AICS TECHNICAL REPORT

No. 2014-002

Optimization of matrix-vector multiplication for Real-Space Density Functional Theory Code on the K computer

By

H. Suno*, Y. Nakamura, Y. Kuramashi, Y. Futamura, and T. Sakurai

RIKEN Advanced Institute for Computational Science

* Corresponding Author E-mail: suno@riken.jp

Submitted on 08/09/2014

Accepted on 03/10/2014

Optimization of matrix-vector multiplication for Real-Space Density Functional Theory Code on the K computer

H. Suno*

RIKEN Advanced Institute for Computational Science, Kobe 650-0047, Japan and
RIKEN Nishina Center, Wako 351-0198, Japan

Y. Nakamura

RIKEN Advanced Institute for Computational Science, Kobe 650-0047, Japan

Y. Kuramashi

RIKEN Advanced Institute for Computational Science, Kobe 650-0047, Japan
Center for Computational Science, University of Tsukuba, Ibaraki 305-8577, Japan and
Graduate School of Pure and Applied Science,
University of Tsukuba, Ibaraki 305-8571, Japan

Y. Futamura

Department of Computer Science, University of Tsukuba, Ibaraki 305-8573, Japan

T. Sakurai

Department of Computer Science, University of Tsukuba, Ibaraki 305-8573, Japan and
Center for Computational Science, University of Tsukuba, Ibaraki 305-8577, Japan

(Dated: September 8, 2014)

Abstract

We have carried out an optimization of the matrix-vector multiplication in the Real-Space Density Functional Theory (RSDFT) program on the K computer. We have performed a reduction of the cache misses, an efficient SIMDization and a reduction of the memory access latency by modifying the source code and inserting optimization directives. The performance efficiency for the finite-difference operation is improved up to a peak ratio of more than 19% thanks to the code tuning. For the whole part of the matrix-vector multiplication in the RSDFT code, the performance efficiency is increased from 2.98% to 8.33%, i.e., the procedure can be now performed 2.8 times faster than the previous version.

*Electronic address: suno@riken.jp

I. INTRODUCTION

The Real-Space Density Functional Theory (RSDFT) code is a program for electronic-structure calculations based on first-principles of quantum mechanics developed by Center for Computational Sciences (CCS) at University of Tsukuba in order to simulate the physical properties of materials in microscopic scale [1]. The code is based on the Density Functional Theory (DFT) proposed by Kohn and Sham 50 years ago [2, 3] and it employs the pseudopotential method [4]. The Real-Space (RS) scheme [5, 6] allows us to efficiently use massively parallel computations, since it does not need any Fast Fourier Transformation (FFT). Although the applications are limited to medium-sized systems consisting of hundreds of atoms until recently, current research interests in the material sciences requires the RSDFT calculations for much larger systems such as nanoscale systems with $O(10^4)$ atoms or more. In order to carry out such large calculations, it is imperative to optimize the code on massively parallel supercomputers such as the K computer at the RIKEN Advanced Institute for Computational Science.

The primary bottleneck in the RSDFT calculations is the Gram-Schmidt orthogonalization and the subspace diagonalization (diagonalizing the Hamiltonian matrix within a limited eigen-subspace), which consume 60% and 20% of the total wall clock time for the typical benchmark data, respectively. These computations can be performed with high efficiency by rearranging them to matrix-matrix multiplications, extending the parallelized axis coordinates (additional parallelizing with respect to the number of orbitals), using an optimized mapping (automatically choosing one-, two-, or three-dimensional torus mapping adapted for a given problem), and the Tofu-specific collective communication algorithms [7–9]. The secondary bottleneck is the Conjugate Gradient (CG) calculation, where almost all the computational cost is spent in the matrix-vector multiplication operating the Hamiltonian matrix on orbitals. Since the CG calculations take small portion of computation time in the RSDFT code, its optimization has been left behind so far. However, we expect that the cost of these calculations potentially increase when the RSDFT program is applied to a wider range of problems, such as band structure calculations using the Sakurai-Sugiura eigensolver[10]. In this report, we make an optimization for the matrix-vector multiplication in the CG calculation on the K computer aiming at a better performance of the RSDFT calculations.

II. MATRIX-VECTOR MULTIPLICATION IN THE RSDFT PROGRAM

The goal of the RSDFT calculations is to minimize the energy functional $E[\rho]$ with respect to the electronic density ρ. The real space scheme defines the theory on a three-dimensional spatial grid with M_L the number of grid points. The orbitals, electronic density and potentials are expressed as column vectors, whose elements are the values at grid points. For example, the orbitals are represented by

$$\vec{\phi} = \begin{pmatrix} \phi^1 \\ \vdots \\ \phi^i \\ \vdots \\ \phi^{M_L} \end{pmatrix}, \tag{1}$$

where the i-th element is the value at the grid point $\mathbf{r}_i$:

$$\phi^i = \phi(\mathbf{r}_i), \tag{2}$$

and the electronic density is expressed in the same way as

$$\rho^i = \rho(\mathbf{r}_i) = \sum_{n=1}^{M_B} f_n |\phi_n^i|^2, \tag{3}$$

where M_B is the total number of orbitals and f_n the occupation number for the n-th orbital. In this scheme, the energy functional can be written as

$$\begin{aligned} E[\phi^1, \phi^2, ..., \phi^{M_L}] \;=\; & -\frac{1}{2}\sum_{n=1}^{M_B} f_n \sum_{i=1}^{M_L}\sum_{j=1}^{M_L} \phi_n^{i*} L_{ij} \phi_n^j \Delta\Omega - \frac{4\pi}{2}\sum_{i=1}^{M_L}\sum_{j=1}^{M_L} \rho^i L_{ij}^{-1} \rho^j \Delta\Omega \\ & + E_{xc}[\rho^1, \rho^2, ..., \rho^{M_L}] + \sum_{i=1}^{M_L} v_L^i \rho^i \Delta\Omega + \sum_{n=1}^{M_B} f_n \sum_{i=1}^{M_L}\sum_{j=1}^{M_L} \phi_n^{i*} V_{ij}^{NL} \phi_n^j \Delta\Omega, \end{aligned} \tag{4}$$

where $\Delta\Omega$ is the volume element, L_{ij}, v_L^i and V_{ij}^{NL} represent the finite-difference operator, local potential and nonlocal pseudopotential, respectively. The exchange-correlation energy E_{xc} characterizes the many-electron effects. Minimization of the energy functional combined with the orthonormalization constraints for the orbitals leads to the eigenvalue equation for the orbitals

$$H\vec{\phi}_n = \varepsilon_n \vec{\phi}_n \tag{5}$$

with

$$H = -\frac{1}{2}L + V + V^{NL}, \tag{6}$$

where the Hamiltonian consists of a sparse matrix of the finite-difference operator L, a diagonal matrix of the local potential V (including the Hartree and exchange-correlation potentials) and a matrix of the nonlocal operator V_{NL} .

The subject of this report is the optimization of the matrix-vector multiplication operating the Hamiltonian matrix on the orbitals. Note that we treat multiple orbitals. The finite-difference operator acts on orbitals as follows:

$$\begin{aligned}\left(\frac{\partial^2}{\partial x^2}+\frac{\partial^2}{\partial y^2}+\frac{\partial^2}{\partial z^2}\right)\phi_n(x_i,y_i,z_i) \approx & \sum_{m=-M_D}^{M_D} C_m\phi_n(x_i+m\Delta_x,y_i,z_i) \\ & + \sum_{m=-M_D}^{M_D} C_m\phi_n(x_i,y_i+m\Delta_y,z_i) \\ & + \sum_{m=-M_D}^{M_D} C_m\phi_n(x_i,y_i,z_i+m\Delta_z),\end{aligned} \tag{7}$$

where $(\Delta_x, \Delta_y, \Delta_z)$ are the grid spacings in the x-, y- and z-directions and C_m's the coefficients of the finite difference. Since the local potential is a diagonal matrix, its operation on an orbital is straightforward:

$$(V\vec{\phi}_n)^i = v(\mathbf{r}_i)\phi_n(\mathbf{r}_i). \tag{8}$$

The operation of the nonlocal potential is performed in two steps. First, we compute an inner product between a projector function p_{alm} and an orbital ϕ,

$$\beta_{alm} = C_{alm}\int_{\Omega_a} d\mathbf{r}p_{alm}(\mathbf{r})\phi_n(\mathbf{r}) \approx C_{alm}\sum_{j\in\Omega_a} p^{k*}_{alm}\phi^j_n\Delta V, \tag{9}$$

and then we compute the linear combination

$$(V^{NL}\vec{\phi}_n)^i = \sum_{a=1}^{N}\sum_{l=0}^{L_a}\sum_{m=-l}^{l} p^i_{alm}\beta_{alm}, \tag{10}$$

where the summation index a covers the number of ions N, and L_a is usually taken to be from 0 to 2. The integral of the inner product Eq. (9) is always performed within a small region around each ion irrespective of the whole system size. The matrix-vector multiplication in the RSDFT code is performed in parallel on the three-dimensional subgrids assigned in MPI processes. The calculations are further parallelized with OpenMP threads in each MPI process.

III. CODE OPTIMIZATION

A. Computer environment and performance test for the original code

The code optimization is carried out on the K computer at the RIKEN Advanced Institute for Computational Science. The machine consists of 82944 computational nodes and 5184 I/O nodes connected by the so-called "Tofu" network, providing 11.28 Pflops of computing capability. The Tofu network has six-dimensional topology with 3D-mesh times 3D-torus shape. Each node has a single 2.0GHz SPARC64 VIIIfx processor equipping 8 cores with SIMD enabled 256 registers, 6MB shared L2 cache and 16GB of memory. The L1 cash sizes per each core are 32KB/2WAY (instruction) and 32KB/2WAY (data). We use four compute nodes on the K computer.

We employ a silicon nanowire consisting of 9084 Si atoms and 840 H atoms as a benchmark data sample. The total grid size is $348 \times 348 \times 72$ in the $x \times y \times z$ directions, which is divided into $2 \times 2 \times 1$ MPI processes. Each process has therefore $174 \times 174 \times 72$ grid points. The Hamiltonian matrix is supposed to operate on 32 orbitals at once. The number of OpenMP threads is 8 corresponding to the number of cores in each node on the K computer. In order to pin down the most computationally expensive parts in the matrix-vector multiplication, we carry out performance test employing the advanced profiler with the insertion of the `start_collection` and `stop_collection` subroutines into the code. Table I shows the results of the performance test for the original RSDFT code. We find that most of the computational cost is paid in the operations of finite-difference (nondiagonal), nonlocal potential (first half), and nonlocal potential (second half). The optimization should be focused on these three parts.

B. Optimizing nondiagonal finite-difference operation

The finite-difference operation is given in Eq. (7). The source code for the operation of the nondiagonal elements is written as follows:

```
real(8) :: www(a1b-Md:b1b+Md, a2b-Md:b2b+Md, a3b-Md:b3b+Md)
real(8) :: htpsi(n1:n2, ib1:ib2) ! n1=1,n2=217982
```

TABLE I: Results for performance test of the matrix-vector multiplication in the original RSDFT code

	time (sec)	peak ratio (%)	MFLOPS/CPU
Copy of orbitals to work array	0.38	0	0
Communication of border information	0.69	0	0
Finite difference (diagonal)	0.36	1.52	1949
Finite difference (nondiagonal)	3.79	7.76	9936
Local potential	0.50	2.20	2813
Nonlocal potential (first half)	1.66	2.89	3695
Communication of projection-orbital-product	0.25	0	3
Nonlocal potential (second half)	5.97	0.80	1018
Total	13.60	2.98	3821

```
do ib=ib1,ib2  ! ib1=1,ib2=32
  n=ib-ib1+1
  do m=1,Md ! Md=6
!$OMP parallel private(i)
    i=n1-1+omp_idisp
    do i3=omp_a3b,omp_b3b ! omp_a3b=0,omp_b3b=7
      do i2=a2b,b2b       ! a2b=0,b2b=173
        do i1=a1b,b1b     ! a1b=0,b1b=173
          i=i+1
          htpsi(i,ib)=htpsi(i,ib) &
            +coef_lap(1,m)*( www(i1+m,i2,i3,n)+www(i1-m,i2,i3,n) ) &
            +coef_lap(2,m)*( www(i1,i2+m,i3,n)+www(i1,i2-m,i3,n) ) &
            +coef_lap(3,m)*( www(i1,i2,i3+m,n)+www(i1,i2,i3-m,n) )
    end do; end do; end do
!$OMP end parallel
end do; end do
```

where the start and end loop-indices given in the comments are those in the master node and

the master thread. Although these numbers may be different for other nodes and threads, the loop lengths are chosen to be almost the same over all the nodes. Figure 1 shows the distribution of the computational cost in the nondiagonal finite-difference operation. We find that the floating point (FP) cache access waiting is dominant. Since the array elements are arranged in the cache memory space in the way that the leftmost index changes most rapidly in Fortran, the neighboring elements in the z-direction like `www(i1,i2,i3±m,n)` locate far away in the memory space. This is a potential source of a number of cache misses when loading those elements. Let us examine it in more detail. The array `www` is defined by a four-dimensional array with a size of $186 \times 186 \times 84 \times 32$ expressed as `www(-6:179,-6:179,-6:77,1:32)`. The array lengths in the first, second and third dimensions accommodate the original grid size of $174 \times 174 \times 72$ assigned to each MPI process and additional six border grid points necessary for the finite-difference operation. The array length in the last (fourth) dimension corresponds to the number of orbitals employed for the benchmark data set. Therefore, in double precision, the first dimension (left side) extends over 1.5kB, the second one over 270kB, the third one over 22MB, and the fourth one over 270MB in the memory space. Taking account of the fact that the K-computer has 6MB of L2 cache per node, we can discuss whether the data are on cache or not. The `Md=6`-th neighboring elements in the $\pm x$ and $\pm y$ directions are on cache since they locate 8B×13=104B and 270kB×13=3.43MB away in the memory space, respectively. On the other hand, the neighboring elements in the $\pm z$ directions are not on cache since they locate 22MB×13=286MB away. They have to be fetched through access to the memory. Based on these consideration we count the number of memory access and the number of operations. We find that 9 floating point operations take place, while (2+3)×8=40B of data are fetched from the memory through one load-store and three load. The B/F (Byte/Flop) ratio should be about 4.

In such three-dimensional finite-difference operations, multiple cores can share the neighboring data in the z-direction with the aid of block-cyclic distribution of memory-shared parallelization[11], so that we can reduce the number of cache misses. The following example shows the B/F ratio is reduced to be 2.7.

```
!$OMP parallel private(i)
do ib=ib1,ib2
```

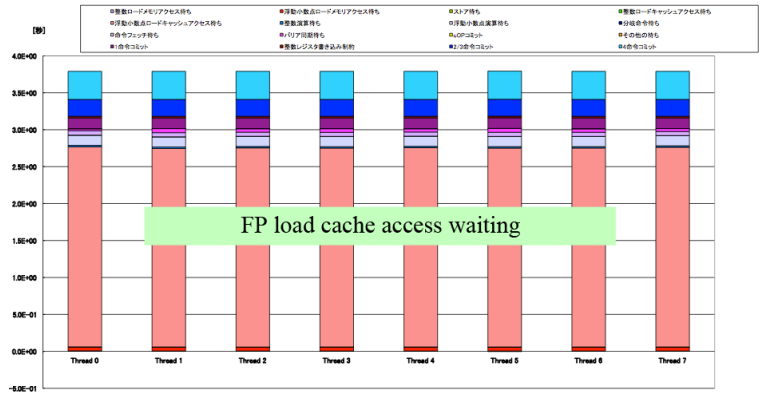

FIG. 1: Distribution of computational cost in the nondiagonal finite-difference operation.

```
  n=ib-ib1+1
  do m=1,Md
!$OMP do schedule(static,1)
    do i3=a3b,b3b
      do i2=a2b,b2b
        do i1=a1b,b1b
          i=((i3-a3b)*(b2b-a2b+1)+(i2-a2b))*(b1b-a1b+1)+i1-a1b+n1
          htpsi(i,ib)=htpsi(i,ib) &
            +coef_lap(1,m)*( www(i1+m,i2,i3,n)+www(i1-m,i2,i3,n) ) &
            +coef_lap(2,m)*( www(i1,i2+m,i3,n)+www(i1,i2-m,i3,n) ) &
            +coef_lap(3,m)*( www(i1,i2,i3+m,n)+www(i1,i2,i3-m,n) )
    end do; end do; end do
!$OMP end do
end do; end do
!$OMP end parallel
```

With this modification, the wallclock time is reduced from 3.79 to 2.59 sec and the peak ratio of floating point operation is increased from 7.76% to 11.40%. It should be noted that we have failed to obtain any improvement by a cyclic distribution with the autoparallelization directive (`!ocl parallel_cyclic(1)`). This is probably due to the details of program structure. We further modify the source code changing the loop position as shown below:

```
!$OMP parallel private(i)
```

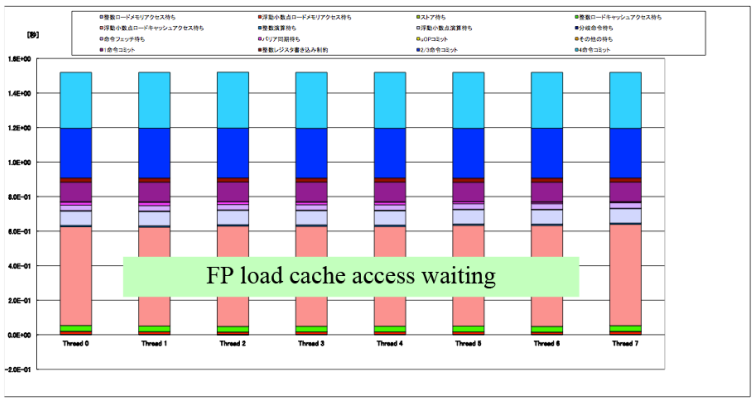

FIG. 2: Distribution of computational cost in the nondiagonal finite-difference operation after tuning.

```
do ib=ib1,ib2
  n=ib-ib1+1
!$OMP do schedule(static,1)
  do i3=a3b,b3b
    do m=1,Md! the loop has come inside here.
      do i2=a2b,b2b
        do i1=a1b,b1b
          i=((i3-a3b)*(b2b-a2b+1)+(i2-a2b))*(b1b-a1b+1)+i1-a1b+n1
          htpsi(i,ib)=htpsi(i,ib) &
            +coef_lap(1,m)*( www(i1+m,i2,i3,n)+www(i1-m,i2,i3,n) ) &
            +coef_lap(2,m)*( www(i1,i2+m,i3,n)+www(i1,i2-m,i3,n) ) &
            +coef_lap(3,m)*( www(i1,i2,i3+m,n)+www(i1,i2,i3-m,n) )
  end do; end do; end do; end do
!$OMP end do
end do
!$OMP end parallel
```

The wallclock time is then reduced to 1.52 sec and the peak ratio of the floating point operation is improved to 19.41%. We successfully obtain better performance. The computational cost is illustrated in Fig. 2, where the floating point load cache access waiting is reduced significantly compared to Fig. 1.

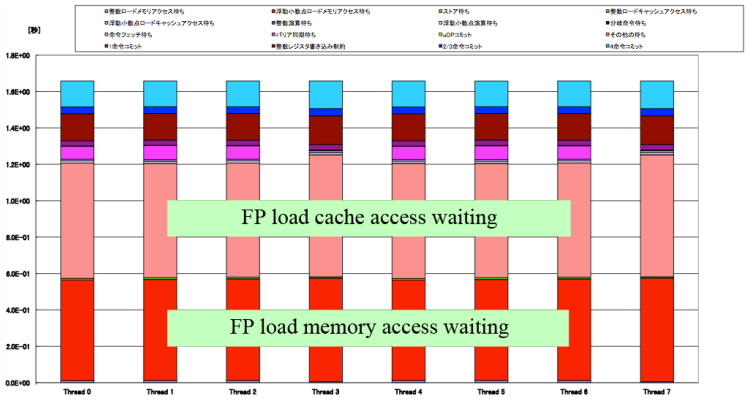

FIG. 3: Distribution of computational cost in the nonlocal potential operation (first half).

C. Optimizing nonlocal potential operation (first half)

The first half of the nonlocal potential opeartion is expressed in Eq. (9). The source code is given as below:

```
!$OMP parallel
!$OMP workshare
uVunk(:,:)=zero
!$OMP end workshare
do ib=ib1,ib2 ! ib1=1,ib2=32
!$OMP do
  do lma=1,nzlma     ! nzlma=10224
    do j=1,MJJ(lma) ! MJJ=0-1056,sum(MJJ)=9501072
      uVunk(lma,ib)=uVunk(lma,ib)+uVk(j,lma,k)*tpsi(JJP(j,lma),ib)
    end do
    uVunk(lma,ib)=iuV(lma)*dV*uVunk(lma,ib)
  end do
!$OMP end do
end do
!$OMP end parallel
```

Since the above computation has a very high B/F ratio of 16 and also involves indirect array access (list accesses or indirect references), it could be the most difficult part to be

speeded up on the K computer [8]. Figure 3 shows the distribution of the computational cost measured by the advanced profiler. We find some load imbalance, potentially because the innermost loop, which is OpenMP threaded, changes its length `MJJ` depending on the variable `lma`. The prefetch is automatically disabled due to the indirect accesses. Accessing the array `uVunk` after having set all the elements to zero is also superfluous referencing, since it results in fetching each array element twice from the memory. Taking account of these problems, we modify the source code as follows:

```
do ib=ib1,ib2
  do lma=1,nzlma
    uVunk1(lma,ib)=zero
!ocl prefetch
    do j=1,MJJ(lma)
      uVunk1(lma,ib)=uVunk1(lma,ib)+uVk(j,lma,k)*tpsi(JJP(j,lma),ib)
    end do
    uVunk1(lma,ib)=iuV(lma)*dV*uVunk1(lma,ib)
end do; end do
```

Here, instead of setting all the array elements to zero at once, we set each element to zero and then use it for the computation while it is still on cache. We also implement forced prefetching by inserting the directive (`!ocl prefetch`). We remove the OpenMP directive to enhance autoparallelizing since the code may be better optimized with autoparallelizing than with OpenMP threading [11]. With these modifications, the wallclock time is decreased from 1.66 sec to 1.09 sec and the peak ratio of the floating point operation is improved from 2.89% to 4.39%. Furthermore, in order to make the maximum use of the `uVk` array on cache, we modify the code as follows by unrolling 4-fold the outermost loop:

```
unroll=4
itemp=ib2-mod(ib2-ib1+1,unroll)
do ib=ib1,itemp,unroll
  do lma=1,nzlma
    uVunk1(lma,ib  )=zero
    uVunk1(lma,ib+1)=zero
```

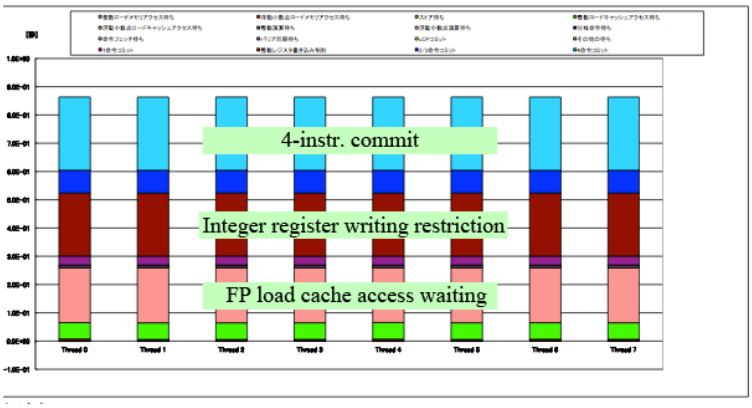

FIG. 4: Computational cost distribution in the nonlocal potential operation (first half) after tuning

```
!ocl prefetch
    do j=1,MJJ(lma)
      uVunk1(lma,ib  )=uVunk1(lma,ib  )+uVk(j,lma,k)*tpsi(JJP(j,lma),ib  )
      uVunk1(lma,ib+1)=uVunk1(lma,ib+1)+uVk(j,lma,k)*tpsi(JJP(j,lma),ib+1)

    end do
    uVunk1(lma,ib  )=iuV(lma)*dV*uVunk1(lma,ib  )
    uVunk1(lma,ib+1)=iuV(lma)*dV*uVunk1(lma,ib+1)

  end do
end do
(remaining operations when the loop length is not divisible by 4)
```

The wallclock time is reduced to be 0.86 sec and the peak ratio of the floating point operation becomes 5.53%, so that the code runs almost twice faster than the original version. The cost distribution in Fig 4 shows significant improvement. The cost called "Integer register writing restriction" means a failure in 4-instruction commit, albeit a success in 3-, 2-, or 1-instruction commit.

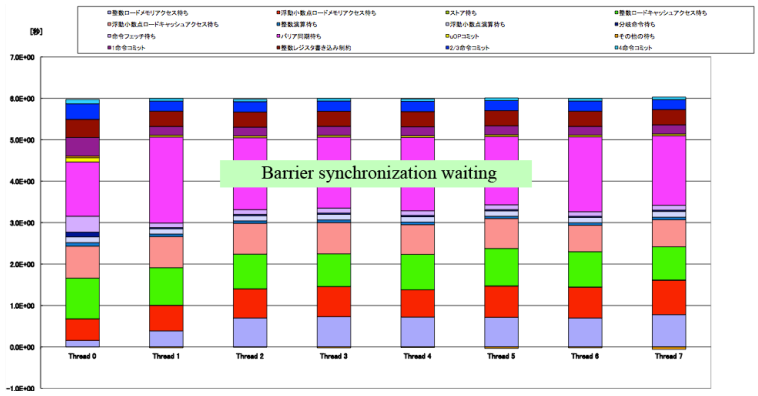

FIG. 5: Distribution of computational cost in the nonlocal potential operation (second half)

D. Optimizing nonlocal potential operation (second half)

The second half of the nonlocal potential operation is expressed in Eq. (10). The original source code is as follows:

```
do ib=ib1,ib2       ! ib1=1,ib2=32
!$OMP parallel do private(i)
  do lma=1,nzlma    ! nzlma=10224
    do j=1,MJJ(lma) ! MJJ=0~1056,sum(MJJ)=9501072
      i=JJP(j,lma)
      htpsi(i,ib)=htpsi(i,ib)+uVk(j,lma,k)*uVunk(lma,ib)
    end do
!$OMP end parallel do
  end do
end do
```

As in the first half of the nonlocal potential operation, prefetching is not performed due to the indirect accesses. The operation is not SIMDized at all because of the indirect writing to an array (zero SIMD rate). The cost distribution measured by the advanced profiler is shown in Fig. 5. The length of the innermost loop changes depending on the variable `lma`, so that load imbalance could happen when the `lma` loop is parallelized. In addition, we find a significant contribution of the barrier synchronization waiting.

Based on the analysis of the advanced profiler, we modify the source code as follows:

```
do ib=ib1,ib2
  do lma=1,nzlma
!ocl prefetch
!ocl norecurrence(htpsi)
    do j=1,MJJ(lma)
      htpsi(JJP(j,lma),ib)=htpsi(JJP(j,lma),ib)&
        +uVk(j,lma,k)*uVunk1(lma,ib)
end do; end do; end do
```

Here, we insert the prefetching directive (`!ocl prefetch`) and the directive (`!ocl norecurrence(htpsi)`) indicating that there is no multiple referencing to the array elements. The OpenMP directive for the innermost loop is removed in order to autoparallelize the outermost loop. With these modifications, though the SIMD rate is still low at 6.2%, the wallclock time is reduced from 5.97 sec to 1.38 sec and the peak ratio of the floating point operation increased from 0.80% to 3.44%. The source code is further modified unrolling the outermost loop for an efficient use of the `uVk` array in the cache as follows:

```
unroll=4
itemp=ib2-mod(ib2-ib1+1,unroll)
do ib=ib1,itemp,unroll
  do lma=1,nzlma
!ocl prefetch
!ocl norecurrence(htpsi)
    do j=1,MJJ(lma)
      htpsi(JJP(j,lma),ib  )=htpsi(JJP(j,lma),ib  )+uVk(j,lma,k)*uVunk1(lma,ib  )
      htpsi(JJP(j,lma),ib+1)=htpsi(JJP(j,lma),ib+1)+uVk(j,lma,k)*uVunk1(lma,ib+1)
      htpsi(JJP(j,lma),ib+2)=htpsi(JJP(j,lma),ib+2)+uVk(j,lma,k)*uVunk1(lma,ib+2)
      htpsi(JJP(j,lma),ib+3)=htpsi(JJP(j,lma),ib+3)+uVk(j,lma,k)*uVunk1(lma,ib+3)
    end do
  end do
end do
(remaining operations when the loop length is not divisible by 4)
```

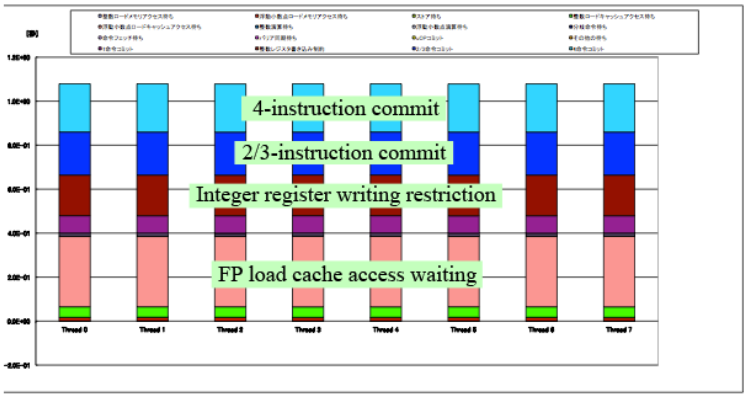

FIG. 6: Distribution of computational cost in the nonlocal potential operation (second half) after tuning

As a result, the wallclock time is reduced to 1.08 sec and the peak ratio of the floating point operation is increased to 4.41%. The code runs nearly six times faster than the original version. The cost distribution measured by the advanced profiler is improved as shown in Fig. 6.

E. Miscellaneous optimization

Since memory accesses are very expensive on scaler supercomputers such as the K computer, it is not a good way to repeat the reference to the same huge array. A bad example is as follows:

```
!$OMP parallel private(n,i)
do ib=ib1,ib2
  n=ib-ib1+1; i=n1-1+omp_idisp
  do i3=omp_a3b,omp_b3b
    do i2=a2b,b2b
      do i1=a1b,b1b
        i=i+1
        www(i1,i2,i3,n)=tpsi(i,ib)

do ib=ib1,ib2
!$OMP parallel do
```

```
  do i=n1,n2
    htpsi(i,ib)=c*tpsi(i,ib)

do ib=ib1,ib2
!$OMP parallel do
  do i=n1,n2
    htpsi(i,ib)=htpsi(i,ib)+Vloc(i,s)*tpsi(i,ib)
```

The entire `tpsi` array is referenced three times and the entire `htpsi` array is referenced twice. These references can be saved by rewriting the code as follows,

```
do ib=ib1,ib2
  do i3=a3b,b3b
    do i2=a2b,b2b
      do i1=a1b,b1b
        i=((i3-a3b)*(b2b-a2b+1)+(i2-a2b))*(b1b-a1b+1)+i1-a1b+n1
        www(i1,i2,i3,ib-ib1+1)=tpsi(i,ib)
        htpsi(i,ib)=(c+Vloc(i,s))*tpsi(i,ib)
end do; end do; end do; end do
```

This modification allows us to reduce the wallclock time from $0.38 + 0.36 + 0.50 = 1.24$ sec to 0.65 sec.

IV. SUMMARY

After the optimizations described above, the performance of the matrix-vector multiplication has been significantly improved as shown in Table II. The wallclock time is decreased from 13.60 sec to 4.81 sec and the peak ratio of the floating point operation is increased from 2.98 % to 8.33 %. The performance is about three times improved compared to the original version. Note that our optimization is performed for a typical choice of the subgrid size, the order of the finite difference and the number of orbitals. We also add a comment that autoparallelizing makes the code run faster than OpenMP threading as reported in the K-computer speedup workshop [11].

TABLE II: Results for performance test of the matrix-vector multiplication in the RSDFT code after tuning

	time(sec)	peak ratio(%)	MFLOPS/CPU
Copy of orbitals to work array			
+Finite-difference (diagonal)			
+Local potential	0.65	1.67	2144
Communication of border information	0.61	0	0
Finite-difference (nondiagonal)	1.52	19.40	24832
Nonlocal potential (first half)	0.86	5.53	7078
Communication of projection-orbital-product	0.08	0.01	9
Nonlocal potential (seconf half)	1.08	4.39	5618
Total	4.81	8.33	10666
Total (before tuning)	13.60	2.98	3821

Acknowledgments

All the results are obtained by using the K computer at the RIKEN Advanced Institute for Computational Science.

[1] J. Iwata, D. Takahashi, A. Oshiyama, T. Boku, K. Shiraishi, S. Okada, and K. Yabana, J. Comp. Phys. **229**, 2339 (2010).

[2] P. Hohenberg and W. Kohn, Phys. Rev. **136**, B864 (1964).

[3] W. Kohn and L. J. Sham, Phys. Rev. **140**, A1133 (1965).

[4] N. Troullier and J. L. Martins, Phys. Rev. B **43**, 1993 (1991).

[5] J. R. Chelikowsky, N. Troullier, K. Wu, and Y. Saad, Phys. Rev. B **50**, 11355 (1994).

[6] K. Yabana and G. F. Bertsch, Phys. Rev. B **54**, 4484 (1996).

[7] Y. Hasegawa, in *K computer Speedup Workshop* (2013).

[8] I. Minami, in *K computer Speedup Workshop* (2013).

[9] I. Minami, in *Symposium on Advanced Computing Systems and Infrastructures* (2012).

[10] Y. Futamura, T. Sakurai, S. Furuya, and J.-I. Iwata, Lecture Notes in Computer Science

7851, 226 (2013).

[11] Y. Aoyama, in *Tora-no-maki for programming on the K computer* (2013).