**2024 School on Electron-Phonon Physics, Many-Body Perturbation Theory, and Computational Workflows**

# BerkeleyGW Hackathon

**Hands-on session (Sun.1)**

Hands-on based on BerkeleyGW's computational kernels

## BerkeleyGW Hackathon Overview

In this tutorial you will explore some of the BerkeleyGW's most intensive computational kernels to learn their basic implementation characteristics, parallelization strategies, parallel performance and scaling of computational cost with respect to system size. The modules involved in the GW-BSE workflow implemented in BerkeleyGW are given in Fig. 1.

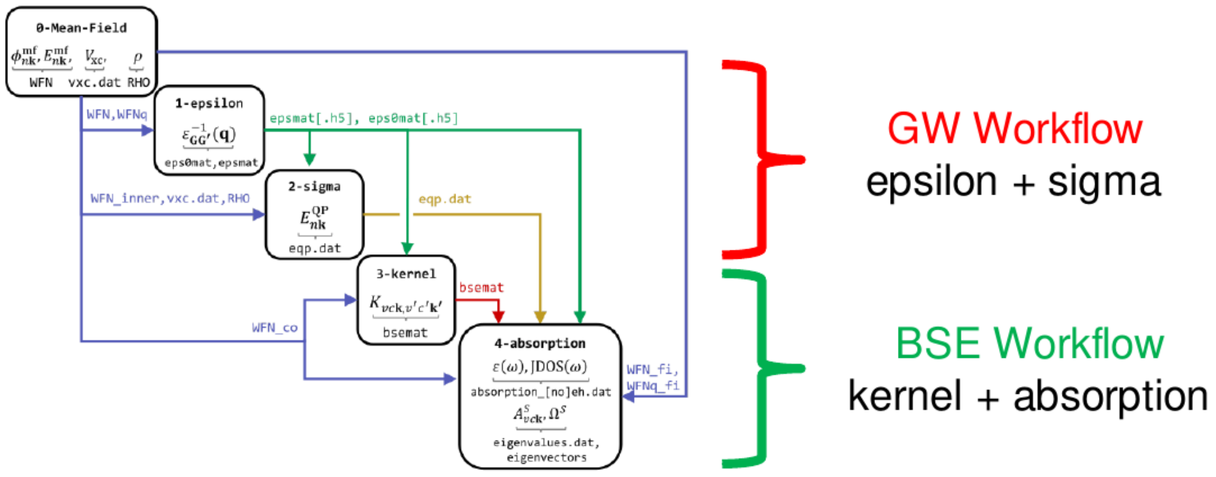

Fig. 1 Shown is the typical workflow, including the required input and output files, for the GW-BSE calculations with BerkeleyGW.

Each of the modules given in the workflow scheme of Fig 1 has its own computational intensive kernels and algorithmic motifs, which are listed below here (excluding step 0, which is specific for the DFT code used to run the ground state calculation):

0. **epsilon** module computes the polarizability function, its frequency dependence and generate the dielectric matrix and its inverse $\epsilon^{-1}$. Basic computational kernels:
    - Node local Fast Fourier transformations (FFT).
    - Large distributed matrix multiplication over short and fat matrices.
    - Low rank approximations/diagonalization.
    - LU decomposition, matrix inversion.
1. **sigma** module computes the screen coulomb interaction $W$ necessary to compute the self-energy $\Sigma$ and solves the GW Dyson's equation to compute the quasiparticle energies of the system. Basic computational kernels:

- Node local Fast Fourier transformations (FFT) ZGEMM.
- Tensor-like contractions
- Large data reductions

2. **kernel** module computes the direct and exchange BSE kernel matrix elements on a coarse k-point grid. Basic computational kernels:
   - Node local Fast Fourier transformations (FFT)
   - Local ZGEMM and dotproducs.
   - Distributed tensor algebra.
3. **absorption** module interpolates the BSE kernel matrix elements onto a fine k-point grid, diagonalize the BSE Hamiltonian, and compute the optical absorption spectrum. Basic computational kernels:
   - Tensor interpolation.
   - Large distributed direct or iterative matrix diagonalization.

### Setup Instructions

This tutorial will use either the CPU or GPU partition of Frontera. First `ssh` into the system:

```
$ ssh -X USERNAME@frontera.tacc.utexas.edu
```

Then copy and extract the tutorial folder:

```
cd $SCRATCH
cp /work2/05193/sabyadk/stampede3/EPWSchool2024/tutorials/Sun.1.BGW_Hack.DelBen.tar .
tar -xvf Sun.1.BGW_Hack.DelBen.tar
```

Each of the subfolder in `BGW_Hackathon_EPW24` folder, namely:

```
EIGENSOLVERS-parallel
EPSILON-parallel
SIGMA-serial
wannier-interpolation-GW
```

contains all material to perform the exercise. Enter each of the folder (**not necessary in the order given above**) and follow the instruction in the `README.md` files.

## Exercise `EIGENSOLVERS-parallel`

In this exercise you'll familiarize with diagonalization libraries which are mostly used in the `parabands` and `absorption` modules of BerkeleyGW. In the `EIGENSOLVERS-parallel` folder you find a collection of miniapps which build up an Hermitian matrix and diagonalize it to test the performance of various libraries. You can also read the matrix to diagonalize from a file (binary format).

- In this exercise we will use Scalapack (pzheevd) and ELPA eigensolvers.
- Have a look to the `arch.mk` file, which scalapack implementation are we using? Which ELPA implementation are we using?

- To build the miniapps type `make scalapack elpa`. What's the name of the generated executables?
- Edit the sbatch script `run_job` to run on Frontera using the command `sbatch run_job`. As an example you can use:

  ```
  export OMP_NUM_THREADS=2
  # for scalapack
  ibrun -np  4  ./pzheevd.ex   4000  2  2  64  > scalapack_4MPI.out
  # for ELPA
  ibrun -np  4  ./elpa.ex      4000  2  2  64  > elpa_4MPI.out
  ```

  Where:

  - 4000 is the number of row/col of the distributed matrix (Hermitian filled with random number)
  - 2 is the number of row processes of the 2D block cyclic data layout
  - 2 is the number of column processes of the 2D block cyclic data layout
  - 64 is the block size of the 2D block cyclic data layout

  Note that the number of row processes times the number of column processes has to be equal to the total number of MPI tasks (`-np` argument).

**Scaling of Computational Cost with System Size**

For both scalapack and ELPA, run a series of diagonalization for matrices of increasing size between 2000 to 5000 using (6-8 points) a single MPI task and two OMP threads (modify `run_job` and submit using `sbatch run_job`):

```
export OMP_NUM_THREADS=2
ibrun -np  1  ./pzheevd.ex   XXX  1  1  64 > scalapack_XXX_4MPI.out
# with XXX between 2000 and 5000
```

Hint: you can run the calculations for all XXX in a single `run_job`.

- For both scalapack and ELPA, plot the time vs matrix size (time is in seconds, look for `PZHEEVD time:` and `ELPA time:`). Which function would you use to fit the data?
- From the fitted data extract the increase of computational cost (time to solution) as a function of the matrix size in $O(N^x)$ notation.
- Estimate how long it would take to diagonalize a matrix of size 50k using a single MPI task (two OMP threads) using Scalapack and ELPA.

**Strong Scaling Performance**

For both scalapack and ELPA and a matrix of fixed size 5000, run a series of diagonalization increasing the number of MPI tasks for each calculation (two OMP threads per task):

```
export OMP_NUM_THREADS=2`
ibrun -np  X  ./pzheevd.ex   5000  Y  Z  64
# X = 1,2,4,6,8 and 12 ; Y*Z=X
```

modify run_job and submit using sbatch run_job.

- Plot the parallel strong scaling for the two libraries. What should be the ideal time to solution when running with 4 and 12 MPI tasks? What's the parallel efficiency for the 4 and 12 MPI tasks run?
- Advance: Using the results from the previous exercises perform a set of calculations to measure the weak scaling for the two libraries.

## Exercise EPSILON-parallel

The small program given in this exercise reproduces one of the most computationally intensive kernel of the epsilon module named chi_summation. This kernel compute the (static) polarizability which is at the core level implemented as a large distributed matrix multiplication running in parallel.

- The goal of this kernel is to simulate the matrix operations on a per MPI task level without running the actual global problem size using nproc MPI tasks. You can even choose to run this on 1 MPI tasks, but the MPI communication pattern will not be present.
- The kernel is simulating a large distributed matrix multiplication between the tall and skinny matrices: basically $M^{\mathsf{T}}xM$.
- The total number of columns of the distributed matrices is given by nmtx parameter in input, 94,317 for the actual input.
- The total number of row of the distributed matrices is given by (Nvalence bands) * (Nbands total - (Nvalence bands)) parameters in input, 54,590,600 for the actual input.
- The matrix is distributed locally as if the calculation is run using nproc simulated MPI tasks with ScaLAPACK 2D grid of size Nprow x Npcol, 9216 MPI tasks distributed as 96x96 for the actual input.
- Number of repetition gives how many iteration of the original algorithm should be performed, for the real run Number of repetition is equal to nproc simulated. If statistics or average are not important, you can set this value to 1.
- For each iteration there is a ZGEMM call plus some non-blocking point to point communication, for the the actual example the sizes are N = 986 ; M = 986 ; K = 5924 with K inner dimension, N, M, K are given in output when running the calculation

Follow these instructions to run the exercise:

- Have a look to the Makefile, what compiler is used? Which BLAS library is used? There are references to ScaLAPACK, why is this required?
- Type make to build the kernel epsilon_kernel.ex
- Have a look to the input structure/parameters input_Si1000_12Ry:

```
1      ! Nspin x Nkpoints
1996   ! Nvalence bands
29346  ! Nbands total
94617  ! nmtx (distributed matrix size)
```

```
9216   ! nproc simulated
96     ! Nprow in ScaLAPACK layout (Nprow * Npcol  = Nproc)
96     ! Npcol in ScaLAPACK layout (Nprow * Npcol  = Nproc)
25     ! Number of repetition eventually will be equal to Nproc simulated
```

- Have a look at the sbatch script, how many MPI ranks and OpenMP threads are we using? How many CPU cores in total? To run the the job on Frontera use `sbatch run_job`
- Have a look to the output file `epsilon_8MPI_1OMP.out`, what's the performance of the ZGEMM operation? What's the formula that gives you the total FLOPs from N,M,K parameters?
- Repeat the calculation by `export OMP_NUM_THREADS=2` and `export OMP_NUM_THREADS=4`. What's changing? What is this variable controlling?
- Have a look to the `timings_zgemm_tot.dat` file, is the workload well load balanced?
- To simulate load imbalance edit the `epsilon_kernel.f90` search for `simulate_load_imbalance = .FALSE.` and replace FALSE with TRUE. Recompile by typing `make` and run rerun using `export OMP_NUM_THREADS=2`. Have a look to the `timings_zgemm_tot.dat` file, how does it compare with the previous case? What's the difference between ZGEMM and Cycle time?
- Advance: Rewrite the communication scheme to use `MPI_Reduce`.

## Exercise `SIGMA-serial`

The small program given in this exercise (`gppKer.f90`) reproduces one of the most computationally intensive kernel of the `sigma` module named `GPP`. This kernel compute matrix elements of the self-energy operator using the generalized plasmon pole model (GPP) which is at the core level implemented tensor-like contraction followed by a few large data reductions. These matrix elements are used to solve the GW Dyson's equation to obtain a set of GW quasiparticle energies ($E_{\text{qp}}$). The miniapp given here runs in serial (one MPI rank) but simulates the core computation for each MPI rank as if running in parallel.

The goal of this exercise is to familiarize with the OpenMP programming model to parallelize the `GPP` kernel both for multicore (CPU) and GPU offload. Find more info on OpenMP proframming model here: `https://www.openmp.org/specifications/`.

- To run and compile on GPU nodes you need to create an interactive session, use the script:

  ```
  ./create_interactive.sh
  ```

- Type `make` to build the miniapp `gppKerFort.ex`.
- Use the instruction below to run the miniapp (NOTE: the number to specify after -c should be equal `OMP_NUM_THREADS` value):

  ```
  export OMP_NUM_THREADS=1
  srun -n 1 -c 1 stdbuf -o 0 ./gppKerFort.ex 1300 100 5000 128
  ```

  The meaning of the input parameters given above is:

```
        1300   ! Number of bands to sum over
         100   ! Number of valence bands
        5000   ! Number of G-vectors up to the screened coulomb cutoff
         128   ! Number of MPI tasks simulated
```

running with the above parameters should give results that match with:

```
 Answer-ch[1]:  (-2063.283631129852,-2027.720486815221)
 Answer-ch[2]:  (-2263.583035797370,-1952.093512376878)
 Answer-ch[3]:  (-2463.613155493892,-1859.538723245008)

 Answer-sx[1]:  (1.394228718523581,-1.393292504701358)
 Answer-sx[2]:  (1.324654706418108,-1.323809581605180)
 Answer-sx[3]:  (1.260162247032606,-1.259397397032201)
```

- Once your kernel is parallelized you may try increasing the problem size to better assess performance, you can use for example for a larger calculation using 10 OMP threads:

  ```
  export OMP_NUM_THREADS=16
  srun -n 1 -c 16 stdbuf -o 0 ./gppKerFort.ex 6000 500 25000 1024
  ```

  By running with these parameters the output results should match:

  ```
   Answer-ch[1]:  (-1151.969785008755,-2563.463943403231)
   Answer-ch[2]:  (-1263.308618426997,-2575.306828443616)
   Answer-ch[3]:  (-1374.767851836923,-2582.147626593300)

   Answer-sx[1]:  (1.137392818924680,-1.137164406737877)
   Answer-sx[2]:  (1.105826254117275,-1.105610345972890)
   Answer-sx[3]:  (1.075555982115415,-1.075351733871395)
  ```

When you run the miniapp it will execute 3 times the same GPP computational kernel, the first is considered reference, running serial on a single core, the second execution is intended for the multicore implementation (you'll need to work on the gppKer_omp_cpu subroutine) and the third execution will be run for the GPU kernel (you'll need to work on the gppKer_omp_target subroutine). The total run time (`Runtime:`) will be printed at the end of each execution and it represents the combination of the time spent in the computation of the self energy sx (exchange) and ch (correlation). You will use this value to track the improvements of your implementation, the smaller the time the best your implementation. Of course you need to make sure your miniapp reproduces the correct results given above.

You now have all the basic information to compile and run the miniapp, therefore you can proceed with the goal of this exercise which is to implement multicore parallelization and GPU offload for the miniapp using the OpenMP programming model.

- The file to modify with the source code is `gppKer.f90`. As mentioned above you'll need to work on the gppKer_omp_cpu subroutine for the CPU multicore OpenMP implementation and `gppKer_omp_target` subroutine for the GPU offloaded implementation using OpenMP-target.
- At the beginning of the program (in `gppKer.f90`) there are three flags that can be modified:

```
run_ref       = .true.
run_multi_cpu = .true.
run_gpu       = .true.
```

If you want to focus on only one of the implementation, you can switch the other kernels' flag into `.false.` to avid their execution. For example you can set `run_ref = .false.` to avoid the execution of the serial (one core) reference execution.

- You can test the parallel efficiency of your multicore implementation by running `gppKerFort.ex` by exporting an increasing number of `OMP_NUM_THREADS`:

  ```
  export OMP_NUM_THREADS=XXX`
  ibrun -np 1 ./gppKerFort.ex 6000 500 25000 1024`
  # With XXX=1, 2, 4, 8 and 16
  ```

- The GPU performance will be considered with respect to a single GPU.
- Useful OpenMP directives (multicore implementation):
  - Create a parallel do over threads:

    ```
    !$OMP PARALLEL DO
    !$OMP END PARALLEL DO
    ```

  - Variable attirbute:

    ```
    !$OMP private()
    !$OMP shared()
    ```

  - Perform the reduction of the variable `var` at the end of the parallel region:

    ```
    !$OMP reduction(+: var)
    ```

- Implement GPU support for the kernel by using OpenMP-target programming models. Measure speedup compared to CPU implementation. OpenMP-target is syntactically similar to OpenMP with extra features to enable the offload to GPU. OpenMP-target directives start with

  ```
  !$OMP target
  ```

  Useful OpenMP-target directives for this exercise are:

  - Copy the elements listed in the parenthesis from the CPU to the GPU and delete them from GPU at the end of the data region.

    ```
    !$OMP TARGET data map(to: )
    !$OMP END TARGET data
    ```

  - Distribute the loops over threads and threads blocks on the GPU:

    ```
    !$OMP TARGET teams loop
    !$OMP END TARGET teams loop
    ```

## Exercise `wannier-interpolation-GW`

In this exercise you will use Wannier functions to interpolate $GW$ band structures, which is a more general method than `inteqp` distributed in BerkeleyGW, in fact `inteqp` only does insulators and graphene, but not metals. You'll use use `Wannier90` to interpolate the GW band structure. Enter the `wannier-interpolation-GW` folder, you'll find the following set of subfolders:

```
1-scf
2.1-wfn-wannier
2.2-wfn-sigma
2.3-wfnq-sigma
3-epsilon
4.1-g0w0
4.2-g0w0-wannier
```

The material used for the exercise is silicon. Access each folder in numerical order and follow the instructions:

- Access `1-scf` and execute `./01-calculate_scf.run` to run the SCF with QE.
- Access `2.1-wfn-wannier`. Take a look at the `bands.in` file, the KPOINTS card should be missing. You'll use `Wannier90`'s `kmesh.pl` to generate a full k-grid

  ```
  W90PATH/utility/kmesh.pl 4 4 4 >> bands.in
  ```

  After this run the script `./01-cp-files.sh` and `./02-calculate_wfn.run` to run the wave-function calculation.
- Open `Si.win` file and see how the projections, atom positions, lattice vectors, `mp_grid` been set up properly and consistently with `bands.in`. If yes, run `Wannier90`:

  ```
  ./03-wannier90.run
  ```
- Open the `Si.wout` file to check the disentanglement outcome and the final spread of the Wannier functions.
- Access `2.2-wfn-sigma/` folder to run the QE calculations to generate the WFN necessary for BerkeleyGW:

  ```
  ./01-cp-files.sh
  ./02-calculate_wfn.run
  ./03-convert_wfn.run
  ```
- Access `2.3-wfnq-sigma/` folder to run the QE calculations to generate the WFNq necessary for BerkeleyGW:

  ```
  ./01-cp-files.sh
  ./02-calculate_wfn.run
  ./03-convert_wfn.run
  ```
- Access `3-epsilon/` folder to run the `epsilon` calculation for the dielectric function using BerkeleyGW:

```
./01-link-files.sh
./02-run_epsilon.run
```

- Access `4.1-g0w0/` and execute:

  ```
  ./01-link-files.sh
  ./02-run_sigma.run
  ```

  **IMPORTANT:** The above script runs the `sigma` calculation in the background. This is because this calculation can be long ($\tilde{3}0$ mins) since we are computing the full k-BZ! This can be relatively easily avoided, and will be left as a Hackathon exercise/homework. (See at the bottom of this exercise the paragraph labelled as **extra**). To track the progress: use:

  ```
  tail -f sigma.out
  ```

  You can stop tracking anytime with: `Ctrl+C`, and the `sigma` calculation will continue running in the background. **If you feel the run is taking too long**, you can terminate your `sigma` run and use what we have pre computed:

  ```
  cp   sigma.out.ref   sigma.out
  cp   sigma_hp.out.ref   sigma_hp.out
  ```

  After the calculation is finished run:

  ```
  ./03-run-sig2wan.run
  ```

  Open the generated file `Si.eqp1.eig`, this is the GW `eqp1` energies written in the `Wannier90.eig` file format.

- Access `4.2-g0w0-wannier/` folder, take a look at the file `01-link-files.sh`, note that the GW eigenvalues will be linked. Run the link command:

  ```
  ./01-link-files.sh
  ```

  Restart `Wannier90` without redoing the Wannierization. Restart will be done with the check-point file `.chk`. Run the following command to interpolate the G0W0 band structure:

  ```
  ./02-wannier90.run
  ```

**extra**: It is straightforward to get rid of the full-BZ sigma calculation, by using the irreducible BZ instead. In that case, one need a script to unfold the k-BZ and write a new `sigma_hp.log` file in the full k-BZ. This will be left as a task if you have extra time in this hackathon or as a homework.