Parallelization of algorithm for Coulomb and Exchange Interaction: *Static (block) vs Dynamic Distribution*

Md Rezaul Karim Nishat
Dept. of ECE
Southern Illinois University Carbondale
Carbondale, IL
rezanishat@siu.edu

***Abstract*—In this paper, we computationally evaluate the atomistic potential as a vital step to find coulomb and exchange interaction energy using full configuration interaction (FCI) method where a parallelized algorithm has been used. A comparison is shown between dynamic and block distribution of atoms into processors as an endeavor of developing existing code for faster execution.**

Keywords— MPI; Block distribution; Dynamic distribution; Coulomb potential;

I. INTRODUCTION

From generic perspective it is important to calculate coulomb interaction energy, because of its significant value (i.e. ~ 60 meV), to get the right bandgap and excited states . Although very often we neglect the effect of exchange splitting due to its very small value i.e. around 200 μ eV in our calculation, the new era of different applications of quantum devices have made this characteristics very important. Single photon emitter (SPE) and entangled photon-pair generator (EPPG) are tremendously important in the field of quantum key distribution (QKD), quantum computation, single photon quantum memory, metrology etc. [1]. From the simulation point of view it is time consuming to evaluate atomistic coulomb potential which is an important part to find the interaction energy. So two different parallelization approach have been compared to find optimized way to compute potential.

II. SIMULATION FLOWCHART

A parallelized algorithm is used to find the interaction energies. Now, calculating coulomb interaction energy and exchange correlation energy has two steps:

(i) Calculating single particle Eigen energy and Eigen function.

(ii) Calculating coulomb and exchange integrals from previously found quantities.

To do these calculations we need a) single electron energies, b) electron/hole wave functions and c) atom positions. Now we get all these data as output of another simulation tool NEMO3D. Fig. 1. Shows the flowchart of the FCI method to calculate the interaction energies.

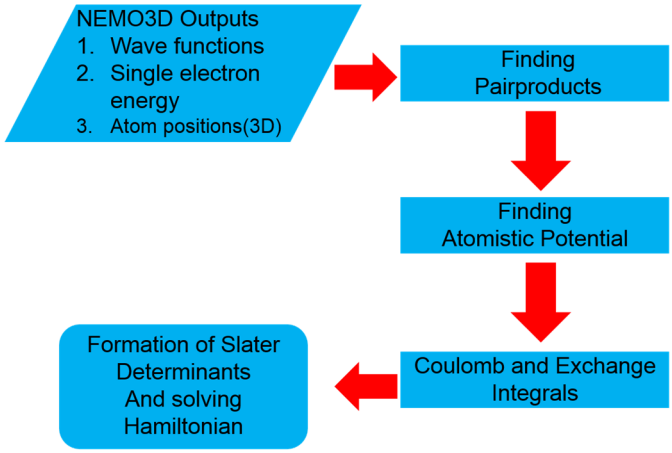

Fig. 1. FCI flowchar for coulomb and exchange interaction energy

Now, finding atomistic coulomb potential is the most time consuming step of this whole process. Because if we look at Fig. 2 which is a crystallographic image of a 2X2X2 nm GaN quantum dot grown on a-plane, there are 512 atoms. To compute the potential for a single atom we need to consider effect from all the 511 other atoms. For the interaction between an electron in state $\psi_i^e(\mathrm{r})$ and a hole in state $\psi_j^h(\mathrm{r})$, the matrix element of Coulomb interaction in electron-hole Hamiltonian is [2]

$$E_{ij}^c = \langle \mathrm{eh} | J | \mathrm{eh} \rangle = \frac{e^2}{4\pi\varepsilon_o \varepsilon_r} \iint \frac{|\psi_i^e(\mathrm{r}_e)|^2 \, |\psi_j^h(\mathrm{r}_h)|^2}{|\mathrm{r}_e - \mathrm{r}_h|} \, d\mathrm{r}_e d\mathrm{r}_h$$

and the exchange interaction,

$$E_{ij}^x = \langle eh | K | eh \rangle$$

$$= \frac{e^2}{4\pi\varepsilon_o \varepsilon_r} \iint \frac{\psi_i^e(r_e)^* \, \psi_j^h(r_h)^* \, \psi_i^e(r_h) \, \psi_j^h(r_e)}{|r_e - r_h|} \, dr_e dr_h$$

Which clearly shows the distance vector between two atoms.

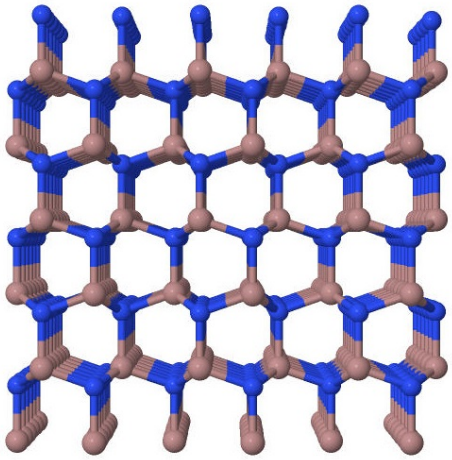

Fig. 2. Atomistic view of GaN a-plane quantum dot

III. PARALLELIZATION ALGORITHM

In the existing code the atoms have been distributed among the processors using block distribution. In Fig. 3 we can see that how *startatom and endatom* have been selected for each processors. And if the number of atoms is not integer multiple of number of processors then the remaining atoms are assigned to last processor i.e. if there are 512 atoms and 8 processors then each processors will compute potential for 64 atoms. If there were 514 atoms then first 7 processors will process 65 atoms each and the last processor will process 59 atoms, which is not an optimized technique.

```
//! Divide atoms equally in order among
processors
nAtomsPerProc = ceil(float(nAtoms)/mpiSize);

 //! Assign start and end atom positions
belonging to a processor
 startAtom = mpiRank*nAtomsPerProc;
 endAtom = (mpiRank+1)*nAtomsPerProc;

 //! Adjust end atom position for last processor if
 //! number of atoms is not an integral multiple of
number of cores
 if(endAtom > nAtoms)
   endAtom = nAtoms;
```

Fig. 3. Atoms are staticly distributed (Block) among processors

After assigning the atoms per processors, the routine for computing potential is called. Fig. 4 shows how this algorithm works.

```
allocateMemory(rPairProductPotentials,
                        P.nAtomsPerProc,
nEffectiveWavefunctions);
   int numPotentials = 0;

 for(int
wfi=0;wfi<nEffectiveWavefunctions;wfi+=1){
     for(int wfj=0;wfj<wfi+1;wfj+=1){

computePotential(
            rPairProductsMatrix[wfi][wfj],
                  atomCoordinates,
                   rPairProductPotentials[wfi][wfj],
                        nAtoms,
                  P.nAtomsPerProc,
               P.startAtom, P.endAtom);
```

```
double potentialStart = MPI_Wtime();
   double dr, dx, dy, dz;
   double dielectricCorrection = dielectricConstant;

for (uLongInt i=startAtom; i<endAtom; i++) {
    rPairProductPotentials[i-startAtom] = 0;

   for (uLongInt j=0; j<nAtoms; j++) {
          dx = atomCoordinates[3*i+0] -
atomCoordinates[3*j+0];
          dy = atomCoordinates[3*i+1] -
atomCoordinates[3*j+1];
          dz = atomCoordinates[3*i+2] -
atomCoordinates[3*j+2];
```

```
dr = std::sqrt(dx*dx + dy*dy + dz*dz);

dielectricCorrection=getDielectric(dr);
          if(i==j)
            dr=centralCellR;
            rPairProductPotentials[i-startAtom]
              +=
rPairProduct[j]/(dielectricCorrection*dr);
    }
   }
   double potentialEnd = MPI_Wtime();
```

Fig. 4. ComputePotential routine

Fig. 4 shows that atom's 3D position pointer **atomCoordinates* is sent to the *computePotential* routine. Each processor performs two *for* loops. First *for* loop computes the potential from corresponding *startatom* to *endatom*. Second *for* loop takes the account for all the atoms including the computing atom itself. Finally the potential for the assigned atoms in a particular processor is saved in another array *rPairProductPotentials*. Number of elements in this array is equal to the number of atoms assigned to corresponding processor.

IV. Dynamic Distribution

In this work the atoms are dynamically assigned to the processors to compute the potential. Fig. 5 shows how it is done.

```
if (P.mpiRank == 0)
   {
   coordinates= atomCoordinates;
    atomcount =0;
for (rank = 1; rank < P.mpiSize; ++rank)
  {MPI_Send(coordinates, 3 ,MPI_DOUBLE,
rank,1,    MPI_COMM_WORLD);
   MPI_Send(& atomcount, 1, MPI_INT, rank,2,
MPI_COMM_WORLD);
       atomcount++;
     coordinates= coordinates+3;  }
```

```
while (flagEnd != 0 ) {
            for (rank = 1; rank < P.mpiSize; ++rank)
            {// check each processor for result
MPI_Iprobe (rank, 11, MPI_COMM_WORLD, & flagContinue, &status);
            if (flagContinue == 1)
                        {
MPI_Recv(& num, 1, MPI_C_DOUBLE_COMPLEX, MPI_ANY_SOURCE,
                                    11, MPI_COMM_WORLD, &status);
if (atomcount != nAtoms)   {
 Repeat send function
                                    }
if (atomcount == nAtoms-1) flagEnd = 0; }}
```

Fig. 5. Sending data from rank 0 processor

From rank 0 the 3D position of one atom each along with the atom index is sent to other processors. Fig. 6 shows the operation happened in other ranks. After calculating the potential for that atom, it sends the potential value back to root processor. So there is only one *for* loop in the compute ranks. In the meantime the root processor continuously checks for data sent from compute ranks via MPI_Iprobe. Whenever any data is available from any compute node, root processor receives that and sends the 3D position of next atom and atom index to latest available rank via

```
while(1)
{   MPI_Recv(&s, 3, MPI_DOUBLE, 0,
1,MPI_COMM_WORLD, &status);
    MPI_Recv(& atomcount, 1, MPI_INT, 0,
2,MPI_COMM_WORLD, &status);
for (uLongInt j=0; j<nAtoms; j++) {
            dx =  s[0] - atomCoordinates[3*j+0];
            dy =  s[1] - atomCoordinates[3*j+1];
```

```
dz =  s[2] - atomCoordinates[3*j+2];
dr = std::sqrt(dx*dx + dy*dy + dz*dz);
     dielectricCorrection=getDielectric(dr);
 if(dr ==0)   dr=centralCellR;
sum  += rPairProduct[j]/(dielectricCorrection*dr);//

MPI_Send(&sum, 1, MPI_C_DOUBLE_COMPLEX, 0, 11,
MPI_COMM_WORLD);
}
```

Fig. 6. Operation in other ranks

MPI_Send*(coordinates,3,MPI_DOUBLE, status.MPI_SOURCE, 1, MPI_COMM_WORLD);*

It also keeps account for atom numbers. That is how potential all the atoms is saved in only root processor.

V. Result and Comparisn

Following tables shows comperative illustration between static and dynamic distribution for 512 and 720 atoms respectively.

Table. 1. Comparisn between static and dynamic distribution for 512 atoms

Number of processors	Static Distribution	Dynamic distribution
P= 4/5	Speed: .0011 s Efficiency: 2.27	speed: .0037 Efficiency: .54
P=8 / 9	Speed: .0005 s Efficiency : 2.5	Speed: .0018 s Efficiency : .61
p= 16/ 17	Speed: .0005 s Efficiency: 1.25	Speed: .0009 s Efficiency: .65

Table. 2. Comparisn between static and dynamic distribution for 720 atoms

Number of processors	Static Distribution	Dynamic distribution
P= 4/5	Speed: .0028 s Efficiency: .89	speed: .0031 s Efficiency: .64
P=8 / 9	Speed: .0011 s Efficiency : 1.13	Speed: .0015 s Efficiency : .74
p= 16/ 17	Speed: .00105 s Efficiency: .59	Speed: .00106 s Efficiency: .58

If we look at the trend, although static distribution shows better performance, the gap between these two approaches reduces with increase of number of atoms. It should be mentioned that

normally the size of devices we work with contains more than hundred thousands atoms. In that case we can be hopeful that for realistic sized device, dynamic allocation will perform better when number of available processors is not an issue(applicable for National Lab clusters). There is another point to be noted. With the conventional block distribution, as each processors saves potential data only for assigned atoms, in later part of the code these data are accumulated by

MPI_Reduce*(&Jc,&Jpmnq,1,MPI_DOUBLE_COMPLEX,MPI_SUM,0,MPI_COMM_WORLD);*

Which can be avoided with dynamic distribution as data for all the atoms are saved in root processor.

VI. Conclusion

This work is still not completed. As dynamic approach is showing promise, it needs to be further tested with higher number of atoms and he whole code needs to be modified adopting this dynamic technique

VII. Acknowledgment

This work was financially supported by the U.S. National Science Foundation Grant No. 1102192. Part of the computational work was performed on the XSEDE and the nanoHUB HPC platforms. Access to Synopsys TCAD toolset via the SIU site is also acknowledged. Also collaborated with Purdue University.

References

[1] Sonia Buckley, Kelley Rivoire and Jelena Vučković, "Engineered Quantum Dot Single Photon Sources," *Rep. Prog. Phys,* vol. 75, 2012. S. Deshpande et al., Nature Comm., vol. 4, number. 1675, DOI: 10.1038/ncomms2691, 2013.

[2] G. Klimeck, et al., IEEE Trans. on Elect. Dev., vol. 54, 9, pp. 2079–99, 2007.