Porting the MPI-Only BerkeleyGW Materials Science Code to Accelerators/Many-Core

Jack Deslippe (LBNL)

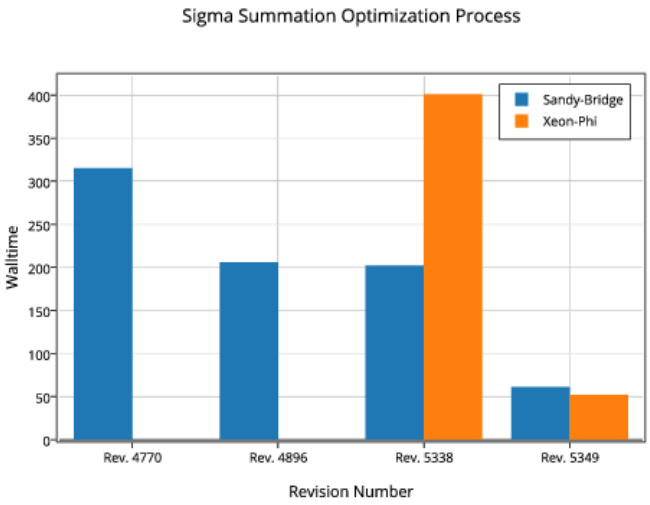
BerkeleyGW 1.0 is a “massively parallel” (MPI Only) package for computing excited-state properties of materials with the conventional GW/Bethe-Salpeter-Equation method (typically such calculations sit on top of standard Density Functional Theory (DFT) calculations with codes like Quantum ESPRESSO, PARATEC, SIESTA etc.). GW calculations on small to medium sized systems typically scale as N^3, but, for large systems (many hundreds of atoms), an N^4 DGEMM like term dominates. Both, the N^3 computation and the N^4 computation have been highly parallelized with MPI [1] in the BerkeleyGW package.

However, the MPI Only programming model has begun to fail our users for several reasons. Firstly, users want to study ever larger systems that require more memory. While BerkeleyGW generally does a good job at distributing critical arrays over MPI tasks, like all codes, there is memory overhead/duplication per MPI task. In some cases, users are resorting to running on one MPI task per node on HPC systems in order to maximize the memory available per MPI task. This is obviously not ideal in an MPI-only code, as it throws away much of the compute capability of the node. Secondly, our users are gaining access to machines with GPU accelerators and Xeon Phi co-processors (such as Stampede at TACC). These platforms have significantly more hardware threads per node and also importantly rely on code vectorizability to perform SIMD like operations.

Therefore, for the BerkeleyGW 1.1 release, we set out to add layers of on-node parallelism to the code. We wanted to do so in a way that would identify enough on-node parallelism to effectively use GPU and other co-processors, a level of parallelism beyond what is needed on traditional CPU based compute nodes.

BerkeleyGW has two main executables. The first, Epsilon.x, is highly dependent on math libraries: 3D FFTs from FFTW and ZGEMM in BLAS in particular. For this executable, we were able to find enough on-node parallelism by simply utilizing the threaded and accelerated libraries. In particular, in our GPU exploratory branch, we use CUBLAS, CUFFT and are exploring MAGMA. In the OpenMP threaded code, we use OpenMP math libraries for FFTW and BLAS in MKL and other implementations.

The second main executable in the BerkeleyGW code is Sigma.x. The critical regions in the code depend highly on hand tuned reduction loops that compute summations over large matrices, producing a single number as output. It is informative to look at the steps we took to optimize this kernel for accelerators and many-core. For illustration, I will discuss the performance optimization process on the Intel Xeon Phi (Knights Corner) such as is installed on Stampede at TACC. Various steps in the optimization process are shown in the figure below, where we compare the walltime of an example problem running on a single dual Sandy Bridge (16 cores) host node or a single Xeon Phi card.



The optimization process occurred over a series of revisions:

Rev. 4770 - Initial MPI-Only Code. The test problem cannot fit into memory on the single Xeon Phi when using any significant fraction of the available cores as MPI tasks. We therefore, do not report timing numbers for the Xeon-Phi.

Rev. 4896 - Refactored code to have the following structure: Outer loop (thousands of iterations) for MPI, inner loop (thousands of iterations) for OpenMP, and large innermost loop for vectorization. We gain some performance due to refactoring and memory locality improvements.

Rev. 5338 - Addition of OpenMP pragmas (using OMP reduction clauses). The example problem can now fit on the Xeon-Phi and utilize all cores/hardware-threads using a combination of MPI tasks and OpenMP threads. The optimal performance occurs with 10 MPI tasks and 12 OpenMP threads per task on the Xeon Phi.

Rev. 5349 - Refactoring inner loop body to ensure vectorization. We removed cycle statements intended to save work during serial execution but prevent compiler from vectorizing. In some cases we split loops, improved flow and reduced flops.

This BerkeleyGW kernel and example is, in many ways, an ideal case for the Xeon Phi: reduction loops can benefit from the increased memory bandwidth and the code was able to be restructured in a way that left very long inner loops (1000-10,000 iterations), which is ideal for vectorization. In general, relying heavily on optimized math libraries and small vectorizable kernels has allowed us to successfully identify and exploit on-node parallelism.

Citations:

1. Deslippe, Jack, et al. "BerkeleyGW: A massively parallel computer package for the calculation of the quasiparticle and optical properties of materials and nanostructures." *Computer Physics Communications* 183.6 (2012): 1269-1289.