributed Memory Architectures," *SIAM J. Sci. Comput. 6:20*, 1999.

[8] B. Alverson, E. Froese, L. Kaplan, and D. Roweth, "Cray XC Series Network," https://www.cray.com/sites/default/files/resources/CrayXCNetwork.pdf.

[9] https://github.com/HIPS/Spearmint.

[10] J. Snoek, H. Larochelle, and R. P. Adams, "Practical Bayesian Optimization of Machine Learning Algorithms," *ArXiv e-prints*, Jun. 2012.

APPENDIX A. ELPA BENCHMARK

```
module helper
implicit none
contains
subroutine transpose_matrix(uplo, ln, a_ref, a_sym, desca)

 character, intent(in)  :: uplo
 integer, intent(in) :: ln
 complex(kind=8), intent(in) :: a_ref(:,:)
 complex(kind=8), intent(out) :: a_sym(:,:)
 integer, intent(in) :: desca(:)
 complex(kind=8) :: alpha, beta
 logical :: lsame

 alpha = (1.0, 0.0)
 beta = (0.0, 0.0)

 call pztranc(ln, ln, alpha, a_ref, 1, 1, desca, &
 & beta, a_sym, 1, 1, desca)

 if (lsame(uplo, "L")) then
   call pztrmr2d("L", "N", ln, ln, a_ref, 1, 1, &
   & desca, a_sym, 1, 1, desca, desca(2))
 else
   call pztrmr2d("U", "N", ln, ln, a_ref, 1, 1, &
   & desca, a_sym, 1, 1, desca, desca(2))
 end if

end subroutine transpose_matrix

elemental subroutine convert_to_int(str, i, stat)
 character(len=*), intent(in) :: str
 integer, intent(out) :: i, stat
 read(str, *, iostat=stat) i
end subroutine convert_to_int

end module helper

program benchmark_elpa
use elpa
use elpa_driver
use helper

implicit none
include 'mpif.h'

integer :: ctxt_sys
integer :: rank, size, i, j
integer :: ln, lnprow, lnpcol, lnbrow, lnbcol
integer :: llda, ml, nl, lsize, numEle
integer :: myrow, mycol, info, err
real(kind=8) :: t1, t2

class(elpa_t), pointer :: e

integer, allocatable :: desca(:)
complex(kind=8), allocatable :: a_ref(:,:), a_sym(:,:), z(:,:)
real(kind=8), allocatable :: w(:)
integer, dimension(4) :: iseed
character(len=10), dimension(4) :: argc
integer numroc, mpi_comm_rows, mpi_comm_cols
integer, parameter :: ELS_TO_PRINT  = 5

! Initialize MPI and BLACS
call mpi_init(err)
call mpi_comm_rank(MPI_COMM_WORLD, rank, err)
call mpi_comm_size(MPI_COMM_WORLD, size, err)

do i = 1, 4
   call get_command_argument(i, argc(i))
end do

call convert_to_int(argc(1), ln, err)
call convert_to_int(argc(2), lnprow, err)
call convert_to_int(argc(3), lnpcol, err)
call convert_to_int(argc(4), lnbrow, err)
lnbcol = lnbrow
if (rank == 0) then
   print *, "ln =", ln
   print *, "lnprow =", lnprow
   print *, "lnpcol =", lnpcol
   print *, "lnbropw =", lnbrow
end if

call blacs_get(0, 0, ctxt_sys)
call blacs_gridinit(ctxt_sys, "C", lnprow, lnpcol)
call blacs_gridinfo(ctxt_sys, lnprow, lnpcol, &
  & myrow, mycol)

if (myrow .eq. -1) then
   print *, "Failed to properly initialize
             MPI and/or BLACS!"
   call MPI_FINALIZE(err)
   stop
end if

! Explicitly get and set the row and column
! communicators, as the API seems to be
! failing to initialize them as they should
err = elpa_get_communicators(MPI_COMM_WORLD,
  & myrow, mycol, &
  & mpi_comm_rows, mpi_comm_cols)


! Allocate my matrices now
ml = numroc(ln, lnbrow, myrow, 0, lnprow)
nl = numroc(ln, lnbcol, mycol, 0, lnpcol)
llda = ml

allocate(a_ref(ml,nl))
allocate(a_sym(ml,nl))
allocate(z(ml,nl))
allocate(w(ln))
allocate(desca(9))

! Create  blacs descriptor for transposing  matrix
call descinit(desca, ln, ln, lnbrow, lnbcol, &
  & 0, 0, ctxt_sys, llda, info)

iseed(1) = myrow
iseed(2) = mycol
iseed(3) = mycol + myrow*lnpcol
iseed(4) = 1
if (iand(iseed(4), 2) == 0) then
   iseed(4) = iseed(4) + 1
end if

! Try initializing and allocating elpa
if (elpa_init(20170403) /= elpa_ok) then
   print *, "ELPA API not supported"
   stop
end if

e => elpa_allocate()

numEle = ml*nl
call zlarnv(1, iseed, numEle, a_ref)
call transpose_matrix("L", ln, a_ref, a_sym, desca)

deallocate(a_ref)
t1 = MPI_WTIME()

e => elpa_allocate()
call e%set("na", ln, err)
call e%set("nev", ln, err)
call e%set("local_nrows", ml, err)
call e%set("local_ncols", nl, err)
call e%set("nblk", lnbrow, err)
call e%set("mpi_comm_parent", mpi_comm_world, err)
call e%set("mpi_comm_rows", mpi_comm_rows, err)
call e%set("mpi_comm_cols", mpi_comm_cols, err)
call e%set("process_row", myrow, err)
call e%set("process_col", mycol, err)
call e%set("solver", ELPA_SOLVER_2STAGE, err)
call e%set("complex_kernel", &
  & ELPA_2STAGE_COMPLEX_AVX512_BLOCK1, err)
call e%eigenvectors(a_sym, w, z, err)
call elpa_deallocate(e)
call elpa_uninit()

t2 = MPI_WTIME()

if (rank == 0) then
   print *, "ELPA time: ", t2 - t1
end if

! deallocate stuff

if (rank == 0 ) then
   do j=1, ELS_TO_PRINT
      write(6,*) "w(", j, ")=", w(j), "z(", j, ")=", z(j,1)
   end do
end if

deallocate(a_sym)
deallocate(z)
deallocate(w)
deallocate(desca)

! Finish MPI
call MPI_FINALIZE(err)

end program benchmark_elpa
```

The following is the code for the pzheevd LAPACK calls.

APPENDIX B. ScaLAPACK BENCHMARK

```
module helper
implicit none
contains

elemental subroutine convert_to_int(str, i, stat)
character(len=*), intent(in) :: str
integer, intent(out) :: i, stat

read(str, *, iostat=stat) i

end subroutine convert_to_int

end module helper

program pzheevd_test
use mpi
use helper

implicit none

integer, parameter :: MY_NPROW      = 2
integer, parameter :: MY_NPCOL      = 2
integer, parameter :: MY_N          = 1024
integer, parameter :: MY_NB         = 256
integer, parameter :: MY_IL         = 1
integer, parameter :: MY_IU         = 1024
integer, parameter :: ELS_TO_PRINT  = 5

integer   :: info = 0
character :: jobz = 'V'
character :: uplo = 'L'
integer   :: ln = MY_N
integer   :: lnbrow = MY_NB
integer   :: lnbcol
integer   :: m, nz
integer   :: err
integer   :: il = MY_IL, iu = MY_IU
real(kind=8) :: dummyL, dummyU
real(kind=8) :: t1, t2
integer :: lnprow = MY_NPROW
integer :: lnpcol = MY_NPCOL
integer :: myrow, mycol
integer :: ia=1, ja=1, iz=1, jz=1
integer :: desca(15), descz(15)
integer :: ctxt_sys
integer :: moneI = -1, zeroI = 0, oneI = 1
integer :: rank, size, i
integer :: llda
character(len=9) :: procOrder = "Row-major"
character(len=10), dimension(4) :: argc

!  ADDED DEFINITIONS
integer :: ml, nl
integer :: iseed(4)
integer :: numEle
complex(kind=8), allocatable :: a_ref(:,:), z(:,:)
real(kind=8), allocatable :: w(:)
complex(kind=8), allocatable ::  work(:)
real(kind=8), allocatable :: rwork(:)
integer,      allocatable :: iwork(:)
real(kind=8) :: temp(2), rtemp(2)
integer :: liwork, lwork, lrwork
integer :: my_rank, j, ierror

!  .. External Functions ..
integer, external ::   numroc

! Initialize MPI and BLACS
call MPI_INIT(err)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, err)
call MPI_COMM_SIZE(MPI_COMM_WORLD, size, err)

do i = 1, 4
   call get_command_argument(i, argc(i))
end do

call convert_to_int(argc(1), ln, err)
call convert_to_int(argc(2), lnprow, err)
call convert_to_int(argc(3), lnpcol, err)
call convert_to_int(argc(4), lnbrow, err)
lnbcol = lnbrow

if (rank == 0) then
  print *, "ln =", ln
  print *, "lnprow =", lnprow
  print *, "lnpcol =", lnpcol
  print *, "lnbropw =", lnbrow
end if

call blacs_get(moneI, zeroI, ctxt_sys)
call blacs_gridinit(ctxt_sys, procOrder, lnprow, lnpcol)
call blacs_gridinfo(ctxt_sys, lnprow, lnpcol, myrow, mycol)

if (myrow .eq. -1) then
   print *, "Failed to initialize MPI and/or BLACS!"
   call MPI_FINALIZE(err)
   stop
end if

! Allocate my matrices now
ml = numroc(ln, lnbrow, myrow, zeroI, lnprow)
nl = numroc(ln, lnbrow, mycol, zeroI, lnpcol)
llda = ml

call descinit(desca, ln, ln, lnbrow, lnbrow, &
& zeroI, zeroI, ctxt_sys, llda, info)
call descinit(descz, ln, ln, lnbrow, lnbrow, &
& zeroI, zeroI, ctxt_sys, llda, info)

allocate( a_ref(ml,nl) )
allocate( z(ml,nl) )
allocate( w(ln) )

iseed(1) = myrow
iseed(2) = mycol
iseed(3) = mycol + myrow*lnpcol
iseed(4) = 1
if (iand(iseed(4), 2) == 0) then
   iseed(4) = iseed(4) + 1
end if
numEle = ml*nl

call zlarnv(oneI, iseed, numEle, a_ref)
call zlarnv(oneI, iseed, numEle, z)

t1 = MPI_WTIME()

call pzheevd(jobz, uplo, ln, a_ref, ia, ja, &
& desca, w, z, iz, jz, descz, temp, moneI, &
& rtemp, moneI, liwork, moneI, info)
lwork     = temp(1)
allocate( work(lwork) )
lrwork    = rtemp(1)
allocate( rwork(lrwork) )
allocate( iwork(liwork) )
call pzheevd(jobz, uplo, ln, a_ref, ia, &
& ja, desca, w, z, iz, jz, descz, work, &
& lwork, rwork, lrwork, iwork, liwork, info)
if (info /= 0) then
   write(6,*) "PZHEEVD returned non-zero info val of ", info
end if

t2 = MPI_WTIME()

if (rank == 0) then
   print *, "PZHEEVD time: ", t2 - t1
end if

call MPI_Comm_rank(MPI_COMM_WORLD, my_rank, ierror)
call blacs_gridexit(ctxt_sys)
call blacs_exit(zeroI)

!  if (rank == 0 ) then
!     do j=1, ELS_TO_PRINT
!        write(6,*) "w(", j, ")=", w(j), "z(", j, ")=", z(j,1)
!     end do
!  end if

deallocate( a_ref, z, w )
deallocate( work, rwork, iwork )

end program pzheevd_test
```

APPENDIX C. COMPILATION

Codes were compiled using Intel compilers version 17.0.2.174.

```
#!/bin/bash

export CRAYPE_LINK_TYPE=dynamic
module swap PrgEnv-cray PrgEnv-intel
module swap intel intel/17.0.2.174

export base_dir=`pwd`

# pzheevd:
ftn -O2 -openmp -o pzheevd benchmark_pzheevd.f90

# elpa:
mkdir ELPA
cd ELPA/
mkdir build
mkdir tmp
export TMPDIR=$base_dir/ELPA/tmp
wget http://elpa.mpcdf.mpg.de/html/Releases/...
2017.05.001.rc2/elpa-2017.05.001.rc2.tar.gz
tar -xvf elpa-2017.05.001.rc2.tar.gz
cd elpa-2017.05.001.rc2
./configure --prefix=$base_dir/ELPA/build \
  --enable-openmp --disable-timings \
  --disable-mpi-module --enable-legacy \
  --enable-avx512 --host=x86_64 \
  --enable-single-precision CC=cc FC=ftn \
  CXX=CC CFLAGS="-fPIC -std=c99 -O3" \
  FCFLAGS="-fPIC -O3" FCLIBS=" "
make
make install
cd ../..
export FFLAGS="-I$base_dir/ELPA/build/include/...
elpa_openmp-2017.05.001.rc2/modules"
export LDFLAGS="-L$base_dir/ELPA/build/lib -lelpa_openmp"
ftn -O2 -openmp ${FFLAGS} benchmark_elpa.f90 -o elpa ${LDFLAGS}
```