ORNL/TM-2006/578

Optimization of the CMDFT code

July 2006

Xiaoguang Zhang and Paul Kent

ORNL-1234

Optimization of the CMDFT code

Xiaoguang Zhang and Paul Kent

Date Published: July 2006

Prepared by
OAK RIDGE NATIONAL LABORATORY
P. O. Box 2008
Oak Ridge, Tennessee 37831-6285
managed by
UT-Battelle, LLC
for the
U. S. DEPARTMENT OF ENERGY
under contract DE-AC05-00OR22725

Contents

List of Figures

List of Tables

Acronyms

BOMD: Born-Oppenheimer Molecular Dynamics
DFT: Density Functional Theory
FFT: Fast Fourier Transformation
SCF: Self-Consistent Field

Executive Summary

This report outlines the optimization of the CMDFT code by Xiaoguang Zhang during June-July 2006. The overall improvement in speed is nearly 40%. Possible further optimizations are also discussed.

1 Introduction

The Born-Oppenheimer molecular dynamics (BOMD) method is a variant of the first-principles Car-Parrinello molecular dynamics. The BOMD method treats classically the nuclei motions and quantum mechanically the electrons using the density functional theory (DFT). The electronic structure is converged at every nuclei conformation. The Georgia Tech CMDFT code implements the BOMD method with a pseudopotential plane-wave basis. The detailed description of the method is given in [1].

The CMDFT code is written in f90, and uses MPI for communications. Here we summarize recent efforts in optimizing of the CMDFT code. All the tests are run on Jaguar.

2 Code efficiency before optimization

The test case used in this report is a nanocluster of 24 cobalt atoms. The reciprocal space grid is chosen as $\texttt{nx} = \texttt{ny} = 128$, and $\texttt{nz} = 160$. The number of wave functions used in the calculation is $\texttt{nwfu} = 240$. The calculation is stopped after 10 SCF iterations, before convergence is reached.

As shown in Table 1, the overall efficiency of the CMDFT code is very low, reaching 2.5×10^9 FLOPS which is only 16% of peak floating point execution speed. This compares poorly to an average code that predominantly uses LAPACK routines. The expected performance for such a code is typically around 50% of peak. Therefore there seems to be a significant room for improvement.

Table 1: Efficiencies of top CMDFT routines before optimization for a 24 atom cluster test case using 32 nodes on Jaguar.

Subroutine	Time (sec)	Time%	%peak FLOPS
`bdvs`	239.28	37.0	17.5
`gtorfft`	152.23	23.5	11.7
`rtogfft`	116.62	18.0	13.1
`hartree`	34.15	5.3	36.7
`ddendr`	28.94	4.5	32.9
`fftsetup`	27.13	4.2	35.1
`hwf`	23.10	3.6	4.3
...			
Total	676.38	100	16.3

The top three routines, `bdvs`, `gtorfft`, and `rtogfft`, are all involved in the

process of finding the eigenenergies and the corresponding wave functions. This process requires the computation of the Hamiltonian matrix $H_{ij} = \langle\psi_i|H|\psi_j\rangle$ and the overlap matrix $S_{ij} = \langle\psi_i|\psi_j\rangle$, where ψ_i are the trial wave functions. The trial wave functions ψ_i are stored in terms of its expansion in plane waves, indexed with the reciprocal wave vectors. However, the vectors $H|\psi_i\rangle$ are computed in real space. So that a Fourier transform of ψ_i is performed in `gtorfft` before the calculation. After the calculation a back Fourier transform is performed in `rtogfft`. At the end, the inner products $\langle\psi_i|H|\psi_j\rangle$ and $\langle\psi_i|\psi_j\rangle$ are computed in `bdvs`.

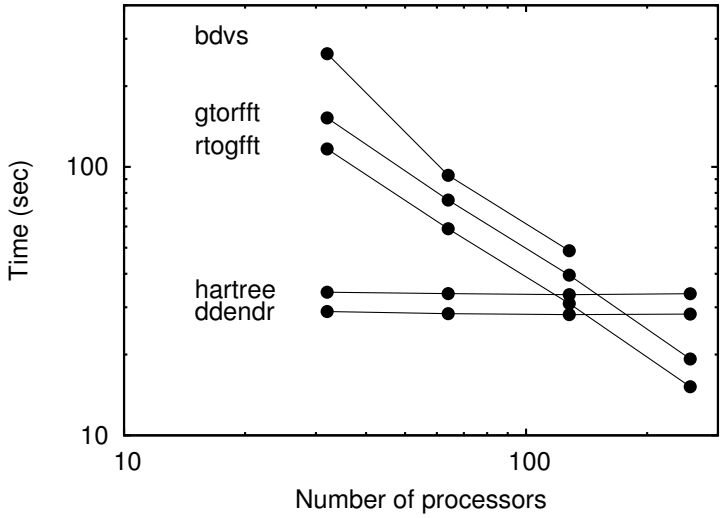

Figure 1: Scaling of CMDFT with number of processors before optimization.

The scaling of the top routines with the number of processors, on the other hand, shows very good linear scaling, as shown in Fig. 1. The flat performance of the two routines, `hartree` and `ddendr`, are due to the initial setup of the FFT in both routines. This will be discussed later along with the FFT optimization.

3 Optimization of `bdvs`

In bdvs, two matrices, H_{ij} and S_{ij} are computed by forming the inner products $\langle\psi_i|H|\psi_j\rangle$ and $\langle\psi_i|\psi_j\rangle$. Because both matrices are symmetric, only half of the matrix elements plus the diagonal ones are needed. This should save the computation time by about half. Unfortunately, to achieve this savings it prevents the usage of high level LAPACK routines. The original code uses `ddot` which results in a

low efficiency of 17%.

My first optimization attempt replaced all `ddot` calls with `dgemv` calls. Although this removes one explicit layer of `do`-loops, it made little difference in performance. One may instead forgo the factor of two savings and compute all elements, thus allowing the use of the highly efficient `dgemm` calls. The hope is that doubling in the number of computations will be more than compensated by the increase in efficiency.

In fact, we do not need to compute all elements to allow the use of the `dgemm` calls. The loop over the number of trial wave functions outside the `ddot` calls is divided into blocks of size `nwfg`. This was originally designed to improve the efficiency of message passing during the FFT calculations. Typical values of `nwfg` is between 10 and 100. Thus if we compute the matrix elements for each block of `nwfg` wave functions together, filling all `nwfg` rows up to the largest size needed within the block, then `dgemm` can be used with a minimal increase in the number of elements computed. In Fig. 2 we show the results of this change. One can see that the execution time of bdvs is dramatically reduced as a function of `nwfg`, reflecting the improved efficiency due to better cache management by using dgemm. On the other hand, the execution time of `gtorfft` changes little with nwfg, allowing more flexibility with the choice of `nwfg`.

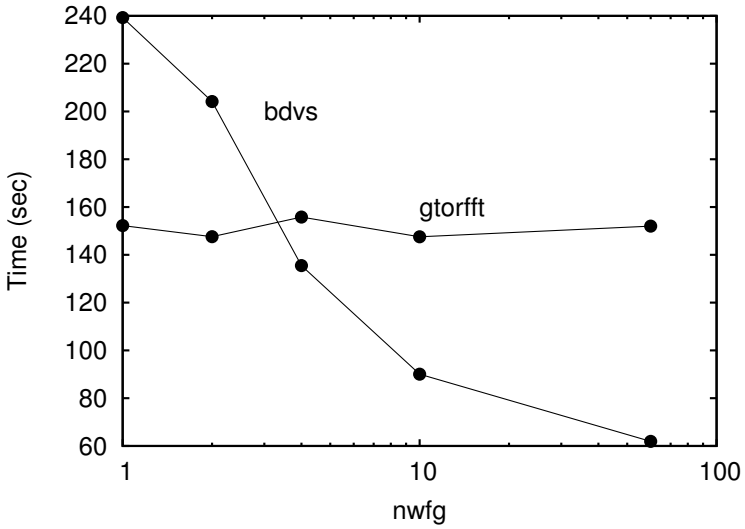

Figure 2: Execution times of `bdvs` and `gtorfft` as a function of the block size `nwfg`.

The changes are demonstrated in the following code example. Before the

change,

```
do jg=1,mwfg
 iwf=iwf1+jg-1
 b(1:ng_n,iwf)=wfg(1:ng_n,iwf)
 sis(iwf,iwf)=Ddot(2*ng_n,wfg(1,iwf),1,wfg(1,iwf),1)
enddo ! jg=1,mwfg
call hwf(wfg(1,iwf1),vloc,nrsp, hb,mwfg)
do jg=1,mwfg
 iwf=iwf1+jg-1
 nhop=nhop+1
 shs(iwf,iwf)=Ddot(2*ng_n,wfg(1,iwf),1,hb(1,jg),1)
 if(iwf.lt.nwf) then
  do jwf=iwf+1,nwf
   sis(jwf,iwf)=Ddot(2*ng_n,wfg(1,jwf),1,wfg(1,iwf),1)
   shs(jwf,iwf)=Ddot(2*ng_n,wfg(1,jwf),1,hb(1,jg),1)
  enddo
 endif
 if(nx1.gt.0) then
  do jx1=1,nx1
   jwf=nwf+jx1
   sis(jwf,iwf)=Ddot(2*ng_n,wfx1(1,jx1),1,wfg(1,iwf),1)
   shs(jwf,iwf)=Ddot(2*ng_n,wfx1(1,jx1),1,hb(1,jg),1)
  enddo
 endif
 if(nx2.gt.0) then
  do jx2=1,nx2
   jwf=nwf+nx1+jx2
   sis(jwf,iwf)=Ddot(2*ng_n,wfx2(1,jx2),1,wfg(1,iwf),1)
   shs(jwf,iwf)=Ddot(2*ng_n,wfx2(1,jx2),1,hb(1,jg),1)
  enddo
 endif
enddo ! jg=1,mwfg
```

Compared to the code after the change,

```
do jg=1,mwfg
 iwf=iwf1+jg-1
 b(1:ng_n,iwf)=wfg(1:ng_n,iwf)
enddo ! jg=1,mwfg
call hwf(wfg(1,iwf1),vloc,nrsp, hb,mwfg)
```

```
      nhop=nhop+mwfg
     call dgemm('t','n',mwfg,iwf1+mwfg-1,2*ng_n,1.d0,wfg(1,iwf1),
    &      2*mg_nx,wfg,2*mg_nx,0.d0,sis(iwf1,1),nmat)
     call dgemm('t','n',mwfg,iwf1+mwfg-1,2*ng_n,1.d0,hb,2*mg_nx,
    &      wfg,2*mg_nx,0.d0,shs(iwf1,1),nmat)
      if(nx1.gt.0) then
       call dgemm('t','n',nx1,2*ng_n,mwfg,1.d0,wfx1,2*ng1,
    &      wfg(1,iwf1),2*mg_nx,0.d0,sis(nwf+1,iwf1),nmat)
       call dgemm('t','n',nx1,2*ng_n,mwfg,1.d0,wfx1,2*ng1,
    &      hb,2*mg_nx,0.d0,shs(nwf+1,iwf1),nmat)
      endif
      if(nx2.gt.0) then
       call dgemm('t','n',nx2,2*ng_n,mwfg,1.d0,wfx2,2*ng2,
    &      wfg(1,iwf1),2*mg_nx,0.d0,sis(nwf+nx1+1,iwf1),nmat)
       call dgemm('t','n',nx2,2*ng_n,mwfg,1.d0,wfx2,2*ng2,
    &      hb,2*mg_nx,0.d0,shs(nwf+nx1+1,iwf1),nmat)
      endif
```

Similar changes are also made in other parts of `bdvs`. The final optimized version of `bdvs` yields a four-fold improvement in speed, reaching 71% peak performance, as shown in Table 2.

Table 2: Efficiencies of top CMDFT routines after optimization for a 24 atom cluster test case using 32 nodes on Jaguar.

Subroutine	Time (sec)	Time%	%peak FLOPS
`gtorfft`	125.49	30.1	14.2
`rtogfft`	103.47	24.9	14.7
`fftsetup`	75.69	18.2	36.4
`bdvs`	65.54	15.7	71.2
`hwf`	23.35	5.6	4.3
...			
`hartree`	0.92	0.2	12.9
`ddendr`	0.82	0.2	6.9
...			
Total	416.26	100	26.3

4 Optimization of the FFT routines

4.1 Setup

In the original code, FFT setup is done in three places, `fftsetup` for the wave function FFT, `hartree` for the Coulomb energy, and `ddendr` for charge density. As evident in the flat scaling of the execution time as a function of the number of nodes in Fig. 1, both `hartree` and `ddendr` are dominated by the setup of the FFT. Our first step in optimization of the FFT, is to merge the setup parts of both routines into `fftsetup`. This step cleans up the code structure significantly, while streamlines operation by removing duplicate parts. Table 2 shows the benefit of this change. Both `hartree` and `ddendr` now takes insignificant amount of time. The execution time of `fftsetup`, although longer than before, is still much less than the sum of execution times of all three routines before the change.

4.2 Computation of FFT

The main compute routines for FFT include the forward FFT routine `gtorfft`, and the backward FFT routine `rtogfft`. The two FTT routines have very poor floating point efficiencies in the low teens. Because `fftsetup` spends most of its time on trial runs of the FFT calculations, we can use its floating point efficiency, about 36%, as the target efficiency for the two FFT routines. This would represent a factor of three improvement in efficiency.

In both `gtorfft` and `rtogfft`, most of the time is spent on message passing between nodes and on transposing arrays to prepare for the FFT calls. Both can be improved by optimizing the distribution of the wave functions on the nodes.

The total number of plane waves is approximately `nx` $\times$ `ny` $\times$ `nz`. This is often a much larger number than the number of processors. Thus each processor is distributed several plane waves. In order to balance the load and memory requirement, the plane waves are not stored in sequence. Instead, wave vectors that share the same y and z indices are grouped into a column. Columns may have difference size (number of x indices) due to the truncation of the reciprocal vectors (x, y, z) at a spherical surface. Thus, these columns are first sorted according to their sizes, then distributed to the nodes according to a load-balanced algorithm.

This load-balanced distribution of the wave vectors significantly complicates the FFT calculation. The first FFT, along the x direction, is straightforward. Then, before the FFT along y, wave vectors need to be redistributed between the nodes, creating significant amount of message passing, and then the arrays need to be transposed to move the y index to the front. The same process is then repeated for z (see Fig. 3).

The key to optimization, is to design a distribution algorithm, that minimizes the message passing and transposing for the y and z directions, while keeping the load balanced. Before we discuss that, we need first to discuss some relevant data dependencies.

4.3 Data dependency in wave function storage

In the CMDFT code, assumptions on the order of the plane wave arrays are used in many subroutines. These assumptions are used to truncate the arrays, to throw away wave vectors not needed due to symmetry, and to map wave functions between x and $-x$ reciprocal vectors. Such strong data dependency must be removed before any change in load-balancing algorithm can be implemented.

Concurrent changes are made to both `wfgsetup` and `wfgsetup1` in order to remove this data dependency. Indexing arrays `i1ton`, `i2ton`, `i3ton`, `ntok1`, `ntok2`, `ntok3`, `n1_inv`, and `n2_inv`, are first generated using a particular order appropriately chosen (this choice will be discussed in more detail below). Then these arrays are used to produce the reciprocal vectors and to perform all other tasks. In this manner, only the generation of the indexing arrays depend on the particular algorithm, while the rest of code is completely independent.

4.4 New load-balancing algorithm

In the original algorithm implemented in `wfgsetup` and `wfgsetup1`, the wave vectors are loaded onto each node considering two factors, the size of the column being loaded, and the total size of the array already on the node. By loading the columns in the decreasing order in size, and onto the node with the least amount of data, an approximate load-balancing is achieved. The new algorithm takes into an additional consideration, the (y, z) coordinate of the column being loaded. The first attempt is to load the column onto a node that already has larger columns that shares an identical z coordinate. By maximizing the number of columns that share the same z coordinate on a single node, we minimize the amount of message passing before the FFT in the y direction.

A related change is made in `fftsetup` which distributes z slabs onto different nodes. Here the order of the loop over z is changed to maximize the possibility that two slabs with the same z can stay on the same node.

The improved result is shown in Table 2. We see that the speed of `gtorfft` is improved by about 18%, and `rtogfft` by about 12%.

4.5 Unresolved problem

Although the algorithm change discussed above is successful, some of the other alternative ordering of the columns produced incorrect answers. This is likely caused by hidden data dependency not yet discovered. Further work is needed to find and remove this dependency.

Once the data structure is completely isolated, we are free to test more load-balancing algorithms to find the optimal choice.

5 Summary of the speedup

After all the changes, the total speed of the CMDFT code is improved by nearly 40%. The floating point efficiency improved from 16% to 26% of the peak. At this point, the execution time is dominated by the FFT. Despite the improvement made on the load-balancing algorithm, both FFT routines still have poor 14% efficiency. Eliminating the hidden data dependency will allow the implementation of a more localized load-balancing scheme to further improve the FFT efficiency. However, staying with three separate one-dimensional FFT's will limit the potential for significant further speed improvement. A change in the algorithm on a higher level to reduce the number of FFT calculations in each SCF iteration can both boost floating point efficiency and improve scaling.

References

[1] Robert N. Barnett and Uzi Landman, "Born-Oppenheimer molecular-dynamics simulations of finite systems: Structure and dynamics of $(H_2O)_2$," Phys. Rev. B 48, 2081-2097 (1993).

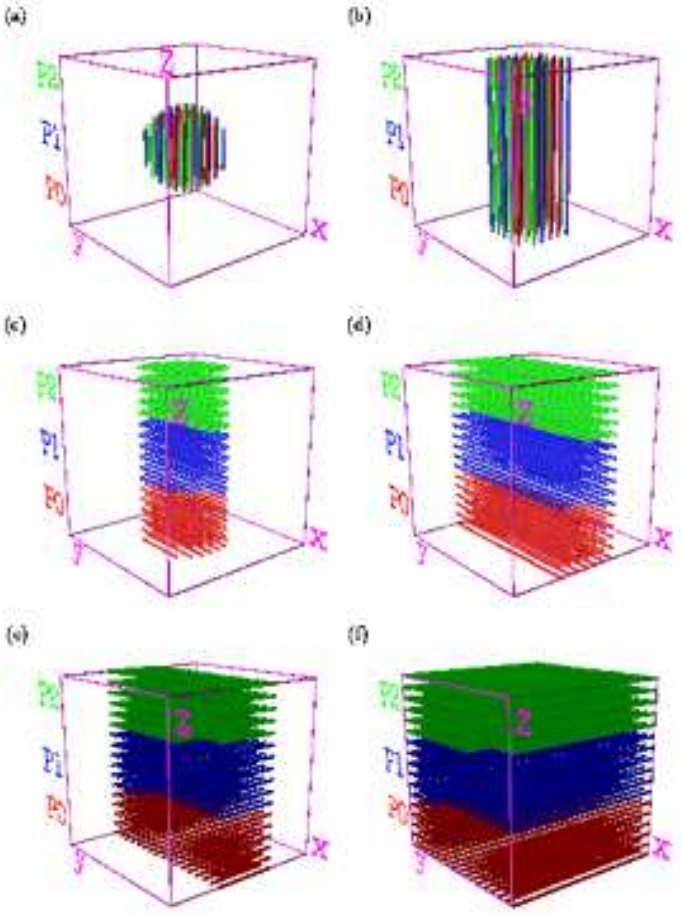

Figure 3: The three-dimensional FFT calculation is done as three sets of one-dimensional FFT's. (a) The sphere of original reciprocal vectors; (b) After FFT along x; (c) Global transpose from (b), this step require the most communications; (d) After FFT along y; (e) Global transpose from (d), this step requires much less communication than from (b) to (c); (f) After FFT along z. Source: Andrew Canning, LBNL