Adapting QSGW to large multi-core systems

Abstract

The quasi-particle self-consistent GW approxation, recently developed and coded by Prof. Mark v. Schilfgaarde (King's College London) is an extremely powerful method for the calculation of the electronic structure, and in particular excitations, which addresses many of the problems which plagued the standard density functional approach. However, the calculations are extremely costly and this project aimed at harnessing the power of massively parallel machines to speed up these calculations. In the most expensive part of the code, an absolute speedup of more than 2000 has been achieved by using about 4500 cores on HECToR.

Contents

Introduction

The Quasi-particle self-consistent *GW* (QS*GW*) approximation has emerged as a kind of gold standard among *ab initio* approaches to electronic structure. It addresses the electronic structure simultaneously at an optimized single-particle level, and at a many-body level. It thus serves as a high-end replacement for the usual Kohn-Sham density-functional theory (KS-DFT), surmounting most of that theory's well known drawbacks, and as a way to bridge the gap between single-particle and many-body theories in a clean, parameter-free way free of ambiguities inherent in the currently popular methods such as LDA+DMFT. The price to be paid is computational cost. QS*GW* is roughly 100-1000 times more expensive than LDA.

In it's development stage at the beginning of this project, the code was essentially a sequential code, and the execution time for physically interesting systems was in the order of month. In order to make QSGW calculations feasible for routine calculations, it was crucial to parallelize the code.

In the next section, we will outline the parallelization opportunities of the QSGW method.

Loop structure of the codes and parallelization strategy

One of the central quantities to be calculated in the QSGW method is the polarizability $\Pi(\mathbf{r},\mathbf{r}',\mathbf{q},\omega)$:

$$\Pi_{IJ}(\mathbf{q},\omega) = \sum_{\mathbf{k}}^{\mathrm{BZ}} \sum_{n}^{\mathrm{occ}} \sum_{n'}^{\mathrm{unocc}} \frac{\langle \tilde{M}_I^{\mathbf{q}} \Psi_{\mathbf{k}n} | \Psi_{\mathbf{q}+\mathbf{k}n'} \rangle \langle \Psi_{\mathbf{q}+\mathbf{k}n'} | \Psi_{\mathbf{k}n} \tilde{M}_J^{\mathbf{q}} \rangle}{\omega - (\varepsilon_{\mathbf{q}+\mathbf{k}n'} - \varepsilon_{\mathbf{k}n}) + i\delta}$$
$$+ \sum_{\mathbf{k}}^{\mathrm{BZ}} \sum_{n}^{\mathrm{unocc}} \sum_{n'}^{\mathrm{occ}} \frac{\langle \tilde{M}_I^{\mathbf{q}} \Psi_{\mathbf{k}n} | \Psi_{\mathbf{q}+\mathbf{k}n'} \rangle \langle \Psi_{\mathbf{q}+\mathbf{k}n'} | \Psi_{\mathbf{k}n} \tilde{M}_J^{\mathbf{q}} \rangle}{-\omega - (\varepsilon_{\mathbf{k}n} - \varepsilon_{\mathbf{q}+\mathbf{k}n'}) + i\delta}$$

The main loops are the ones over **q** and ω, i.e. the wave vectors and frequencies for which the function has to be calculated, and the internal summation over the wave vectors k and band indices n and n'. These nested loops were treated by nested levels of MPI parallelization, utilizing different processor groups.
The calculation of the self-energy has a very similar structure of loops.

The outermost **q** loop was rather straight forward, as the calculations involved are independent of each other, and required hardly any communication, apart from broadcasting global data to all nodes. When possible, output files were written locally on the head node of the processor group for a given **q** vector.

The inner **k**-summations required an additional reduction. In addition, expensive calculations of matrix elements were parallelized (mostly over the basis set) using OpenMP. Results for the scaling with respect to these three levels of parallelization will be given and discussed below.

Implementation details

MPI parallelization:

The nature of the nested loops (**k** and **q**) called for independent parallelization at these two levels. To this end, the pool of N MPI processes (described by `MPI_COMM_WORLD`) was split into N_q groups of N/N_q processes. The loop over **q** vectors was then parallelized over the N_q groups:

```
q_start = q_comm%group_ID
q_step =  num_q_groups

do 1001 iq = iqxini + q_start, iqxend, q_step

      ! [code for each q vector ...]

enddo ! 1001
```

Inside each q group, the parallelization over the **k**-points followed the same principle. The processes of the **q**-group were further divided into groups, over which the k-loop is parallelized.

```
k_start = k_comm%group_ID
k_end   = nqbz
k_step  = num_k_groups

do 1000 k = 1 + k_start, k_end, k_step

      ! [code for each k-vector…]

enddo ! 1000
```

Each processor group is summing the accumulating variable (`rcxq`) over it's set of k-vectors, which leaves a final reduction(`MPI_SUM`) over the different **k**-groups:

```
#ifdef  USE_MPI

      CALL MPI_ALLREDUCE(MPI_IN_PLACE, rcxq, rcxq_size,
     &     MPI_DOUBLE_COMPLEX, MPI_SUM,
     &     k_inter%communicator, ierror)

#endif
```

`k_inter` are MPI process groups, which contain processes of equal rank from the different **k**-groups (see details in the next section).

The incentive for implementing the k-groups was to keep the flexibility of introducing a further level of MPI-parallelization (e.g. over bases functions) for each given **k**-point. In practice, however, at the current stage of the project, the **k**-groups are of size 1, i.e. the number of k-groups N_k equals the number of processes in each **q**-group.

$$N_k = N / N_q.$$

Optimal performance is to be expected when the number of groups equals the number of q/k points, or is a divisor of this number. Currently, these group sizes are not determined automatically, but have to be specified in `MPI_name.init` files, which are read by the codes.

MPI module

In order to handle the different levels of parallelization, an MPI module has been written, which provides a data structure for MPI processor groups, and routines to split a given processor group into subgroups. The data structure `comm_struct` has been introduced to hold all relevant information about a given processor group:

```
type comm_struct
        integer :: size
        integer :: ID
        integer :: communicator
        integer :: group_ID
        integer :: num_of_siblings
        integer :: ID_of_head
end type comm_struct
```

where `size` is the number of processors in the group, `ID` labels each group (at a given level), `communicator` is the handle of the MPI communicator. `num_siblings` stores the information on how many groups exist at this given level, and `ID_of_head` is the rank (with respect to `MPI_COMM_WORLD`) of the head of the group.

The routine `mpi_make_groups()` is provided to split an existing processor group `base_comm` into a set of subgroups `new_comm` and the corresponding groups `inter_comm`, which communicate between equal ranks of different subgroups.

```
subroutine mpi_make_groups (base_comm, loop_count, num_groups, new_comm,
inter_comm)

type(comm_struct), intent(IN)    :: base_comm
integer,           intent(IN)    :: loop_count
integer,           intent(INOUT) :: num_groups
type(comm_struct), intent(OUT)   :: new_comm, inter_comm
```

The following table illustrates the scheme of groups:

```
!-------------------------------------------------------------------------
! level 0:  (COMM_WORLD)  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | ...
!-------------------------------------------------------------------------
! level 1:  comm_1        |    group 0    |    group 1    |    group 2    |
..
!  ID                     | 0 | 1 | 2 | 3 | 0 | 1 | 2 | 3 | 0 | 1 | ...
!-------------------------------------------------------------------------
!           inter_1       | 0 |           | 1 |           | 2 |
!                           | 0 |           | 1 |           | 2 |
!                             | 0 |           | 1 |           | 2 |
!                               | 0 |           | 1 |           | 2 |
!-------------------------------------------------------------------------
! level 2:  comm_2 (1)    | gr 0  |  gr 1 |
!  ID                     | 0 | 1 | 0 | 1 |
!-------------------------------------------------------------------------
!           inter_2(1)    | 0 |   | 1 |
!                           | 0 |   | 1 |
!-------------------------------------------------------------------------
! level 2:  comm_2 (2)                  | gr 0  |  gr 1 |
!  ID                                   | 0 | 1 | 0 | 1 |
!-------------------------------------------------------------------------
!           inter_2(2)                  | 0 |   | 1 |
!                                         | 0 |   | 1 |
!-------------------------------------------------------------------------
! level 2:  comm_2 (2)                                | gr 0  |  gr 1 |
!  ID                                                 | 0 | 1 | 0 | 1 |
!-------------------------------------------------------------------------
!           inter_2(3)                                | 0 |   | 1 |
!                                                       | 0 |   | 1 |
!-------------------------------------------------------------------------
```

The module was written quite generally, supporting many nested levels of loop parallelization. In the current project, two levels were used.

I/O parallelization

Many input/output files of the code were already written as separate files for each **q**-point in the original sequential version of the code. In these cases, quite naturally the reading and writing is now done by the head node of each **q**-group.

Global input files are read only on `rank=0` and the data is then broadcast to all other processors.
In the program hx0fp0, which generates the susceptibility (see equation above), a major bottleneck was writing the output files. Originally, the files containing the screened Coulomb interaction for each q-vector, contained all Frequencies w as records in a direct access file. Depending on the system size, each file could reach a size of about 1 GByte. In order to accelerate the output of the files, each record, i.e. each frequency, is now written by a different processor, and the files are subsequently concatenated into the original format.

OpenMP parallelization

While the MPI parallelization went in a top-down philosophy, OpenMP was implemented in a bottom-up way, parallelizing simple loops at a lower level of the code, such as matrix multiplications used in the calculation of matrix elements between basis functions and Bloch wave functions

lmgwsc script

The lmgwsc code, which is to be called by the user, is actually a shell script, which calls separate binaries for different steps of the calculation. These steps include:

- Solution of the band-structure for a given effective potential
- Calculation of the bare Coulomb interaction v(r,r')
- Calculation of the Exchange (Fock) part of the self energy
- Calculation of the susceptibility and the screened Coulomb interaction
- Calculation of the Correlation part of the self energy
- Calculation of the new effective potential for the next iteration

The script also checks the convergence of the procedure and stops the procedure when a given accuracy is reached.

This script, however, cannot be called in parallel on parallel computers, but the individual binaries for the respective steps of the calculation have to be called in parallel. Therefore, the script had to be modified and the command to start a parallel execution, e.g. `aprun` on Hector, has been inserted via environment variables `$MPIRUN_`*`code`*, into the script. Code specific scripts allow for different number of processors for different steps of the calculation.

Also, the use of these variables makes it easy to adopt the codes to a different machine, which uses different MPI commands, such as `mpirun`.

Comparison against the milestones of the proposal

WP1. Parallelization of the self-energy Σ and the polarizablity Π calculation over k, q and ω. (1 April 2013 - 31 May 2013)

Parallelization of the loops over q, k ω, and ω': This work package requires modification of the programs generating the polarization function and the self-energy.

Polarization function:(trivial: q,ω; with global reduction: k) two routines will have to be modified:
hx0fp0.sc.m.F for q loop and x0fk_v3.F for ω and k loops If each loop is distributed over around 16 MPI tasks, this will give rise to a parallelism up to around 4096 MPI tasks.

Self-energy (incl. screened Clb. Interaction): (trivial q; with global reduction: k, ω') two routines will have to be modified: hsfp0.sc.m.F for q loop and sxcf_fal2.sc.F for the k and ω' loops. If each loop is distributed over around 16 MPI tasks, this will give rise to a parallelism up to around 4096 MPI tasks.

Deliverable 1: New code will give the same results as original code and scale up to 2000 cores on HECToR. **(Due 31 May 2013)**

The deliverable has been met, as demonstrated in the in the results section below

WP2. Memory use on parallel machines. (1 June 2013 - 30 September 2013)

Parallelization over basis functions, distributed storage of matrices: In this second, and major work package, the distributed memory model will be implemented to handle Π_{IJ} sub-matrices for large models. Loop structures will be re-ordered and distributed memory, mixed mode routines for matrix multiplications and inversion will be deployed. Furthermore, OpenMP will be used to parallelize other inner loops, which do not contain calls to mathematical libraries. This level of parallelism will not affect the scaling of the first level. The parts of the package which will be affected from this change are:

Polarization function:(hx0fp0.sc.m.Fand x0fk_v3.F): loops over the *(I,J)*-blocks have to be moved from x0fk_v3.F to hx0fp0.sc.m.F. Allocation of the large arrays will be moved inside the loops over the blocks. There are about 15 large arrays, which will be distributed over MPI tasks.

Bare and Screened Coulomb interaction: (hvccfp0,hx0fp0.sc.m.F, wcf.F): Both the bare (*v*, in hvccfp0) and the screened (*W*, in wcf.F) Coulomb interactions will be generated and stored in partitioned form so that it can be used directly for the generation of the self-energy. Restructuring of the loops is similar to the polarization function. For the matrix inversion in (wcf.F) different parallel algorithms (ScaLapack, PLASMA, iterative solution via Dyson equation) will be used depending on the system size.

Self-energy: hsfp0_sc, sxcf_fal2.sc.F: same as above for loop structures. Deployment of parallel libraries for the matrix multiplication in the required matrix elements of the self-energy:

$$\langle \Psi_{qn} | \Sigma_c(\omega) | \Psi_{qm} \rangle = \sum_{\mathbf{k}}^{BZ} \sum_{n'}^{All} \sum_{IJ} \langle \Psi_{qn} | \Psi_{q-\mathbf{k}n'} \tilde{M}_I^{\mathbf{k}} \rangle \times \langle \tilde{M}_J^{\mathbf{k}} \Psi_{q-\mathbf{k}n'} | \Psi_{qm} \rangle \int_{-\infty}^{\infty} \frac{id\omega'}{2\pi} W_{IJ}^c(\mathbf{k}, \omega') \times \frac{1}{-\omega' + \omega - \varepsilon_{q-\mathbf{k}n'} \pm i\delta}, \quad (34)$$

Deliverable 2: Distributed memory code gives same results as original code, and calculations for up to 100 atoms will be possible. For such systems, the code should scale well up to several 10000 cores, using OpenMP within the NUMA regions and MPI at several layers between them. A final project report will be produced in both pdf and html format ready for publishing on the HECToR website. **(Due 30 September 2013)**

This deliverable has not been fully met, due to problems, encountered at the start of the project (see below). So far, only the OpenMP parallelization has been implemented. It also addresses the memory issues, through shared memory use.

The work of WP2 has started, though. The arrays which should be distributed have been identified, and a small test code, demonstrating the use of BLACS and ScaLapack has been written.

Problems encountered

One difficulty in this project was, that the code contains a substantial amount of legacy code, written in a mixture of Fortran77 and Fortran95. Although the sequential code was well tested and working without errors, some constructs of the code made it difficult to debug new developments, or even broke with the introduction of parallelism. It took a considerate amount of time at the beginning of the project to find and correct these problems.

Scaling results

The two most costly steps in a self-consistent QSGW calculation are the calculation of the susceptibility and screened interaction, and the calculation of the self-energy contributions. In the following, we present timings for these two programs (hx0fp0 and hsfp0).

All calculations have been performed for a supercell of 16 Fe atoms. Similar calculations have been performed previously without the MPI parallelization and

a complete run took several months on 12 cores. These calculations employed a q-mesh of 10x10x10 q-points, which correspond to 125 irreducible points.

In our test calculations, we reduced the number of q-points to 5x5x5, resulting in 10 irreducible points.

Self energy calculation: hsfp0 (exchange mode)

Figure 1 displays the total speedup, relative to a sequential run of the code.

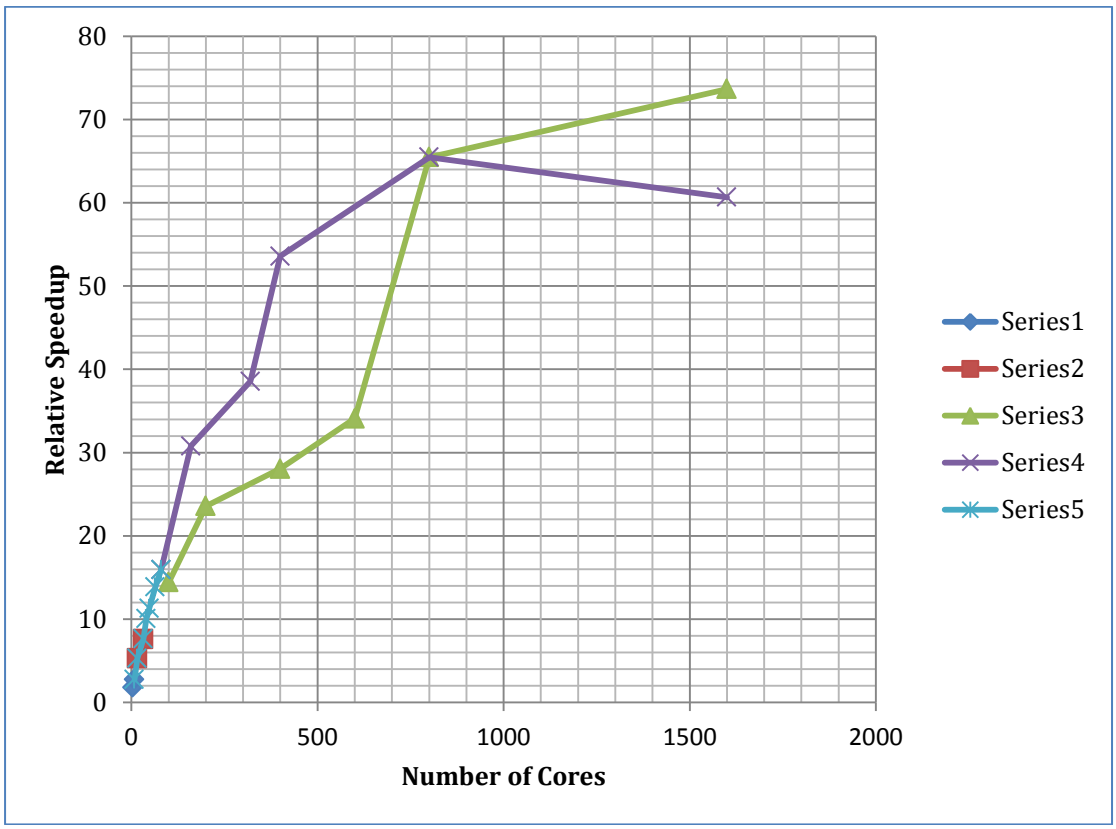

Figure 1: total speedup as function of processor count. The speedup values are with respect to a calculation using 1 MPI process and 4 OpenMP threads. The different series of calculations are shown in Table 1.

Figure 2 shows the relative speedup for the q-loop, k-loop and OMP parallelization. The speedup with respect to the outer q-loop is nearly linear, and is limited eventually by the number of q-vectors (here 10). It is to be expected that the nearly linear behavior extends further for calculations with more q-points. The parallelization over k-vectors breaks in earlier, most likely due to the communication, which is not present in the q-loop. The absent gain of performance for 20 processors for the k-loop is evident, as there are only 10 k-points in this sum. The slight reduction in performance is highlighting the cost of communicating. The data for OpenMP parallelization shows that 8 OMP threads seem to be the best setup.

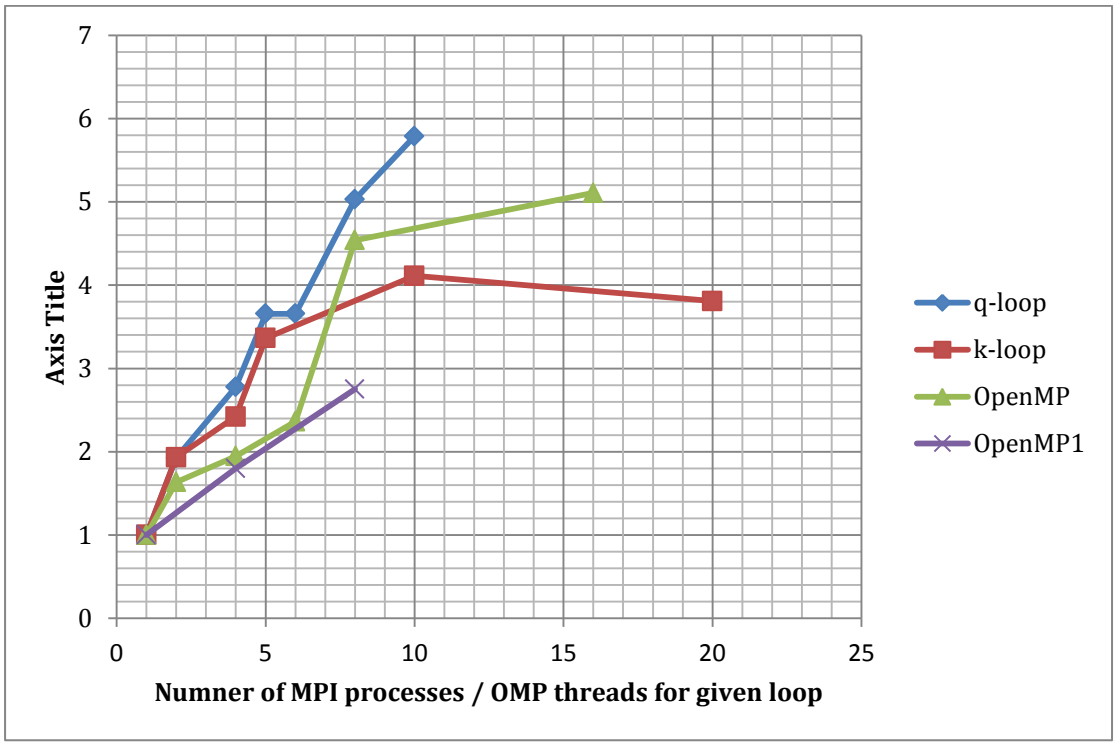

Figure 2: Speedup of the independent parallelization strategies.

The table shows the runtimes (wall clock) for the generation of the exchange part of the self energy. All runs were started from the same initial conditions, except for different processor counts for the various loops.

Table 1:

q-loop	k-loop	OMP	#Cores	Runtime (minutes)
		Series 1		
1	1	1	1	**412.5**
1	1	4	4	**229.4**
1	1	8	8	**149.9**
		Series 2		
10	10	1	100	**28.6**
10	10	2	200	**17.5**
10	10	4	400	**14.7**
10	10	6	600	**12.1**
10	10	8	800	**6.3**
10	10	16	1600	**5.6**
		Series 3		
1	2	8	16	**78.1**
1	4	8	32	**54.4**
		Series 4		
10	1	8	80	**25.9**
10	2	8	160	**13.4**
10	4	8	320	**10.7**
10	5	8	400	**7.7**
10	10	8	800	**6.3**
10	20	8	1600	**6.8**
		Series 5		
1	1	8	8	**149.9**
2	1	8	16	**77.9**
4	1	8	32	**54**
5	1	8	40	**41**
6	1	8	48	**36.5**
8	1	8	64	**29.8**
10	1	8	80	**25.9**

Screened Coulomb interaction: hx0fp0

Figure 3 shows the total speedup of the hx0fp0 code, which calculates the susceptibility and the screened Coulomb interaction, and is the most compute intensive part of the QSGW cycle. It should be noted that the values are relative to a calculation with 80 nodes. From the nearly linear behavior of the separate parallelizations (see below) we can estimate the sequential performance by extrapolation. We expect that this 80 core reference run should perform roughly 50 times faster than a sequential run. The "real" speedup should therefore be about 50 times the values, given in Figure 3. This would result in a maximum speedup of about 1380 for the 2640 core run. As the graph suggests, the calculation has not yet reached the saturated regime, and a further increase of the performance with higher core counts can be expected.

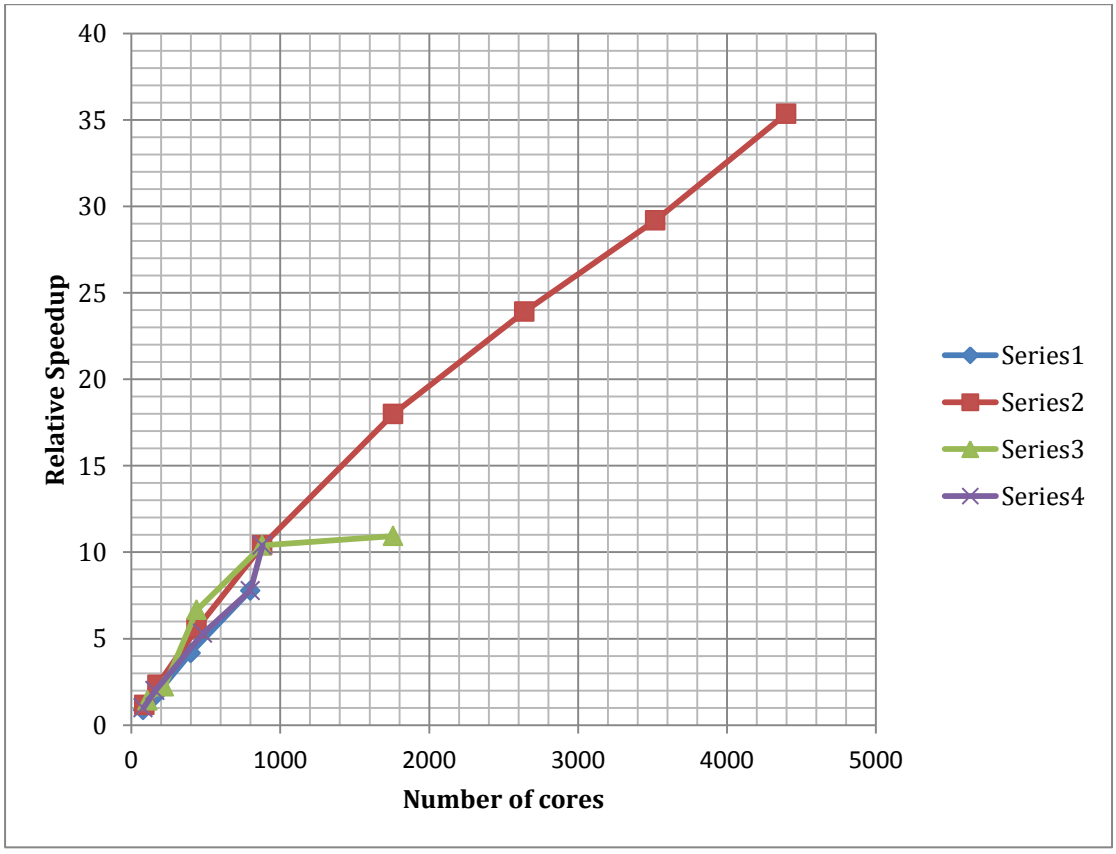

Figure 3: total speedup of the code calculating the screened Coulomb interaction. Please note that the baseline for this plot is a calculation, already using 80 cores. The different series of calculations are shown in Table 2.

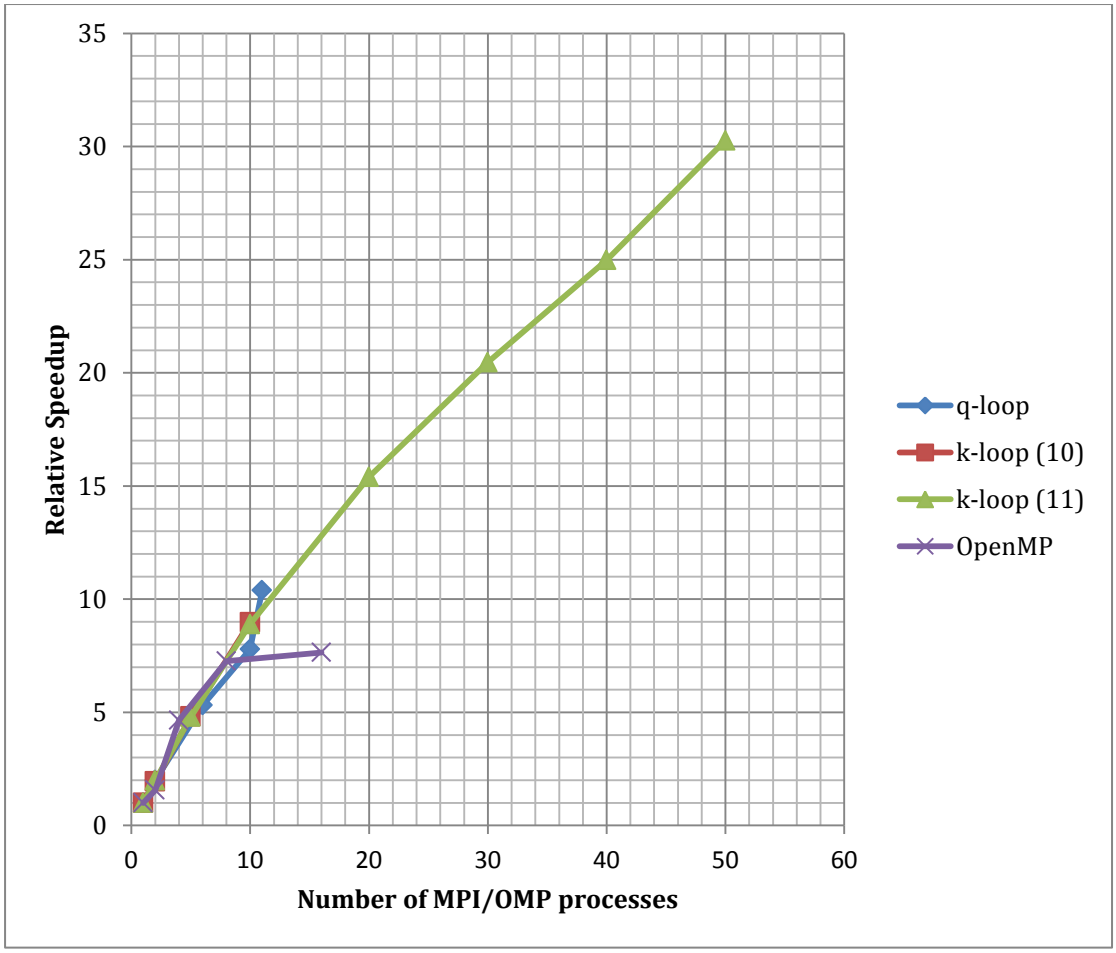

Figure 4: Speedup of the independent parallelization strategies.

As in the previous case, Figure 4 displays the speedup, achieved by parallelizing each of the loops separately. One can see a nearly perfect scaling behavior for both the q- and the k-loops. The scaling is better than in the self-energy generator, as the computational workload here is much higher.
The steep performance increase when going from 10 to 11 processor groups for the **q**-loop is due to the fact, that the code adds an additional q-point to the original list, in order to avoid problems caused by the singularity of the Coulomb interaction at **q**=0. This means, that with 10 processor groups, this last q-point has to be calculated by one processor group after the first pass of the loop over 10 points. Using 11 processor groups allows calculating all q vectors simultaneously. This, however, makes it necessary to run the different steps of the calculation with different number of processors, as was discussed above in context of the `lmgwsc` script.

The table shows the runtimes (wall clock) for the generation of the screened Coulomb interaction.

Table 2:

q-loop	k-loop	OMP	#Cores	Runtime (minutes)
		Series 1		
10	1	8	80	**579.5**
10	2	8	160	**297.3**
10	5	8	400	**120.2**
10	10	8	800	**64.5**
		Series 2		
11	1	8	88	**429.8**
11	2	8	176	**215.2**
11	5	8	440	**89.5**
11	10	8	880	**48.3**
11	20	8	1760	**27.9**
11	30	8	2640	**21**
11	40	8	3520	**17.2**
11	50	8	4400	**14.2**
		Series 3		
11	10	1	110	**351**
11	10	2	220	**224.4**
11	10	4	440	**75.4**
11	10	8	880	**48.3**
11	10	16	1760	**45.9**
		Series 4		
1	10	8	80	**502.2**
2	10	8	160	**250.8**
6	10	8	480	**94.5**
10	10	8	800	**64.5**
11	10	8	880	**48.3**

Self energy calculation: hsfp0 (correlation mode)

The correlation part of the self-energy is computationally more demanding as the exchange part, as it involves the additional loop over frequencies.

Figure 5 displays the total speedup.

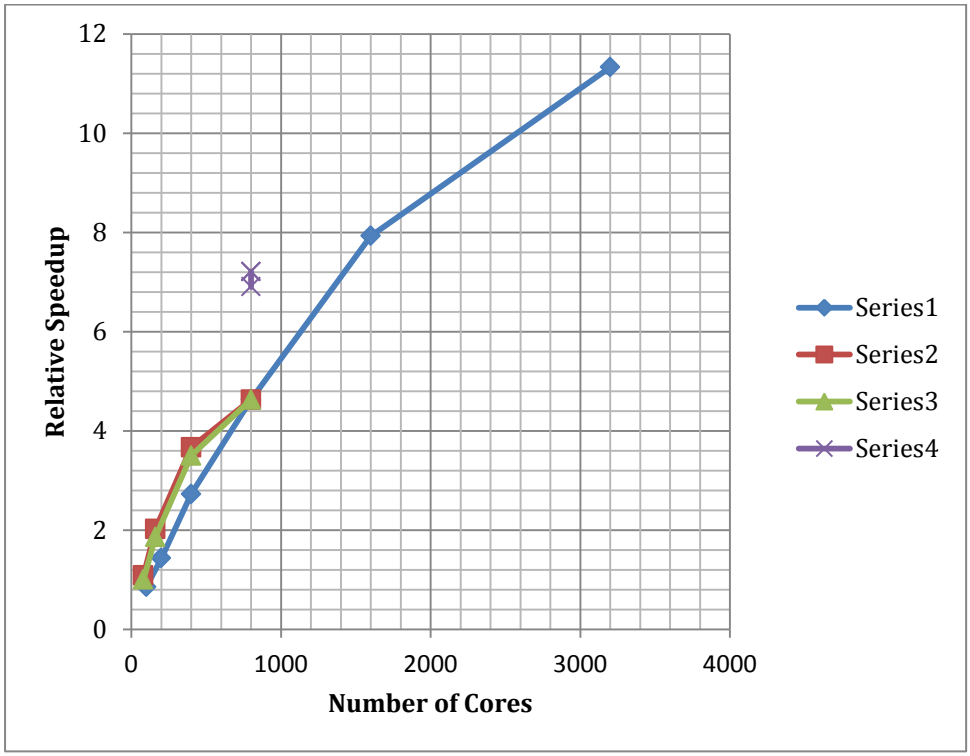

Figure 5: total speedup of the code calculating the correlation part of the self energy. Please note that the baseline for this plot is a calculation, already using 80 cores. The different series of calculations are shown in Table 3.

The better performance of the Series 4 runs as compared to the other runs is due to the better scaling of the OpenMP as compared to the MPI in this particular code (see below). In series 4, more cores were used for the OpenMP parallelization.

As in the previous case, Figure 6 displays the speedup, achieved by parallelizing each of the loops separately. One can see a nearly perfect scaling behavior for both the q- and the k-loops.

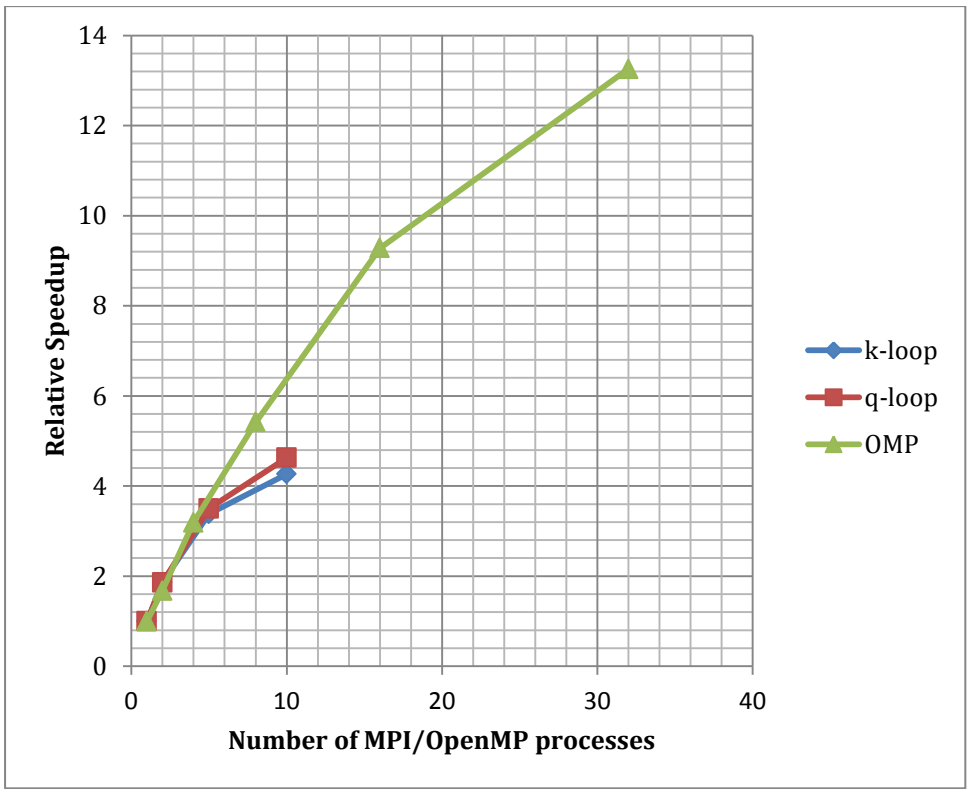

Figure 6: Speedup of the independent parallelization strategies.

It is interesting to see that in this part of the code, the OpenMP parallelization is more efficient than the MPI. This might indicate that also the other codes have some more optimization potential with respect to the OpenMP parallelization, which could be exploited further.

The table shows the runtimes (wall clock) for the generation of the correlation part of the self energy.

Table 3:

q-loop	k-loop	OMP	#Cores	Runtime (minutes)
		Series 1		
10	10	1	100	**287.7**
10	10	2	200	**171.5**
10	10	4	400	**90.2**
10	10	8	800	**53.1**
10	10	16	1600	**31**
10	10	32	3200	**21.7**
		Series 2		
10	1	8	80	**226.5**
10	2	8	160	**121.7**
10	5	8	400	**67**
10	10	8	800	**53.1**
		Series 3		
1	10	8	80	**245.8**
2	10	8	160	**132.3**
5	10	8	400	**70.2**
10	10	8	800	**53.1**
		Series 4		
5	5	32	800	**34.1**
10	5	16	800	**34.1**
5	10	16	800	**35.6**

Conclusion

It has been demonstrated that the costly QSGW calculations can be accelerated substantially by employing massively parallel machines, such as HECToR. The most severe bottleneck of the calculations was the assembly of the screened Coulomb interaction, which – for the given setup – would take the order of days for one cycle. With 2640 cores, this calculation was performed in 21 minutes, which would bring a whole self-consistent calculation from the order of weeks into the order of hours or in the worst case days. Even moderate parallelism of 440 cores resulted in a runtime of only 1.5 hours, which seems to be a good compromise between cost and speedup.

Impact

The QSGW methodology is rather new and the code was, so far, mainly used by the developer and his group. Without parallelization, the method would be too slow as to allow routine calculations. With actual QSGW calculations in reach, it is planned to include the code into the code base of the high performance computing consortia, in particular the materials chemistry consortium. CCP9 dedicated one of the Hands-On workshops to the LMF/QSGW code, in order to introduce the code to the community.

Outlook

This project has demonstrated the parallelization potential of the code. On the other hand, the benchmark results show that the scaling is not yet ideal, and further profiling should be performed. The amount of data written to and read from disk in each step also suggests to make use of MPI-I/O. Furthermore, other parts of the cycle, which have not been part of this project start becoming costly for larger systems. Examples are the code to set up the bare Coulomb interaction or the code to write out the required eigenvectors of the single particle band structure. These parts are still running sequentially and will benefit from similar parallelizations, as applied to the codes discussed here.

Furthermore, the accelerated code now opens the door to further developments on the methodological side. It is planned to submit a proposal for the implementation of a different method (the so-called Sternheimer equation) to solve parts of the equations. This planned project will build upon the codes and experiences generated in this dCSE project.

Acknowledgement

This project was funded under the HECToR Distributed Computational Science and Engineering (CSE) Service operated by NAG Ltd. HECToR – A Research Councils UK High End Computing Service - is the UK's national supercomputing service, managed by EPSRC on behalf of the participating Research Councils. Its mission is to support capability science and engineering in UK academia. The HECToR supercomputers are managed by UoE HPCx Ltd and the CSE Support Service is provided by NAG Ltd. http://www.hector.ac.uk