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 was to improve the I/O of BerkeleyGW's Absorption executable implementing HDF5 file formatting to parallelize the reading and writing of data. We moved the I/O of the absorption executable into an external kernel for easier testing and simplicity, and we based our changes off of code from the Epsilon executable that has already implemented HDF5 parallelization.

Summary of Background

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).

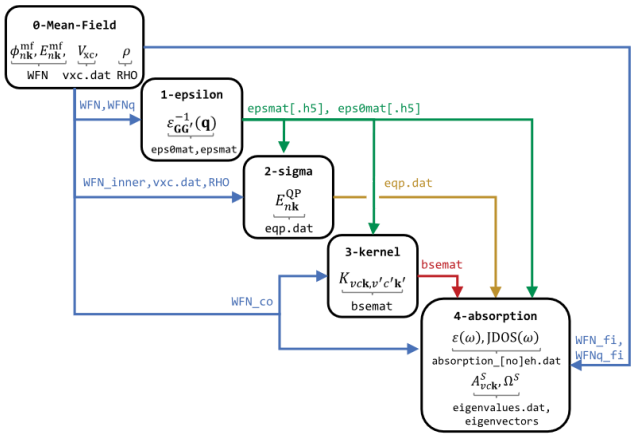

Figure 1: Diagram of the BerkeleyGW workflow.

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 1.

Overall, BerkeleyGW is already highly parallelized in its calculations, it is able to study systems of up to 500 atoms, it can already be run on up to one million cores, and it can perform at speeds of up to 50 GB/s (hdf5, Vigil-Fowler). However, the I/O for the executables is serial which creates a huge bottleneck for the whole program. For our project, we focused on the absorption section of the code to remove this bottleneck and allow the I/O to scale with the rest of the program for the future.

HDF5

For this project, we changed the way data was stored from Fortran binaries to HDF5, which is the 5th iteration of the "Hierarchical Data Format" file type. While technically the 5th iteration, in reality this version is a complete change to the system used, incorporating new file format and APIs. HDF5 allows for synchronous and dynamic reading and writing of data which can be stored as "objects" within the file. This is where the hierarchy comes in. Data can be either a simple type that comes with the format or a collection of one or more simple types to form a complex data type. Along with this, metadata for the types can be stored with these data objects. These key functionalities of HDF5 allow for the complex scientific data being used by BerkeleyGW to be both stored and accessed efficiently, while also allowing the data to be accessed concurrently for our purposes without the same concern for race conditions when multiple processes access the same file.

Related Works

BerkeleyGW is such a large and known project in their field, so there aren't many other projects that are comparable to it. Works that are comparable include QUANTUMESPRESSO, which is another set of code that does complicated scientific calculations, and Octopus, which is a different part of BerkeleyGW that does the mean-field calculations that are used as input. More comparable to our project, the Epsilon section of BerkeleyGW has already had it's serial I/O overhauled to utilize parallelism and HDF5 this year. It was a primary resource for us due to its similarity to what we had to implement. The Epsilon code parallelized the I/O over the electron bands, whereas we parallelized over the K-Points of the input.

Original Code

The original Absorption code that we modified was an original part of the program. This means that the code has had incremental changes in the last thirty years that has resulted in the code becoming cluttered and overly complex. Part of the challenge was determining what code was actually related to I/O and what code was related to other parts of the program. There was not a dedicated code base for the I/O so much of this code was scattered amongst the core program code and required a good understanding of the whole code base in order to understand how it worked. The code also included many of the data dependencies which added to the

complexity of the file and made readability increasingly difficult. By lifting parts of this code out of those files and into a kernel, we could organize the code we needed in a more logical manner and remove many of these data dependencies.

Implementation

Our goal was to take the Absorption section of the code and change it from reading in information in serial to being able to perform the I/O in parallel. Additionally, to improve the program's ability to quickly read in information, we were to implement HDF5, using it to change the data's filetype from Fortran binaries to HDF5.

As mentioned previously, the Epsilon section of the code had already been updated from serial to parallel and from Fortran binaries to HDF5. This allowed us to use the Epsilon code as the basis for the changes that we were making to the Absorption code. The calculations in the Epsilon code were done using the electron bands of the mean-field that is initially given to the code. The Absorption code's calculations focus on k-points, which is another part of the calculations, so the Epsilon code used k-points but did not focus on them. This meant it was possibly for us to swap the usage of bands and k-points in the Epsilon code with some additional changes to ensure the math was still correct, and then work off of that to implement the use of HDF5 and the parallel I/O.

Another part of our goal was to make a kernel program out of the Absorption code. This would allow us to make changes and test the program more easily than if we were using the entirety of the code. However, this proved to be a lot more difficult than originally thought. The original Absorption code had a lot of calls to functions or variables in other files, so all of those had to be removed and replaced with functions added into the code or hard-coded information that needs to be changed manually.

When we first received the Epsilon code, we started making the swap from bands to k-points, editing the way the processes were distributed, and adding in code from other files that was still necessary. We quickly ran into an issue where we were unsure what mathematical information we could change and what information depended on other calculations that we did not know about. So, we made the changes we could then let our contact make sure we didn't mess with anything we weren't supposed to and make his own changes for parts of code that involved the calculations that were above our understanding. After that, we continued to edit the kernel to get it running on Beocat.

For the implementation of the code, the kernel works using two main functions: distrib, and read_hdf5_kpoints_block. The kernel first initializes everything it needs for MPI and HDF5, then allocates some memory for the header and reads it in serially since it is a small enough amount of information that it doesn't need to be done in parallel. Then, the distrib subroutine is

called. Each process goes through the list of k-points and adds any unowned k-points to it's list of k-points it owns. It then accesses a global list and marks the k-point it just grabbed as owned. After the distrib subroutine returns, the program allocates space for the data that's about to be read in, calls a couple subroutines to create the HDF5 object. Once those objects are created, read_hdf5_kpoints_block is called. Each process then goes through their list of owned k-points, creates the HDF5 object for each individual k-point and reads in the data. HDF5 allows for each process to access different objects in the same dataset due to it's filesystem-like objects accommodating the parallel processes. Each of the processes do this until they get through their list and then deallocate any local space that was taken. Finally, when all the other processes are done, the root process deallocates the space that was initially used and finalizes MPI and HDF5 before ending the program.

Results:

Unfortunately, we were unable to get any meaningful results for our project. When we first got the kernel to compile on Beocat, we were able to run it on small data samples, but these results don't mean much since the way the program runs on smaller datasets is different from how it would on larger ones, and we know that the real bottleneck occurs in those larger datasets. However, when we attempted to run the kernel on a larger set of data, we had runtime errors with the dataset. Within the timeframe of this report**,** we were unable to find a solution to the errors, thereby unable to test the kernel on meaningful data and unable to make changes to the kernel based on those tests.

Issues:

The main difficulty for us while working on the project was how closely the code is tied to the chemistry and physics necessary for BerkeleyGW. We needed help from our contact several times to simply get to a point where we could begin to edit or optimize the I/O without risking reading in entirely wrong information or changing something that was important to the calculations. Along with this, many of the variable or subroutine names were vague or unrecognizable to us due to being a couple of random letters that represented something in the calculations. Even though there is documentation and comments throughout the code, we consistently had to look up what variables meant or ask our contact, which made working with the code more difficult for us. The complexity of the code caused us to need our contact's help repeatedly to get past roadblocks that required a deeper understanding of BerkeleyGW that we simply didn't possess. This was compounded by relying on Zoom and emails to collaborate with our contact, which led to imprecise and delayed communications. Our initial need for explanation and questions, combined with delays and the inability to meet in person, required us to simply do our best to interpret what was happening while keeping track of the questions we had, all the while hoping that it wouldn't put us too far behind.

Additionally, the simple need to learn the basics of a new language, Fortran90, as well as the HDF5 file system added more overhead to the project than we originally thought. On their own, learning Fortran90 and HDF5 wasn't that difficult, and learning them both together was reasonably challenging as well. However, learning both of those on top of needing to learn a lot about how BerkeleyGW works and about how the preexisting code works, we had to commit a lot of extra time just to learn everything we needed before we could even edit the code. This, along with the other issues, made our progress slow-going and lead to us not being able to get any usable results within the time frame of the project.

What we learned?

Working on BerkelyGW has been a learning experience and a really valuable opportunity for exploring real life implementation of highly parallelized concurrent systems. We learned a lot about what it takes to work on large complex applications and the technologies that go into them. Specifically, we learned how MPI is applied in utilized in very large applications that operate on tens of thousands to millions of cores at a time. We also had the opportunity to gain knowledge of Fortran90 and HDF5 and how to implement them on large concurrent systems. On the other hand, we learned the difficulty that comes with working on complex systems that are reliant on high level math and science we don't have an understanding of, while working with others remotely and the communication barriers that go along with that. In the end, this project was a great experience, an opportunity to learn more about concurrency and high power computing, but the project, the complex, preexisting code, and the high level math and science it involved was too much for us to get an understanding of in the time we had.

References

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

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

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