|epcc|

THE UNIVERSITY OF EDINBURGH

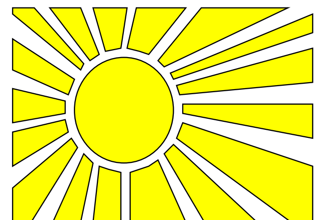
EPCC-SS-2001-12

I/O Issues and Benchmarking of the Parallel GW Space code

Scott Fraser

Abstract

The serial GW Space code written by the solid state physics group in York had previously been parallelised using MPI code, with a significant gain in performance during the intensive calculations but a considerable loss due to I/O issues. The I/O routines were re-written using new MPI and MPI-2 routines to attempt to increase the efficiency. The benchmarks of the resulting code are presented here, along with a discussion of the various I/O techniques.



Contents

1 Introduction **3**

2 Background **3**
2.1 The Physics 3
2.2 The *GW* space-time method 3
2.3 Parallelisation Strategy 3

3 I/O Techniques **4**
3.1 Method 1 4
3.1.1 Input using blocking sends 4
3.1.2 Reading input with all PEs 5
3.2 Use of MPI-2 to improve code 5
3.2.1 Reading input using MPI-2 5
3.2.2 I/O for temporary files (Basic data types) 5
3.2.3 I/O for temporary files (Derived data types) 5

4 Results **6**
4.1 Benchmarks for the initial MPI code 6
4.2 Timings for the MPI-2 Code 8

5 Conclusions **8**
5.1 Further work 8

6 Appendix A **A–9**

7 Appendix B **B–10**

8 Appendix C **C–11**
8.1 gwst.F90 . . . . C–11
8.2 polar.F90 . . . . C–11
8.3 selfx.F90 . . . . C–11
8.4 sigma.F90 . . . . C–11
8.5 greensrt.F90 . . . . C–12

1 Introduction

The GW space code is used to calculate the excited states of materials such as semi-conductors. However the code is very computationally intensive, and for realistic problems the execution time is too high. Accordingly a parallel version of the code has been developed using MPI.

While the execution time is lowered somewhat, the code is limited by several I/O routines. This project focuses on these I/O issues with the aim of improving the performance of the code, with both MPI and new MPI-2 methods.

2 Background

2.1 The Physics

In solid-state physics, we can calculate the ground state energy of a system using density functional theory (DFT) exactly in principle. However for many systems it is the excited states which are of interest, in particular intrinsic semi-conductors which are insulators in low energy states, but conductors in excited states. The normal method used in the DFT approach is the local density approximation (LDA) of Kohn and Sham[1], which formulates the problem in terms of a similar system of noninteracting electrons.

The main difference between our hypothetical system and the real system is that the exchange correlation self-energy operator Σ is ignored. However it has been shown[2] that for semiconductors, the *GW* approximation formulated by Hedin in 1965[3] allows computation of the band gaps in excellent agreement with experiment.

The code used here was developed by Reiger *et al*[4], the principle merit of which is that it does not rigidly adhere to reciprocal space in it's calculation. As parts of the calculation are more efficiently calculated in real space, the code takes advantage of highly optimised FFT routines to switch to the most efficient representation for each part of the calculation.

A parallel version of the code using MPI had previously been developed. This code used a great deal of computational time with expensive I/O procedures, and here these were to be improved by means of new MPI code and also by introducing MPI-2 code.

2.2 The *GW* space-time method

A detailed description of the method is given by Reiger *et al* [4] including the algorithm, explanations of the discretisation of the equations, numerical considerations and the scaling of the code.

However the GW approximation is described in appendix A, and description of how the code is related to the algorithm used is given in Appendix B.

2.3 Parallelisation Strategy

The code was initially parallelised using MPI[5], with details given in Appendix B.

The calculations in the code involve several large arrays, which are calculated and written to file. Accordingly these were split across the processors which would then calculate a small part of the overall array, before being gathered together and written to file.

The remainder of the changes were due to ensuring that all the required variables were on each processor, and that each processor knows which part of the array it should calculate. It was also necessary to reorder a few arrays.

Profiling of this code showed that a large proportion of the execution time is spent in MPI_BCAST calls, due to I/O issues.

3 I/O Techniques

3.1 Method 1

The first approach was to attempt to decrease the time taken for the input.

3.1.1 Input using blocking sends

The algorithm for this piece of code was as follows:

```
rec = 1
OPEN(handle,file,access='direct',recl)
IF(rank.EQ.0)THEN
        DO i=rec,rec+numread
            READ array(handle,rec=i)
        ENDDO
ELSE
        CALL MPI_RECV(rec,rank-1)
        DO i=rec,rec+numread
            READ array(handle,rec=i)
        ENDDO
ENDIF
CALL MPI_SSEND(rec,rank+1)
```

Where numread is the amount of data read in by each processor, roughly equal to the total amount divide by the number of processors. Each processor reads up to it's required record and then sends its final record number on to the next processor which starts reading from that record.

This method of input would be useful in the case where the data on file is an array and each processor required only one small section of the array. For the code used here each processor required all the data, and the method above will work only for unformatted input, while the input data was only available as formatted data.

It would also be possible to change this to write, rather than read, to a file. Either of these methods would produce exactly the same result as the MPI-2 read/write considered later. It would be interesting to test the performance of such an algorithm compared to the MPI-2 code.

3.1.2 Reading input with all PEs

The next idea to be considered was that of allowing all the PEs to access the file and read in the data. This removes the time-consuming broadcasts and two of the subroutines in the input phase.

Due to the fact that this reduces the portability of the code as some operating systems lock files when they are being read as well as when data is written to them, the code was written with a logical variable pe_all_read set in the control file.

This determines at runtime whether just one or all PEs should read the code.

3.2 Use of MPI-2 to improve code

The primary investigation into the use of MPI-2 centered around writing the temporary files created within program. Temporary files are created, written to and then are read again later in the program. Such temporary storage is necessary as the data is too large to be stored in memory. The use of MPI-2 to read in the data was also considered.

3.2.1 Reading input using MPI-2

Reading the input data with MPI-2 code was looked at but in this case it would require a great deal of work, with little benefit over previous methods, as it is geared towards unformatted input while as already mentioned the current code uses formatted input.

3.2.2 I/O for temporary files (Basic data types)

The original method using MPI-2 code to write out temporary files used basic datatypes and is not the most efficient way to write data.[6].

Each time the processor is about to write a part of the array to file, it calculates the offset at which it should do so and then uses the MPI-2 call MPI_FILE_WRITE_AT_ALL.

The program loops over itself several times to gradually build up the structure of the file. Using this MPI-2 method introduces another 2 loops in the test code. This is because the array used for testing had 3 indices, and each processor must write every part to a completely different section of the file.

When the small test code was integrated into the main program and the code was timed, the MPI-2 code performed considerable slower than the original MPI code. Therefore attention was turned to more efficient methods.

3.2.3 I/O for temporary files (Derived data types)

A more thorough investigation of MPI-2 revealed that using derived datatypes has the potential to increase the speed of the I/O processes by an order of magnitude compared to basic

datatypes[6]. In this case the implementation is much easier, as there is a new function in MPI-2, MPI_TYPE_CREATE_SUBARRAY which will give each PE a view of a section of an array only.

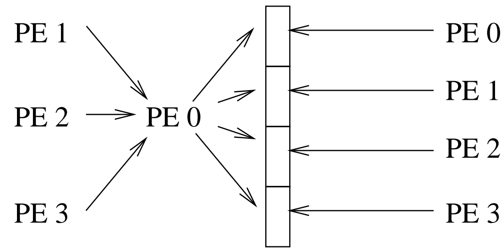


Figure 1: Schematic representation of standard MPI (left) and MPI-2 writing to file

Figure 1 shows how the different approaches view the file. With standard MPI all data is gathered onto one processor, which then writes to the file. With MPI-2 each processor writes it part of the array directly to the correct offset in the file.

The result is an algorithm similar to the following:

```
CALL MPI_TYPE_CREATE_SUBARRAY(...,filetype,...)
CALL MPI_TYPE_COMMIT(filetype)
CALL MPI_FILE_SET_VIEW(...,MPI_REAL,filetype,...)
CALL MPI_FILE_WRITE_ALL(...,subarray,...)
```

Where the MPI-2 call writes the subarray to the correct offset in the file such that when all PEs are finished, the array is in the correct order in the file.

While this was straightforward, difficulties arose later in the code when the data was read in again. This is because in the MPI-2 code the portion of code above is looped over and consecutive arrays are written in sequence to a file.

In the corresponding serial/MPI code, consecutive arrays are not written in sequence to the file. Further, with one particular array (**p0k** in **polar.F90**) it is read in later in a completely different sequence.

As MPI-2 lacks flexibility over the way it writes data to file, this means that for each set of arrays written out in this manner the code must be altered to read in section of code it would previously have done. This is a non-trivial matter, so the code has been tested on a section where the arrays are written sequentially to file.

4 Results

4.1 Benchmarks for the initial MPI code

Results are presented for a Cray T3E 1200 (CSAR service at Manchester):

Num/PEs	Time/min	Speedup
Serial	91.83	1
1	99.05	0.93
8	29.29	3.14
16	23.25	3.94
32	21.07	4.36
64	20.72	4.43

Table 1: Benchmarks for the MPI code

The previously developed MPI-1 code was recompiled with the optimisation flags -O3 and -Ounroll2 and benchmarked against the serial code with the same flags.

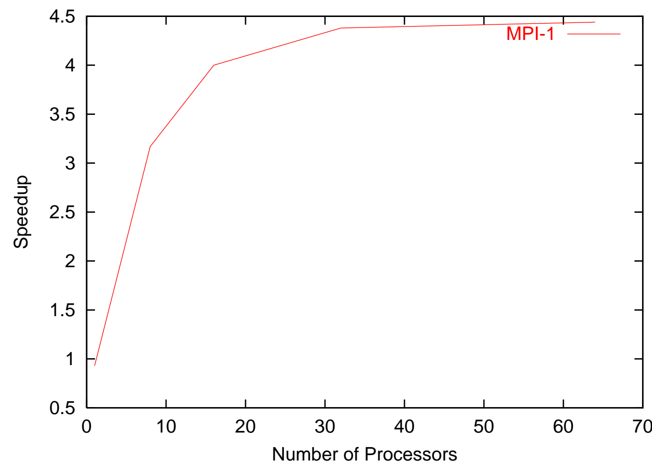


Figure 2: Speedup for MPI-1 Code

The code does not scale well because of the MPI_BCAST calls and sequential components. The code is most efficient at around 8 to 16 processors.

The use of pe_all_read was also tested for input:

Reading	pe_all_read=.false.	pe_all_read=.true.
Control file	3.79E-2	5.26E-2
Data up to **cast1.F90**	3.86E-2	7.42E-2
Data up to **cast3.F90**	0.193	0.188
eldank	2.34E-2	2.02E-2
qvec and rmax	0.856	0.836

Table2: Timings for various sections with pe_all_read

This increased the speed over most of the sections. Reading the control file took longer as 2 extra calls of MPI_BCAST were necessary.

In the original code one processor was used to read the data in, and in *cast1.F90* and **cast3.F90** these were bundled into a derived datatype and a single broadcast was then used. Table 2 shows that up to **cast1.F90** the original code was actually faster. Future work will investigate this further

In the other cases, where it was simply a case of reading data straight from file and then broadcasting it, the new method was faster.

4.2 Timings for the MPI-2 Code

Method	Time(s)
Original MPI	0.15
MPI2(basic datatypes)	0.397

Table 3: Timings for MPI-2 in polar.F90, basic datatypes

The first attempt at MPI-2 using basic datatypes was not fully implemented into the code. However it was tested over one small section which showed that the code was between 3 and 4 times slower than the original MPI code over that section.

Method	Time(s)
Original MPI	67.34
MPI2(derived datatypes)	87.82

Table 4: Timings for MPI-2 in polar.F90, basic datatypes

Using derived datatypes improved the time taken for the MPI-2 a great deal, and also made the implementation easier. However the original MPI code was still more efficient than the MPI-2 code.

5 Conclusions

While there is a substantial decrease in execution time for the MPI-1 program, it does not scale well as a speedup of slightly less than 4.5 with 64 processors leaves a great deal to be desired.

The speedup obtained with 8 and 16 processors is fairly satisfactory. The code is clearly limited by the I/O and the fact that large amounts of the code run sequentially.

The methods investigated for increasing the I/O did not produce a substantial difference. While reading in the data on all processors produces some gain, it is not very significant. More surprisingly, the MPI-2 code was outperformed by the original code. In addition, MPI-2's lack of flexibility meant that it could not be applied to the most time-intensive I/O call (in **polar.F90**) in the time given. While it would be possible to unravel the original code to determine in which sequence the arrays are written to file, this is a fairly difficult procedure.

5.1 Further work

- The MPI code using blocking sends should be tested more. If it proves more efficient than the original MPI code it could perhaps be written as a function and implemented as an MPI-2 style function call.
- Places where using pe_all_read is effective should be identified in the code and applied. The section where this method is slower than the original should be checked.
- A more thorough benchmarking of MPI-2 I/O processes should be done. While literature is available[6] to support the claims that MPI-2 I/O is faster, in this project this has not been the case. It would be useful to know where and when MPI-2 is more effective.

6 Appendix A

Hedin's proposal was that the self-energy operator can be approximated by

$$\Sigma(\mathbf{r},\mathbf{r}';\omega) = \frac{i}{2\pi}\int_{-\infty}^{\infty} d\omega' W(\mathbf{r},\mathbf{r}';\omega) \times G(\mathbf{r},\mathbf{r}';\omega+\omega')e^{i\omega'\delta} \tag{1}$$

where δ is an infinitesimal positive time and *W* is the screened Coulomb interaction given by

$$W(\mathbf{r},\mathbf{r}';\omega) = \int d^3\mathbf{r}'' v(\mathbf{r}-\mathbf{r}'')\epsilon^{-1}(\mathbf{r}'',\mathbf{r}';\omega) \tag{2}$$

in which $v(\mathbf{r}-\mathbf{r}'')$ is the Coulomb interaction $1/|\mathbf{r}-\mathbf{r}'|$ and *G* is the one-particle Greens function. Here, and in most practical calculations, *G* is approximated by the non-interacting Greens function at the LDA level:

$$G^{LDA}(\mathbf{r},\mathbf{r}';\omega) = \sum_{nk} \frac{\psi_{nk}(\mathbf{r})\psi^*_{nk}(\mathbf{r}')}{\omega - \epsilon_{n\mathbf{k}} - i\eta} \tag{3}$$

where η is a positive infinitesimal for occupied one-particle states, and negative for unoccupied states. The $\psi_{n\mathbf{k}}$ are eigenfunctions with the associated eigenfunctions $\epsilon_{n\mathbf{k}}$ which are obtained from a LDA calculation of the system.

The inverse dielectric function in Eq.(2) is obtained using the random phase approximation (RPA)

$$\epsilon^{RPA}(\mathbf{r},\mathbf{r}';\omega) = \delta(\mathbf{r}-\mathbf{r}'') - \int d\mathbf{r}'' v(\mathbf{r}-\mathbf{r}'')P^0(\mathbf{r}'',\mathbf{r}';\omega) \tag{4}$$

and here the irreducible polarization propagator P^0 is given by

$$P^0(\mathbf{r},\mathbf{r}';\omega) = -\frac{i}{2\pi}\int_{-\infty}^{\infty} d\omega' G^{LDA}(\mathbf{r},\mathbf{r}';\omega) \times G^{LDA}(\mathbf{r},\mathbf{r}';\omega-\omega') \tag{5}$$

In the method used here the efficiency is obtained due to the fact that when (1) and (5) are transformed from the frequency domain to the time(real or imaginary) domain they become simple multiplications. For real times (3) becomes

$$G^{LDA}(\mathbf{r},\mathbf{r}';\omega) = \begin{cases} i\sum_{nk}^{occ} \psi_{nk}(\mathbf{r})\psi^*_{nk}(\mathbf{r}')\exp(-i\epsilon_{n\mathbf{k}}\tau), & \tau < 0 \\ -i\sum_{nk}^{unocc} \psi_{nk}(\mathbf{r})\psi^*_{nk}(\mathbf{r}')\exp(-i\epsilon_{n\mathbf{k}}\tau), & \tau > 0 \end{cases} \tag{6}$$

Once we have a value for the self-energy we treat this as a peturbation on the LDA potential to calculate the corrections to the LDA eigenvalues.

7 Appendix B

The algorithm used in the code is as follows:

(1) Construction of the Greens function in real space and imaginary time. (**polar.F90**)

$$G^{LDA}(\mathbf{r},\mathbf{r}';i\tau) = \begin{cases} i\sum_{nk}^{occ}\psi_{nk}(\mathbf{r})\psi_{nk}^{*}(\mathbf{r}')\exp(\epsilon_{n\mathbf{k}}\tau), & \tau < 0 \\ -i\sum_{nk}^{unocc}\psi_{nk}(\mathbf{r})\psi_{nk}^{*}(\mathbf{r}')\exp(\epsilon_{n\mathbf{k}}\tau), & \tau > 0 \end{cases} \tag{7}$$

(2) use this to calculate the RPA irreducible polarizability in real space and imaginary time,(**polar.F90**

$$P^{0}(\mathbf{r},\mathbf{r}';\omega) = -iG^{LDA}(\mathbf{r},\mathbf{r}';i\tau)\times G^{LDA}(\mathbf{r},\mathbf{r}';-i\tau) \tag{8}$$

(3) transform P^0 to reciprocal space and imaginary energy (polar.F90) and construct the dielectric matrix, (in **weps.F90** or **wepsio.F90** depending on run-time options)

$$\bar{\epsilon}(\mathbf{k},\mathbf{G},\mathbf{G}';i\omega) = \delta'_{\mathbf{GG}} - \frac{4\pi}{|\mathbf{k}+\mathbf{G}||\mathbf{k}+\mathbf{G}'|}P^{0}(\mathbf{k},\mathbf{G},\mathbf{G}';i\omega) \tag{9}$$

(4) then invert the dielectric matrix for each *k* point and each imaginary energy (in **weps.F90** or **wepsio.F90** depending on run-time options)

(5) calculate the screened Coulomb interaction in reciprocal space (**weps.F90**)

$$W(\mathbf{k},\mathbf{G},\mathbf{G}';i\omega) = \frac{4\pi}{|\mathbf{k}+\mathbf{G}||\mathbf{k}+\mathbf{G}'|}\times\bar{\epsilon}(\mathbf{k},\mathbf{G},\mathbf{G}';i\omega) \tag{10}$$

(6) Fourier transformation of *W* to real space and imaginary time, (in **weps.F90** or **wepsio.F90** depending on run-time options)

(7) calculation of the self-energy operator. This is composed of 3 terms, the bare exchange,(calculated in **selfx.F90**), a long-range term with multiplicative screening(*W*) and a short ranged term (**sigma.F90**).

$$\Sigma(\mathbf{r},\mathbf{r}';i\tau) = iG(\mathbf{r},\mathbf{r}';i\tau)W(\mathbf{r},\mathbf{r}';i\tau) \tag{11}$$

(8) evaluation of the expectation values $\langle\psi_{n\mathbf{k}}|\Sigma(i\tau)|\psi_{n\mathbf{k}}^{*}\rangle$ (**smatrel** subroutine in **sigma.F90**)

(9) Fourier transformation of the expectation values to imaginary energy. (in **smei.F90** or **smeiwgl.F90**, depending on run-time options)

The code proper ends here but two further steps are required to obtain data which can be compared to experiment:

(10) fitting of a model function to the expectation values of the self-energy, allowing analytic continuation onto the real energy axis.

(11) evaluation of the quasiparticle corrections to the LDA eigenvalues by first-order perturbation theory in $\langle\Sigma - V_{xc}^{LDA}\rangle$

8 Appendix C

The most time intensive routines in the serial code were parallelised as follows [5]:

8.1 gwst.F90

In the main routine changes were made to allocate necessary variables or arrays on all PEs. Much initialising data is read in from **gwst.ctr**, which is moved in the parallel code to a separate subroutine (**readin.F90**). Here derived datatypes are used so all logicals, integers and reads are distributed in one broadcast. There are also several arrays in **gwst.F90** which are read in from a data file **input** on PE 1 (rank 0) and are then broadcast to all PEs. Two additional subroutines using derived data types (**cast1.F90** and **cast2.F90**) are used to broadcast sets of variables as well.

8.2 polar.F90

Here new arrays are introduced(**greensnPE**,**greenspPE** and **p0kPE**) which hold part of the complete arrays(**greensn**,**greensp** and **p0k**. Each PE calls the subroutine **greensrt.F90** with **greensnPE** or**greenspPE** as output, i.e. each processor calculates one part of the whole array. Then an **MPI_GATHERV** call is used, and PE 1 writes the complete arrays to file. Here the new datatypes **GREENTYPE** and **FINALTYPE** are used to ensure the gather works correctly.

Then **p0kPE** is calculated on all PEs, gathered with **MP_GATHERV** into **p0k** and written to file by PE 1 with the use of the datatypes **P0KPETYPE** and **RECVTYPE**.

8.3 selfx.F90

Data is again required to be read in on PE 1 and then broadcast. A subroutine **sfq.F90** is called and returns the array **fqPE** on each processor. In a similar manner as before, this is a small part of the array **fq**, and the parts are gathered as before. It is also necessary to reorder the array in **ReOrder.F90** to obtain **fqfinal**.

8.4 sigma.F90

In this routine the arrays **unk**, **vcr** and **wR** are read in on PE 1 and broadcast to the other PEs. However it is probably faster to re-calculate **greens** as before rather than reading it from a file, and this is the method used by default.

The subroutines **smatrel.F90**, **sfq2.F90** are also divide up amongst the PEs for different parts of the arrays **greensnPE/fr/fqPE**. The first index of **fq** requires reordering as did **fqPE** in **selfx.F90**. Here the output has different dimensions, so the routine **sfq2.F90** is used rather than the original.

8.5 greensrt.F90

Here the scope of some of the arrays have been limited to the appropriate size on each PE only:

greens(nstart(rank+1),
green(begin:end,tmin:tmax),
greenc(begin:end,kstars%npos),
greenct(begin:end,kstars%npos),

where **nstart**, **begin** and **end** are calculated in **InitMPI.F90**. Hence only parts of the full arrays are calculated on each PE.

References

[1] W.Kohn and L.Sham. *Phys. Rev.*, 140:A1133, 1965.

[2] M.Hybertsen and S.Louie. *Phys. Rev. Lett.*, 55:1418, 1985.

[3] L.Hedin. *Phys. Rev.*, 139:A796, 1965.

[4] M.Reiger, L. Steinbeck, I. White, H. Rojas, and R.Godby. *Computer Physics Communications*, pages 2113–2228, 117 1999.

[5] E.Breitmoser. GW Space Code. EPCC report, June 2001.

[6] R.Thakur, W. Gropp, and E.Lusk. A case for using mpi's derived dataypes to improve i/o performance. *Proceedings of SC98: High Performance Networking and Computing*, November 1998.

Scott Fraser has completed three years of a 4 year MPhys degree in Theoretical Physics at the University of York, where the original serial code for this project was developed.

Lorna Smith and Elena Breitmoser were the long-suffering supervisors on this project.