Technical Report on e-CSE project QSGW

Martin Lüders[1], Mark van Schilfgaarde[2]

[1] STFC, Daresbury Laboratory
[2] Kings College London

October 12, 2016

Abstract

Quasi-particle selfconsistent GW (QS$GW$) ia a novel methodology for electronic structure calculations, which addresses many shortcomings of the standard method, density functional theory. However, the calculations are very compute intense. In this eCSE project, parallel I/O using HDF5 and memory distributed parallelism using parallel BLAS (PBLAS) and ScaLapack has been implemented.

1 Introduction

Computational materials science has evolved into one of the most active and industry relevant fields of physics. For many classes of materials, the predictive power of density functional theory (DFT) is sufficient to motivate its use for automated searches for novel materials.

However, there are still many classes of materials, and also properties, for which DFT is not sufficient. Amongst the most most pressing problem is the failure to describe excited states in semiconductors or transition metal oxides. More generally, DFT (in its standard form) does not – by construction – describe excitations. Another problem is the failure to describe strongly correlated materials.

The Quasi-particle self-consistent GW (QS$GW$) is a novel theory, perhaps the most promising one that is independent of DFT, which has already proven successful for the calculation of band gaps and density of states for a wide selection of materials. In contrast to DFT it forms a natural bridge for extensions to other many-body effects captured, e.g. by DMFT. Ground state properties such as the total energy can be calculated with much higher accuracy than the LDA.

Over the past years QS$GW$ has been implemented by Mark v. Schilfgaarde. T. Kotani and coworkers. The resulting code is at the basis of the current "Software for the Future" CCP9 flagship project, which has the aim to turn the code into a user-friendly community code. A preliminary version of the web site (soon to be advertised) can be found here.

This eCSE project is part of this bigger picture and focusses on the parallelization of the code, and on adapting the code to large multi-core HPC systems.

The central quantity of QS$GW$is the so-called self-energy,

$$\Sigma(\mathbf{r},\mathbf{r}',\omega) = \frac{i}{2\pi}\int \mathrm{d}\omega' G^0(\mathbf{r},\mathbf{r}',\omega-\omega')W(\mathbf{r},\mathbf{r}',\omega')e^{i\delta\omega'}, \quad (1)$$

which is given in terms of the Green's function of a non-interacting reference system,

$$G^0(\mathbf{r},\mathbf{r}',\omega) = \sum_i \frac{\Psi_i(\mathbf{r})\Psi_i^*(\mathbf{r}')}{\omega-\varepsilon_i \pm i\delta}, \quad (2)$$

and the screened Coulomb interaction, $W(\mathbf{r},\mathbf{r}',\omega')$, between the electrons. Details about the method can be found in publications [1, 2].

The most compute intense parts of the code calculate the following equations, defining the self energy, $\Sigma(\omega)$ in terms of the screened Coulomb interaction $W_{I,J}(\mathbf{q}, \omega)$:

$$
\begin{aligned}
&\langle \Psi_{\mathbf{q},\mathbf{n}} | \Sigma(\omega) | \Psi_{\mathbf{q},\mathbf{m}} \rangle = \\
&= \sum_{\mathbf{k}}^{\mathrm{BZ}} \sum_{n'}^{\mathrm{all}} \sum_{I,J} \langle \Psi_{\mathbf{q},\mathbf{n}} | \Psi_{\mathbf{q}-\mathbf{k},\mathbf{n}'} \bar{M}_I^{\mathbf{k}} \rangle \langle \bar{M}_J^{\mathbf{k}} \Psi_{\mathbf{q}-\mathbf{k},\mathbf{n}'} | \Psi_{\mathbf{q},\mathbf{m}} \rangle \times \\
&\quad \times \int_{-\infty}^{\infty} \frac{i d\omega'}{2\pi} W_{I,J}(\mathbf{k}, \omega') \frac{\mathbf{1}}{-\omega + \omega' - \varepsilon_{\mathbf{q}-\mathbf{k},\mathbf{n}'} \pm \mathbf{i}\delta'}
\end{aligned} \tag{3}
$$

The screened Coulomb interaction, which is given as a matrix in the product basis (labelled by I, J) for each $\mathbf{q}$ and ω is obtained from the bare Coulomb interaction $V(\mathbf{q})$ and the so-called polarization propagator $\Pi_{I,J}(\mathbf{q}, \omega)$, can be obtained by matrix inversion (an underlined quantity represents a matrix in I and J):

$$
\underline{W}(\mathbf{q}, \omega') = [\underline{\mathbf{1}} - \underline{\mathbf{V}}(\mathbf{q}) \underline{\mathbf{\Pi}}(\mathbf{q}, \omega)]^{-\mathbf{1}} \underline{\mathbf{V}}(\mathbf{q}) \,. \tag{4}
$$

The polarization propagator $\Pi_{I,J}(\mathbf{q}, \omega)$, in turn, is defined by:

$$
\begin{aligned}
\Pi_{I,J}(\mathbf{q}, \omega) &= \sum_{\mathbf{k}}^{\mathrm{BZ}} \sum_{n}^{\mathrm{occ}} \sum_{n'}^{\mathrm{unocc}} \frac{\langle \bar{M}_I^{\mathbf{q}} \Psi_{\mathbf{kn}} | \Psi_{\mathbf{q}+\mathbf{k},\mathbf{n}'} \rangle \langle \Psi_{\mathbf{q}+\mathbf{k},\mathbf{n}'} | \Psi_{\mathbf{k},\mathbf{n}} \bar{M}_J^{\mathbf{q}} \rangle}{\omega - (\varepsilon_{\mathbf{q}+\mathbf{k},\mathbf{n}'} - \varepsilon_{\mathbf{k},\mathbf{n}}) + i\delta} \\
&+ \sum_{\mathbf{k}}^{\mathrm{BZ}} \sum_{n}^{\mathrm{unocc}} \sum_{n'}^{\mathrm{occ}} \frac{\langle \bar{M}_I^{\mathbf{q}} \Psi_{\mathbf{kn}} | \Psi_{\mathbf{q}+\mathbf{k},\mathbf{n}'} \rangle \langle \Psi_{\mathbf{q}+\mathbf{k},\mathbf{n}'} | \Psi_{\mathbf{kn}} \bar{M}_J^{\mathbf{q}} \rangle}{-\omega - (\varepsilon_{\mathbf{q}+\mathbf{k},\mathbf{n}'} - \varepsilon_{\mathbf{k},\mathbf{n}}) - i\delta}
\end{aligned} \tag{5}
$$

Here the $\Psi_{\mathbf{kn}}$ are the single-electron wave functions resulting from the non-interacting reference Hamiltonian $\hat{H}_0$. The product basis $\bar{M}_I^{\mathbf{q}}$ is obtained from products of pairs of single-electron wave functions.

The size of the matrices $\langle \bar{M}_I^{\mathbf{q}} \Psi_{\mathbf{kn}} | \Psi_{\mathbf{q}+\mathbf{k},\mathbf{n}'} \rangle$ scales with the 3rd power in the system size (number of atoms in the simulation cell); the computation time as the 4th power.

2 Previous work

In a previous d-CSE project on Hector, a basic parallelization over the outermost loops, i.e. the loop over the $\mathbf{q}$-vectors for which the self-energy is calculated, and the $\mathbf{k}$ summation in the susceptibility has been implemented. While these steps did speed up the code for metallic systems (which require many **q**- and **k**-points) the project brought to light a number of bottlenecks, which were not obvious in runs for small systems and with low processor count:

- Parsing of the input files:
  The original code openend and read keywords from the input files whenever they were required in the code. While this approach is no problem on few processors, it started being a problem at larger processor counts.
- Sequential writing of output files:
  The output of data became a bottleneck. A temporary workaround of writing separate files per processor was implemented but did not provide an acceptable solution for a user-friendly code.
- Memory bottleneck for large systens (many atoms):
  The study of larger systems in terms of the number of atoms poses two problems: larger systems require less **q**- and **k**-points, hence making the parallelizatin less efficient, while at the same time increasing the amount of data required for each point, eventuallly reaching the limitations of the memory per node

These bottlenecks motivated the current project, which set out to address these points.

3 Current work

Due to the limited amount of time available in the eCSE project, one goal was to achieve the above goals with minimal changes to the overall structure of the code.

3.1 Global parallelization strategy

At its current state, the code uses (up to) 4 levels of parallelism:

1. Outer loop: **q** for which the qantity is to be calculated
2. Brillouin zone summation over **k**.
3. Parallelization over the basis functions I, J and matrix distribution via PBLAS, ScaLapack
4. Possible threaded MKL library for the remaining BLAS calls (not considered here)

To this end, the `MPI_COMM_WORLD` communicator is split into 3 layers of groups, corresponding to the **q** and **k** loop paralleliztion. The processors beloning to one (**q**, **k**) group form the square grid for the PBLAS parallelization.

3.2 Parsing the input file

The original input file parser was already implemented as a Fortran90 module, which encapsulated the file access. A new module has been written which has the same interface as the old one, and hence can easily replace all existing calls to the parser by changin the included module.

However, the new module implements a new data structure which represents the input file in memory. At the beginning of the code, this data structure is populated by parsing the input file on the root processor, and then broadcasting the data to all processors. Subsequent calls to the parser no longer read the input file from disk, but only access the data in memory. This avoids the bottleneck of all processes trying to access the input file simultaneously.

3.3 Parallel I/O

The first main objective of the project was to implement parallel I/O to the main parts of the code. In order to combine parallel I/O with machine-independent files, we decided for the HDF5 format.

In order to lower the barrier for other developers to adapt the HDF5 format, a small library has been written which encapsulates a large part of the HDF5 related book-keeping and minimizes the amount of code required by the application, while on the other hand leaving enough flexibility to deal with different data types, and MPI process configurations. The module does define the `complex(kind=1.d0)` data type and establishes default values for certain HDF5 properties.

For the large (and distributed) arrays, the hyperslab methodology is used to allow certain MPI processes to access parts of the array.

As within each **q** group the summation over **k** is performed using `MPI_Reduce` before the data is written, only those MPI processes corresponding to the 1st **k** point for each **q** are accessing the HDF5 files.

Due to the fact that the number of processors per group is not necessarily commensurate with the loop count, in general not all processors will execute the same number of loop iterations. For this reason, the access to the HDF5 files had to be set up as individual, as opposed to collective, access.

3.4 Memory distribution

The second objective was to distribute the big arrays over the nodes to facilitate the calculation of large systems, and at the same time parallelize the generation of the matrix elements over the processors.

The largest arrays in the code are matrices between the product basis and the single-electron wave functions: $\langle \bar{M}_I^{\mathbf{q}} \Psi_{\mathbf{kn}} | \Psi_{\mathbf{q+k},\mathbf{n}'} \rangle$, where the number of product basis functions scales roughly linearly with the number of single electron basis functions. Note, however, that $\bar{M}_I^{\mathbf{q}}$, $\Psi_{\mathbf{kn}}$ and $\Psi_{\mathbf{q+k},\mathbf{n}'}$ each scale linearly with the system size, the total number of matrix elements scales as the third power.

To overcome the memory limit of the nodes, the distribution is implemented via BLACS distribution of the arrays. For many of the bottlenecks, it was sufficient to replace the zgemm BLAS3 calls by corresponding pzgemm calls.

For the matrix elements $\langle \bar{M}_I^{\mathbf{q}} \Psi_{\mathbf{kn}} | \Psi_{\mathbf{q+k},\mathbf{n}'} \rangle$, the original code used 3-dimensional arrays for indices I, n and n'. For BLAS calls they were treated as 2-dimensional, merging either the first or the second index.

For instance, code such as,

```
      allocate(zzmel(ngb,         nt,    ntqxx) )
      allocate( zmel(ngb,         nstate,ntqxx) )
      allocate( ppovlz(ngb, ngb) )


      call zgemm('C','N',ngb,nstate*ntqxx,ngb,
     .         (1d0,0d0),
     .         ppovlz,ngb,
     .         zzmel,ngb,
     .         (0d0,0d0),
     .         zmel,ngb)
```

have been replaced by

```
type(PBLAS_DIMS_2D)       :: zmel_dims, zzmel_dims, gb
integer, dimension(DLEN_) :: DESC_zmel, DESC_zzmel, DESC_ppovlz



gb         = make_PBLAS_dims(ngb,ngb)

zzmel_dims = make_PBLAS_dims3(ngb,nt,ntqxx,2)
zmel_dims  = make_PBLAS_dims3(ngb,nstate, ntqxx, 2)


call myDESCINIT(ICTXT, gb,         DESC_ppovlz, ierror)
call myDESCINIT(ICTXT, zmel_dims,  DESC_zmel,   ierror)
call myDESCINIT(ICTXT, zzmel_dims, DESC_zzmel,  ierror)

call Pzgemm('C','N',ngb,nstate*ntqxx,ngb,
.         (1d0,0d0),
.         ppovlz, 1, 1, DESC_ppovlz,
.         zzmel,  1, 1, DESC_zzmel,
.         (0d0,0d0),
.         zmel,   1, 1, DESC_zmel )
```

The data structure `PBLAS_DIMS_2D` has been introduced to encapsulate information about the dimensions and the local array boundaries:

```
type PBLAS_DIMS_2D

   integer :: global_r, global_c ! global dimensions
   integer :: NB_r, NB_c         ! block size
   integer :: local_r, local_c   ! local dimensions
   integer :: lo_r, lo_c         ! lowest index of slab
   integer :: hi_r, hi_c         ! highest index of slab

end type PBLAS_DIMS_2D
```

As seen in the code example, a number of helper routines have been implememted, which set up the distribution scheme, or create the PBLAS array descriptors.

A complication for the PBLAS implementation was that some routines of the code addressed contiguous sub arrays of the large arrays.

This is due to the fact that the product basis consists of different types of functions (e.g. atomic and plane-wave like wave functions), which require different treatment for setting up the matrices.

In order to prevent further complex book keeping, the decision has been taken to break up the matrices avoiding the block cyclic distribution, which could lead to several blocks being located on one processor.

In this way, most routines could be converted to PBLAS simply by replacing the BLAS calls by the corresponding PBLAS calls. Only a few routines required explicit communication to swap submatrices between the processors.

4 Results and Conclusion

Unfortunately, due to the average queuing times on Archer, it was not possible to get enough timing information by the time of writing the report to present a detailled study of the performance. Test runs have been performed for a 16 atom and a 32 atom supercell of GaAs. Preliminary results show, that the code is working and produces the correct results.

The calculations even for the 16 atom cell is not possible without either underpopulating the nodes, or using the memory distributed verison of the code. Without underpopulating the nodes, calculations become possible when using at least a 3x3 BLACS grid. Systematic studies of the performance as function of processor count are ongoing. Preliminary timing data for the calculation of the screened Coulomb interaction and for the correlation energy are given in Table 1.

These results show a number of points:

- The calculation of W does not show any speedup with the number of cores. This indicates, that the time is dominated by the I/O time. In the current implementation, the HDF5 writes had to be independent due to incummensurate iteration counts of the loops with respect to the processor number. An attempt to bypass the problem by using independent communicators for the various groups, lead to wrong results, indicating that some data race conditions. The problem does not occur in the self energy part, where the largest file is only read in, not requiring any blocking.

Table 1: Wall clock times for calculations of the screened Coulomb interaction W and the correlation part of the self energy, Σ_c for a 16 atom supercell of GaAs. N_q and N_k are the number of processor groups used to parallellize the loops over $\mathbf{q}$ and $\mathbf{K}$ points. N_{BLACS} is the linear dimension of the BLACS grid used to disribute the large matrices, and N_{under} is the factor for underpopulating the nodes. The total number of core is $N_{\mathrm{total}} = N_q \times N_k \times N_{\mathrm{BLACS}}^2 \times N_{\mathrm{under}}$. The times are given in hh:mm format. OOM indicates that the job was killed with an out-of-memory message.

N_q	N_k	N_{BLACS}	N_{under}	N_{MPI}	N_{total}	Time (W)	Time (Σ_c)
5	5	2	1	100	100	OOM	in queue
5	5	2	2	100	200	3:21	3:37
5	5	3	1	225	225	3:51	1:59
5	5	3	2	225	450	3:16	1:56
5	5	4	1	400	400	3:48	1:12
5	5	4	2	400	800	3:20	1:16
5	5	5	1	625	625		0:54

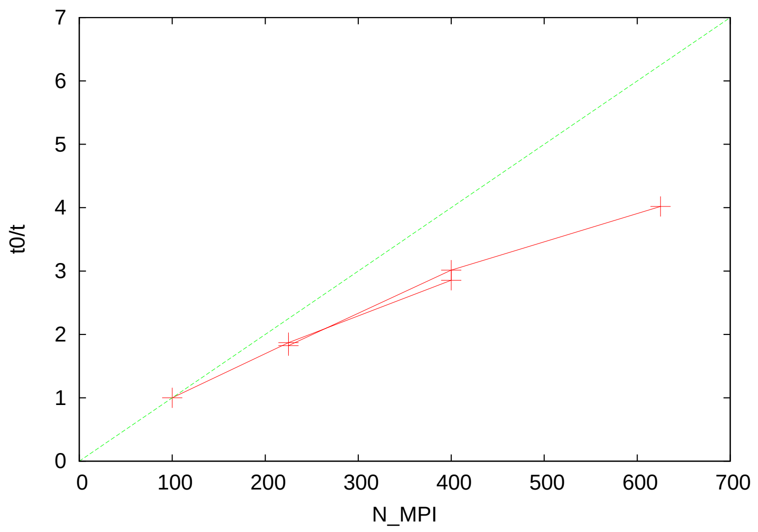
- Underpopulating the nodes leads to a small speedup in the calculation of the screened Coulomb interaction, as it improves the I/O bottleneck. As expected it has no noticable effect for the calculation of the correlation energy.
- The parallelisation using PBLAS (mainly pzgemm) seems to give very good speedup of the code. Figure 1 shows the speedup of the correlation code as function of number of MPI tasks.

4.1 Remaining bottlenecks

Despite the parallel I/O implementation with HDF5, writing the screened Coulomb interaction remains the main bottleneck. The main reasons for that are:

- the file size:
  for a 32 atom GaAs supercell, the screened Coulomb file was 215 GB. A 96 atom cell will require a file 27 times larger.
- individual file access:
  In general, the number of $\mathbf{q}$- and $\mathbf{k}$-points is incommensurate

Figure 1: Speedup (inverse run time) of the correlation code as function of MPI tasks used. The two different sets of data correspond to the runs with and without underpopulating the nodes. A linear slope would correspond to ideal scaling.



with the number of MPI groups, used for parallelizing the corresponding groups.

In order to allow for systems larger than the current ones, it is essential to avoid the huge file containing the screened Coulomb interaction all together. It is possible to revert the loop structure and furthermore, combine the two codes calculating the screened Coulomb interaction, and the self energy into one code. In this way, the data for one $\mathbf{q}, \omega$ pair have to be kept in memory at a given time.

However, this requires more fundamental changes to the code, which have not been possible in the limited time of this eCSE project.

The PBLAS and HDF5 modifications of the code, performed during this eCSE will help the planned changes, and are necessary also for the planned "inverted loop structure".

5 Outlook

With the HDF5 and PBLAS implementations in place, it will be possible to continue the planned changes as part of the CCP9 flagship project (supported by the STFC through the SLA core support), or through another eCSE proposal.

The modified version of the code is currently available in a separate branch of the git repository [need a link here]. After some more testing it will be merged into the master branch.

Acknowledgements

This work was funded under the embedded CSE programme or the ARCHER UK National Supercomputing Service (http://www.archer.ac.uk).

References

[1] T. Kotani and M. van Schilfgaarde. All-electron *GW* approximation with the mixed basis expansion based on the full-potential LMTO method. *Sol. State Comm.*, 121:461, 2002.

[2] Takao Kotani, Mark van Schilfgaarde, and Sergey V. Faleev. Quasiparticle self-consistent *GW* method: A basis for the independent-particle approximation. *Phys. Rev. B*, 76:165106, 2007.

[3] The code is hosted on http://bitbucket/lmto/