On the Diagonalization of large Hamiltonian Matrices within AUTOSTRUCTURE.

N. R. Badnell

Department of Physics, University of Strathclyde, Glasgow G4 0NG, UK

1 Background

We discuss diagonalization of large Hamiltonian (i.e. real symmetric) matrices within AUTOSTRUCTURE (AS), where we require all of the e-vectors as well as all the e-values, not just those associated with a 'few' low-lying states. Such a situation arises with its application to large photoabsorption calculations for Solar opacity (Delahaye *et al* 2021). Currently, AS has no explicit capability for parallel diagonalization (e.g. via scaLAPACK). The parallel version of AUTOSTRUCTURE (PAS) is parallel in the angular momentum of a collision process, only.

Serial diagonalization should still make use of the BLAS/LAPACK library, it being an N^3 problem.

The key routines are DSYEVD and DSYEVR.

However, unless there is really no alternative..., you do not want to compile the libraries yourself. Proprietary optimized libraries are much faster, typically by a factor 3 or 4. And, they are also (likely) parallelized using OpenMP, which can bring another factor of 3 or 4 speed-up.

I have investigated the use of the optimized libraries of Intel, Oracle and Nvidia/PGI.
(For gfortran, the Intel Math Kernel Library (MKL) is readily usable and recommended. All comments pertaining to the Intel MKL also apply to its use by gfortran.)

This covers all free broad-based Fortran compilers (IBM only works on its hardware and LLVM flang is still immature.)

The NAG compiler suite is not free for personal use, but in any case recommends use of the Intel MKL again for BLAS/LAPACK. Thus, we do not consider it further.

2 Details

2.1 OMP_NUM_THREADS

It is important to know if 'multi-threading' is switched-on by default, or not, for any given library. Note, the literature refers to threads but in reality the usual BIOS setting is one thread per core. So we are really talking about cores, but the relevant OpenMP parallelization parameter is OMP_NUM_THREADS.

If the OMP_NUM_THREADS environment variable is *not* set then Oracle defaults the number of threads to 1, but Intel & Nvidia/PGI default to the number of threads=cores available, 20 in my case, which may not be advisable.

I have used (csh)

setenv OMP_NUM_THREADS 12

which pretty much saturates the problem being considered, and compiled my test program, test_dsyev.f, with -O3.

Note, multi-threading can also be switched-on via compiler directives. Needless to say, such directives are compiler dependent. Specifying OMP_NUM_THREADS is compiler independent.

Multi-threading also affects the time usage reported. The Fortran intrinsic subroutine cpu_time reports the processor time (in seconds) with repeated calls defining time intervals.
Intel actually reports a sum of processor times over the threads. Effectively, an interval is the sum of all intervals. The usual time is roughly the cpu_time divided by the number of threads.

The OpenMP library function omp_get_wtime reports the wallclock time, which is a more meaningful quantity for a general code like AS. Its use has been implemented in AS v29.18 via a new utility_time

module. It is used by default in the LAPACK version.
Both Oracle & Nvidia/PGI report identical times from cpu_time and omp_get_wtime. As such, no change to the default use of cpu_time by utility_time is needed.

Finally, OpenMP directives (in proprietary optimized libraries) should automatically be disabled when an MPI parallel environment is detected. But it might be safer to explicitly set OMP_NUM_THREADS to 1 when running PAS (with pasdeck_lap.f95).

2.2 DSYEVD

DSYEVD uses a real vector workspace of dimension LWORK=2*N*N (+6*N+1). Thus N=32766 gives LWORK .lt. 2^{31} while N=32767 gives LWORK .gt. 2^{31} So, compile your source code '-i8' and all is well since these libraries are all 64-bit integer? No! They come with 32-bit interfaces. The Oracle and Nvidia/PGI libraries cannot handle a user input integer .ge. 2^{31}... Only Intel works as expected, with '-i8' (changing your source code INTEGER assignment alone is not enough.)
With Oracle, I can directly access the underlying 64-bit top-level routine DSYEVD_64. This moves the failure lower down the calling chain, where I cannot access the 64-bit routines directly.
So, we have a hard limit of N=32766 for using DSYEVD with Oracle and Nvidia/PGI optimized libraries, unless there is a way to bypass the 32-bit interface (if that is indeed the source of the problem.)
Compiling DSYEVD and all its dependencies from source with '-i8' is easy and works 'fine', but it is not recommended timewise — see the opening remark — and is the last resort, better to install the Intel MKL and use ifort or gfortran.

WARNING: if you compile AUTOSTRUCTURE **with '-i8' it will break the link with the post-processors. You will need to compile them '-i8' as well since (unformatted) passing files use the machine default precision for integers. This enables you to set the standard integer precision to 8 (via SP in module PRECSN) for large cases without the need to maintain two versions of the post-processors. No passing integer ever needs such (SP=8). Instead, compile AS with -c but without -i8 and then link the resulting object file with -i8 to the MKL (ifort -qmkl).**

2.3 DSYEVR

DSYEVR only uses a real matrix workspace of dimension NxN (compared to 2*N*N of DSYEVD) and is just as fast. All arguments of DSYEVR are explicitly 32-bit safe. Of course, Fortran stores the matrix effectively as a vector, of length N*N. Again, N=46340 gives N*N .lt. 2^{31} and N=46341 gives N*N .gt. 2^{31}. AS will have already required you to compile with SP=8 by the time N .ge. 46341. There may still be related 'i8' issues with DSYEVR, as there were with DSYEVD, but none have been seen with the Intel MKL to-date.

But..., the numerical algorithm used by DSYEVR generates NaNs and infinities as part of its operation. It thus requires that the compiled code handles such floating point exceptions 'gracefully', by which we mean it conforms to the IEEE standard default which is that floating point exceptions are not 'trapped' (in the jargon) and execution continues. As one might expect, each compiler has its own default set of traps. Nvidia/PGI and Oracle both trap NaNs and infinities. However, Intel (MKL) conforms to the IEEE standard. As such, Intel works fine for DSYEVR but Nvidia/PGI and Oracle need workarounds.

3 Usage & Timings

So, now is a good time to learn the flags to use for your compiler to switch-off trapping:

Intel: None needed (-fpe3 is the default)

gfortran: None needed (-ffpe-trap=list, where list is empty by default)

Nvidia/PGI: -Kieee

(there is also -Ktrap=none, but the former is both necessary and sufficient for our purposes.)

Oracle: -ftrap=%none

DSYEVR calls function ILAENV which is just a wrapper to call function IEEECK (amongst others). If ILAENV finds that NaNs/infinities cannot be handled gracefully (IEEECK=0) then it returns ILAENV=0, and DSYEVR *silently* takes a much slower approach (than DSYEVD) since it only has N*N workspace. Thus, AS (and test_dsyev) also checks ILAENV and explicitly warns the user and recommends use of DSYEVD instead. . . .

Both Oracle and Nvidia/PGI LAPACK libraries route DSYEVR down the slow road. This would appear to be the end of the matter since there is likely no way to change the mind of the pre-compiled library — we don't know the compiler flags used or what might have been 'hard-wired'.

However, having all the source code, I investigated how these two compilers interacted with IEEECK and ILAENV, as well as Intel & gfortran for timing comparisons:

3.1 Intel

(Plus any usual flags e.g. -m64 -mcmodel=medium -O3)

ifort test_dsyev.f -o test_dsyev.xi -qmkl

i.e. nothing special, just the link to the MKL.
Note, use -Tf asdeck.f95 when compiling, e.g. AS, since ifort does not recognize the .f95 extension.

E.g. I ran N=32,000 with DSYEVR in 22.4 mins with 12 threads (19.5 mins with 20 threads) and got the same (test case) e-values as DSYEVD found in 19.6 mins (13.0 mins with 20 threads).

I also ran N=50,000 with DSYEVR in 81 mins with 12 threads (73 mins with 20 threads) and got the same (test case) e-values whether I compiled with '-i8', or not, the former being a tad slower (5%)

3.1.1 gfortran+MKL

(Plus any usual flags e.g. -m64 -mcmodel=medium -O3)

gfortran test_dsyev.f -o test_dsyev.xg -lmkl_gf_lp64 -lmkl_gnu_thread -lmkl_core -lgomp

i.e. nothing special, when linking to the MKL (which is more verbose than for the native Intel ifort.)

E.g. I ran N=32,000 with DSYEVR in 22.9 mins with 12 threads (20.8 mins with 20 threads) and got the same (test case) e-values as DSYEVD found in 19.7 mins (13.0 mins with 20 threads).

Not surprisingly, the timings are close to those of Intel ifort — it's all about the MKL.

3.2 Nvidia/PGI

(Plus any usual flags e.g. -m64 -mcmodel=medium -O3)

pgf95 -Kieee test_dsyev.f ilaenv.f ieeeck.f -o test_dsyev.xpg -lblas -llapack

enables DSYEVR to take the fast route, and the results agree with DSYEVD.

E.g. I ran N=32,000 with DSYEVR in 24.3 mins with 12 threads (21.1 mins with 20 threads) and got the same (test case) e-values as DSYEVD found in 20.8 mins (17.2 with 20 threads).

3.3 Oracle

(Plus any usual flags e.g. -m64 -xmodel=medium -O3)

f95 -ftrap=%none test_dsyev.f ilaenv.f dsyevr.f -o test_dsyev.x -library=sunperf

does not need ieeeck.f but adding write statements to ilaenv.f shows that the library DSYEVR does *not* call ILAENV, which is the first executable statement in the dsyevr.f source code! Including its source enables DSYEVR to take the fast route, and the results agree with DSYEVD.

E.g. I ran N=32,000 with DSYEVR in 44.6 mins with 12 threads (41 mins with 20 threads) and got the same (test case) e-values as DSYEVD found in 31.1 mins (27.5 mins with 20 threads).

The $\sim 30\%$ speed-up between using Nvidia/PGI and Oracle is also typical of non-threaded large-scale AS calculations. Note that the timings for DSYEVD vs DSYEVR are close for Nvidia/PGI (especially for 12 threads) but DSYEVR is somewhat slower than DSYEVD with Oracle.

4 Summary

For matrices of rank N .le. 32766 use DSYEVD, ideally from a Proprietary optimized library. This works straight out-of-the-box and is the default for the BLAS/LAPACK enabled version of AS.

For matrices of rank N .ge. 32767 use DSYEVR, ideally from a Proprietary optimized library. Currently, only the Intel MKL works straight out-of-the-box ('-i8' is not necessarily required), of those that I have tested so-far. For others (Oracle & Nvidia/PGI), error trapping may (will) need to be disabled. And ilaenv.f & ieeeck.f (and dsyevr.f, Oracle) may (will) need to be compiled from source still when linking to a Proprietary optimized library. This is to avoid DSYEVR taking a slow path using DSTEBZ/DSTEIN. AS warns the user (to screen) if this is happening. If it does still, even with the above, new workarounds are needed.

The BLAS/LAPACK version of AS v29.17 onwards now allows the user to select the diagonalization routine to be used at runtime via e.g. DDIAG='DSYEVR' in SMINIM. (Previously it required un/commenting/out the appropriate source code.) The default is DDIAG='DSYEVD', i.e. unchanged.

We repeat: multi-threading with a Proprietary optimized library will be a factor 10 or more faster than a single-thread 'serial' code compiled from source.

5 Outstanding Issues

Is there a way to use DSYEVD for N .ge. 32767 with Oracle & Nvidia/PGI optimized libraries?

To Do (mid term)

For large-scale problems for which only a 'small' subset of e-energies and possibly e-vectors are needed for a few rates, the Davidson (1975) algorithm should be implemented.

To Do (long term)?

Currently, there is no apparent need to implement (MPI) parallel diagonalization with scaLAPACK, which is non-trivial. If Davidson is not applicable then the calculation of the large number of rates almost certainly dominates the problem. Note also the increased memory requirement associated with MPI. Diagonalization is just one part of a much larger AS memory requirement.

Work Arounds (Physics)

If the diagonalization in AS cannot be achieved/is taking too long, with *current resources,* switch-off mixing between configurations (KUTLS= -1 in SALGEB). Mixing within each Condon & Shortley nl-configuration is retained still. This (KUTLS= -1) repartitions the problem in terms of config.Jp (instead of just Jp). Thus, the Hamiltonians to be diagonalized are much smaller. Note, KUTLS .ge. 0 is possible but this does *not* repartition. It just zeroes-out interactions between configurations .gt. KUTLS, as such mixing maybe unphysical for high-n configurations. The size of the Hamiltonian is unchanged. Combining the two partitioning schemes is non-trivial.
Alternatively, if you want to retain mixing between configurations, then the number of configurations must be reduced.

6 Observations on Rate Calculations

The calculation of autoionization and radiative rates (and photoionization cross sections) is another N^3 problem: viz. multiplication of the unmixed interaction matrix by the initial- and final-state mixing matrices — the e-vectors just obtained by Hamiltonian diagonalization. For argument, we assume that all vectors are of length N and so we consider NxN matrices.

The relevant BLAS routines are DGEMM, DGEMV and DDOT. Again, those from optimized libraries contain OpenMP directives and they scale well (linearly) with the number of threads, provided that the problem is large enough.

On the face of it, one simply applies the matrix multiplication routine DGEMM twice (leaving aside the fact that the final rate is symmetric in the initial and final states...) However, on an organisational level, to handle large problems AS stores this information in long vectors and not necessarily contiguously as subsequently accessed, although ideally it should. Currently, all rate calculations (we include photoionization in this) make use of DDOT which simply multiplies two vectors together.

If we use DDOT N times then we are mimicking a single usage of DGEMV (symmetry aside) — matrix-vector multiplication — but the CPU time overhead/inefficiency of using DDOT repeatedly, as opposed to DGEMV once, means that the latter is faster but even by $N = 30000$ it is 'only' a factor of 3. Also, if it is the final matrix multiplaction then DDOT can take advantage of symmetry and reduce the time difference by a factor of 2 by being applied only $N^2/2$ times.

Similarly, if we use DGEMV N times then we are mimicking a single usage of DGEMM — matrix-matrix multiplication — and again the overhead/efficiency means that the latter is faster, significantly so now, typically by a factor of ~ 10.

Since autoionization rates are contained within a single symmetry group (but partitioned in B-B, B-C and C-C blocks, for multiple C), it may be possible to implement DGEMM and so obtain a significant speed-up. However, calculation of the radiative rates likely dominates.

Radiative rates are a little more problematic — not only do they connect different symmetry groups, it is highly advantageous to compute transitions to all lower states from a given upper state. It may be possible to implement DGEMV for the lower mixing transformation of the (already) upper mixed interaction vector. However, the speed-up of DGEMV usage over DDOT looks likely to be modest.

7 References

Davidson E R, J.Comput.Phys. 17, 87 (1975)
Delahaye F, Ballance C P, Smyth R T and Badnell N R, MNRAS 508, 421 (2021)

Appendix

Weblinks to the compilers discussed:

Intel (Linux, Mac OS, Windows) Recommended (Free for personal use)
https://software.intel.com/en-us/fortran-compilers

gfortran (Linux, Mac OS, Windows) Recommended (Free)
http://gcc.gnu.org/wiki/GFortranBinaries

Nvidia/PGI (Linux, Mac OS, Windows) (Free for personal use)
https://developer.nvidia.com/nvidia-hpc-sdk-downloads

Oracle (Linux) (Free for personal and commercial use)
https://www.oracle.com/tools/developerstudio/downloads/developer-studio-jsp.html