I/O Optimization for BerkeleyGW's Absorption Executable

Gabe Dick, Nick Schoeck, and Graham Wood
Department of Computer Science
Kansas State University
Manhattan, KS 66502

Abstract

The topic of our project is to improve the I/O of BerkeleyGW's absorption executable implementing HDF5 file formatting to parallelize the reading and writing of data. We will move the I/O of the absorption executable into an external kernel for easier testing, and we will be basing our changes off of code from another executable that has already implemented HDF5 parallelization. Our changes will be evaluated based on an improvement of efficiency and time taken while the kernel is reading in data for the absorption executable.

Introduction

BerkeleyGW, as a part of the National Renewable Energy Lab in Colorado, is a long-standing project working on a set of code written in Fortran90 that uses the GW approximation to "...calculates the quasiparticle properties and the optical responses of a large variety of materials," (*About*). Instead of using an n-body calculation to find the effects of every electron on every other electron in a group, BerkeleyGW uses a mean-field calculation of the group of electrons to find the self-energy of the quasiparticle, a particle's interaction with the many-body system where it creates a cloud that disrupts its motion (*BerkeleyGW*, Deslippe), as a part of a Coulomb Interaction, a quantification of force between two electrically charged particles (*BerkeleyGW*, Deslippe). An example of the GW approximation, shown in Figure 1, would be as follows: there exists a uniform system of many electrons.

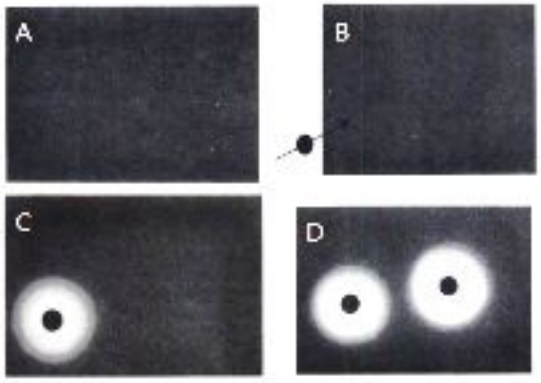

Figure 1: Pictures showing the process of the GW approximation (*GW*, Hull)

Then, an extra electron moves into the system and disrupts the others. The electrons that are pushed away create a hole in the system where the new electron moves into the space where the other electrons moves away. The new electron and the hole it created are what is called a

quasiparticle when the two are treated as one unit. Finally, when that process is repeated so there are two additional electrons, the interactions between those two is the Coulomb Interaction.

The BerkeleyGW code accomplishes these calculations through three main executables: epsilon, sigma, and absorption, also called BSE for Bethe-Salpeter equation. These three executables work together to do a portion of the calculations and then pass on its results to the next sections to continue doing calculations, as shown in Figure 2.

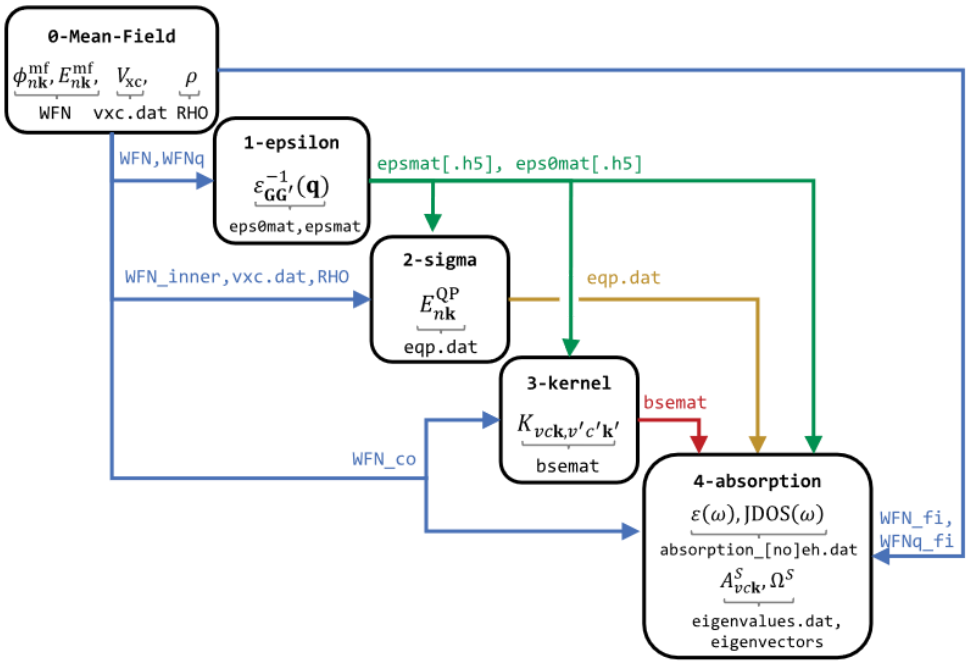

Figure 2: A diagram showing the relationship between BerkeleyGW's executables (*Tutorial*)

Epsilon takes the mean-field calculation and generates the dielectric matrix of the system and its inverse. Sigma takes those results to compute the self-energy corrections to the DFT eigenenergies using the GW approximation. Finally, absorption solves the BSE for a correlated electron-hole excitations.

Overall, BerkeleyGW is already highly parallelized in its calculations, can study systems up to 500 atoms, can be run on up to one million cores, and can perform at speeds up to 50 GB/s (hdf5, Vigil-Fowler). However, the I/O for the executables is serial which creates a huge bottleneck. For our project, we are focusing on the absorption section of the code and optimizing its input from the other executables.

Currently, the absorption code reads and interprets the data in serial using a Fortran binary, then distributes the data to do the calculations in parallel. Our approach to solving this issue is to use HDF5 files to read and write data instead of Fortran binaries and use it to parallelize the I/O of the code. HDF stands for "hierarchical data format", and HDF5 is a file format built to be extremely portable and scalable (*HDF5 Technologies*). It is designed to not

limit the size of files or the number of objects, to represent complex objects and metadata, and that it can be used on a variety of different machines. Additionally, HDF5 has features that are specifically beneficial to parallel systems, like synchronous reading and writing using its "parallel I/O driver… [that] reduces access times on parallel systems by reading/writing multiple data streams simultaneously" (*HDF5 Technologies*).

HDF5 utilizes a directory/file scheme for the hierarchical nature of its data, which makes sequential processing of data more efficient when compared to relational databases that are better suited to queries that rely on field matching. Additionally, HDF5 allows Groups which are used to organize data to point to the same object in storage. A situation can therefor exist where the path */A/k* and */B/m* can point to the same object. HDF5 also allows for the creation and use of complex or derived datatypes, which are a combination of standard datatypes. These are helpful for representing complex scientific data. All data, regardless of datatype, is stored in an array known as the dataspace which describes the layout of a dataset's elements. This means that, even if a dataset contains only one element, it is contained in an array. This is because dataspaces can have fixed or unlimited dimensions, which means that the dataspace can grow in size to allow for additional data to be added when needed. Dataspaces also allow for the selection of subsets of the dataset through coordinates.

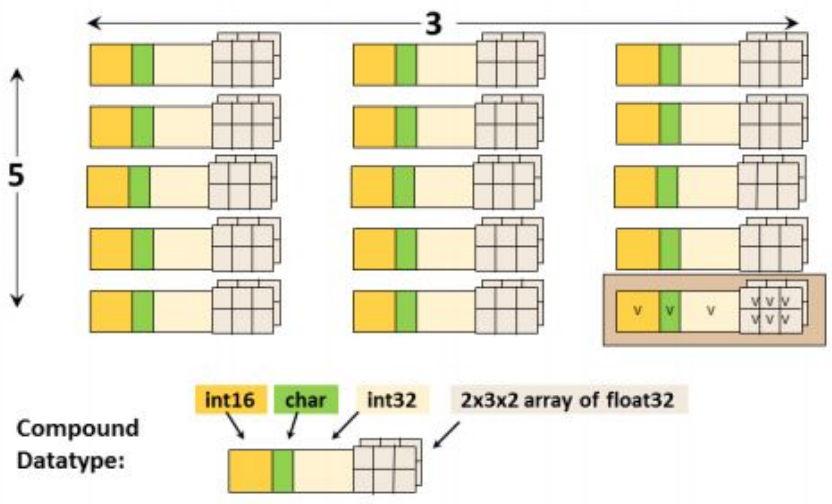

Figure 3: A visualization of HDF5's data storage (*HDF5 Technologies*)

Finally, HDF5 allows for the expression of properties and attributes for objects in HDF5. These are essentially types of metadata that either describe the data object in some way, or define how HDF5 handles the data object. Attributes, for example, are small, optional pieces of metadata that are considered sometimes as "name=value" pairs. They are like datasets in that they have a datatype and dataspace, but they are simple and cannot be derived, nor can they have properties. They are strictly meant to describe the nature or intended use of the data objects they are assigned to. As such, they do not allow for separate storage specification through attributes, and they also do not support partial I/O operations. Properties are characteristics or features for a data object and are not optional, but they have default values, so setting them is not always

necessary. For instance, there is a property called the data storage layout that determines how the data will be stored and is set to contiguous by default. This layout property can be changed, however, to be either chunked, or chunked and compressed, which would change how the data objects are stored in memory by HDF5.

The data being produced by BerkeleyGW is fairley large and complex, so the benefits that HDF5 offers will be helpful for making the absorption code's I/O more efficient and portable when compared to a Fortran binary.

This solution hasn't been implemented before because BerkeleyGW has been around since the 1990s, so some outside technologies used to improve the code haven't existed until recently. Additionally, we know this solution is worth considering and that it can be effective because BerkeleyGW has already been implementing HDF5 in other parts of the code, and it is our job for this project to take what has already been done in the code and restructure it to be used for the absorption code.

The rest of the paper will go over works that are related to BerkeleyGW and what they have done related to our solution, discuss more specifics on what we will be doing to change and implement the code, and how we will test and evaluate the changes we make to the code.

Related Works

A related work that we looked at is Quantum Espresso. Quantum Espresso is a electron-structure calculation and materials modeling application. It is similar to BerkeleyGW in that is is a highly parallelized mathematical application that is also implementing HDF5 for it's IO. Quantum Espresso is still in the process of implementing HDF5 which has lead us to mainly rely on the already implemented HDF5 in the Epsilon portion of BerkeleyGW, as mentioned above. The Epsilon code of BerkeleyGW had it's serial IO overhauled to utilize HDF5 this year and was a great resource because of its similarity to what we will have to implement. The Epsilon code parallelized the IO over the electron bands, whereas we will be parallelizing over the K-Points of the input.

Because of the serialized nature of the IO BerkeleyGW is currently experiencing, during an hour run of the GW approximation, 80% (48 minutes) of the time is spent on IO. This is a huge bottleneck because it only leaves 20% of the time to do the actual calculations. This greatly limits the size of calculations that the BerkeleyGW team is able to do because of how highly time is valued on such large supercomputers. Once the implementation of HDF5 is completed, we expect a speedup of 2x-4x. This will greatly reduce the amount of time spent on IO and allow the BerkeleyGW team to utilize more of their time with calculations. With this additional time they will be able to perform the GW approximation on larger systems of molecules and allow BerkeleyGW to scale better as it continues to grow.

Implementation

To determine what we are going to do to the absorption code, it is important to analyze and understand the HDF5 implementation in the epsilon code since we will be adapting from that. In Figure 4 we see the simple way in which k-points are divided up between nodes using a mod function. Where peinf%npes is the total number of processors in the calculation, , xct%nkpt_fi is the number of k-points/grid points (set by grid chosen by user), peinf%ikt tells you the number of k-points a given processors owns, and peinf%ik tells you the index in the global list of k-points of a given k-point that a certain processor owns.

```
SAFE_ALLOCATE(peinf%ik, (peinf%npes,peinf%nkpe))
SAFE_ALLOCATE(peinf%ikb, (peinf%nblocks))
SAFE_ALLOCATE(peinf%ivb, (peinf%nblocks))
SAFE_ALLOCATE(peinf%icb, (peinf%nblocks))
SAFE_ALLOCATE(peinf%ikt, (peinf%npes))
SAFE_ALLOCATE(peinf%ibt, (peinf%npes))

peinf%ik=0
peinf%ikt=0
peinf%ikb=0
peinf%icb=0
peinf%ivb=0
peinf%ibt=0
```

Figure 4: HDF5 implementation in epsilon

```
if (peinf%inode == 0) then
  call read_info(TRUNC(sFileName), iflavor)
  call read_kpoints(TRUNC(sFileName), kp)
  call read_gspace(TRUNC(sFileName), gvec)
  call read_symmetry(TRUNC(sFileName), syms)
  call read_crystal(TRUNC(sFileName), crys)
endif
```

Figure 5: Processing data in epsilon

The epsilon code first processes the contents of all the files through various functions. There are multiple functions similar to these which are responsible for reading in the header (read_hdf5_header_type) and gvectors (subroutine read_gspace). Once the data is processed, all the data is broadcasted to all the nodes using MPI BCAST and processing can begin.

Another important subroutine is the read_hdf5_bands_block in the epsilon code. This subroutine is responsible for reading in the HDF5 wavefunctions in parallel. We will be primarily focusing on adapting this function for the absorption code since we will be parallelizing over the k-points rather than the bands the epsilon code is parallelizing over.

```
! FHJ: loop thru indices on `eqp` grid, and then find the
!      corresponding labels on the `kp` grid
 do is=1,xct%nspin
   do irk=1,xct%nkpt_fi
     !irkq = index of q-pt on `kp` grid
     irk_kp = indxk(kg%indr(irk))

     ! If we read all bands, then both vb and cb are counted from bottom up
     if (read_all_bands_) then

       do ib_kp=1, kp%mnband
         eqp%evqp (ib_kp, irk, is) = kp%el  (ib_kp, irk_kp, is)
         eqp%evlda(ib_kp, irk, is) = kp%elda(ib_kp, irk_kp, is)
         eqp%ecqp (ib_kp, irk, is) = kp%el  (ib_kp, irk_kp, is)
         eqp%eclda(ib_kp, irk, is) = kp%elda(ib_kp, irk_kp, is)
       enddo
```

To contrast, the code in Figure 6 is an example of the big loops used currently used in the absorption code to perform IO. Here we can see we are looping over the indices of the bands. Because we are looping over all the bands in the function, it is a serial process. This is part of the code that we will be replacing with the parallelized HDF5 code.

Figure 6: do...while loop used in absorption I/O

Evaluation

To evaluate our changes to the absorption I/O, we will be moving the related code into an external kernel. We can then use the code in the kernel to test the changes we make on a smaller scale, rather than testing our changes with the entire calculation every time. That way, we can quickly and easily make changes, test it, then see how it affects the runtime. It is also simpler to get the kernel running on Beocat rather than dealing with the entire program. Then, we'll take the runtime of the I/O without changes, around 48 minutes of an hour long total runtime, and compare it with the runtime of the program after we've made changes.

Our initial solution will be taking the preexisting code we have from the epsilon executable and making the necessary changes for it to be applicable to our absorption code. From there, we can edit the way the kernel handles and organizes the data to see if there are any additional improvements that can be made. Finally, after we have gotten to a point in testing that shows satisfactory improvements, we can implement the I/O of the kernel back into the full program and test it there to ensure that it works and shows the improvements of the kernel for the whole calculation.

Conclusion

In conclusion, our goal working with BerkeleyGW is to improve the I/O of the absorption code by using HDF5 file formatting for its data, and making use of features HDF5 has to parallelize the reading and writing of that data. In the end, this will lessen or eliminate the bottleneck that the I/O currently creates when running calculations, and dealing with that bottleneck means the runtimes are faster, more calculations can be done in a smaller period of time, and that larger calculations can be done in a more reasonable time.

References

BerkeleyGW. (2017, October 30). *About*. Retrieved from https://berkeleygw.org/about/.

BerkeleyGW. (2017, October 30). *Tutorial*. Retrieved from https://berkeleygw.org/tutorial/.

Deslippe, J., Samsonidze, G., Strubbe, D. A., Jain, M., Cohen, M. L., & Louie, S. G. (2012). *BerkeleyGW: A massively parallel computer package for the calculation of the quasiparticleand optical properties of materials and nanostructures*. Computer Physics Communications, 183(6), 1269–1289. Retrieved from https://www.sciencedirect.com/science/article/pii/S0010465511003912?via%3Dihub

The HDF Group. (2014, February 13). *HDF5 Technologies*. Retrieved from https://support.hdfgroup.org/about/hdf_technologies.html.

Hull, Olivia. (2019) *The GW Appoximation*.

Vigil-Fowler, Derek. (2019) *hdf5*, BerkeleyGW.