Parallelization of Metalwalls

Abel Marin-Lafleche

Contents

1 Long Range potential kernel

The long range potential equation is derived from the energy equation as $V_i^{\text{lr}} = \frac{\partial U_c^{\text{lr},*}}{\partial q_i}$. Using the formula given in the Ewald summation description, this yields:

$$V_i^{\text{lr}} = \frac{2}{ab}\sum_j Q_j \sum_{\substack{(l,m)\in\mathbf{Z^2}\\(l,m)\neq(0,0)}} \int_{-\infty}^{\infty} du\, \frac{\cos\left(l\frac{2\pi}{a}x_{ij} + m\frac{2\pi}{b}y_{ij} + uz_{ij}\right)}{\left(l\frac{2\pi}{a}\right)^2 + \left(m\frac{2\pi}{b}\right)^2 + u^2} e^{-\frac{\left(l\frac{2\pi}{a}\right)^2+\left(m\frac{2\pi}{b}\right)^2+u^2}{4\alpha^2}} \tag{1}$$

The integral over du is discretized using the midpoint rule with a step size of $du = \frac{2\pi}{c}$. The term $\int_{-\infty}^{\infty} du\, f(u)$ is therefore replaced by $\frac{2\pi}{c}\sum_{n\in\mathbf{Z}} f\left(n\frac{2\pi}{c}\right)$.

We call a *k-point* the triplet

$$k_{lmn} = (l\frac{2\pi}{a}, m\frac{2\pi}{a}, n\frac{2\pi}{c}) \text{ where } (l,m,n) \in \mathbf{Z^3}. \tag{2}$$

For large values of the k-point norm

$$|k_{lmn}| = \sqrt{\left(l\frac{2\pi}{a}\right)^2 + \left(m\frac{2\pi}{b}\right)^2 + \left(n\frac{2\pi}{c}\right)^2},$$

the term $\exp(\frac{-|k_{lmn}|^2}{4\alpha^2})$ is negligible. A cut-off value $k_{\max}$ is used and only terms for which $|k_{lmn}| < k_{\max}$ are taken into account. Thus the infinite sums are truncated such that

$$|l| < \left\lceil k_{\max}\frac{a}{2\pi}\right\rceil, |m| < \left\lceil k_{\max}\frac{b}{2\pi}\right\rceil \text{ and } |n| < \left\lceil k_{\max}\frac{c}{2\pi}\right\rceil.$$

The term inside the sum is symmetric with respect to the (l, m, n) triplets. Half of the computation effort is avoided by computing only terms for which

$$(l = 0, m \in [1, m_{\max}], n \in [-n_{\max}, n_{\max}]) \text{ and}$$
$$(l \in [1, l_{\max}], m \in [-m_{\max}, m_{\max}], n \in [-n_{\max}, n_{\max}]).$$

Trigonometric rules are used to reduce the complexity of computing all values of V_i^{lr} from $\mathcal{O}(N^2)$ to simply $\mathcal{O}(N)$. Using the equality $\cos(a-b) = \cos(a)\cos(b) + \sin(a)\sin(b)$, it appears that for each (l, m, n) triplet the same factors are used for all i:

$$C_{l,m,n} = \sum_j Q_j \cos\left(l\frac{2\pi}{a}x_j + m\frac{2\pi}{b}y_j + n\frac{2\pi}{c}z_j\right)$$
$$S_{l,m,n} = \sum_j Q_j \sin\left(l\frac{2\pi}{a}x_j + m\frac{2\pi}{b}y_j + n\frac{2\pi}{c}z_j\right)$$

In order to avoid the call to the expensive intrinsic function cos and sin, the required values are pre-computed and stored in six arrays:

$$C_{lj} = \cos(l\frac{2\pi}{a}x_j), S_{lj} = \sin(l\frac{2\pi}{a}x_j),$$
$$C_{mj} = \cos(m\frac{2\pi}{b}y_j), S_{mj} = \sin(m\frac{2\pi}{b}y_j),$$
$$C_{nj} = \cos(n\frac{2\pi}{c}z_j), S_{nj} = \sin(n\frac{2\pi}{c}z_j).$$

Then, using trigonometric rules, the values of

$$C_{lmnj} = C_{lj}C_{mj}C_{nj} - S_{lj}C_{mj}C_{nj} - C_{lj}S_{mj}C_{nj} - C_{lj}C_{mj}S_{nj},$$
$$S_{lmnj} = S_{lj}C_{mj}C_{nj} + C_{lj}S_{mj}C_{nj} + C_{lj}C_{mj}S_{nj} - S_{lj}S_{mj}S_{nj}$$

are easily recovered. This trick requires a storage of $2N(l_{\max}+m_{\max}+n_{\max}+3)$ double precision words when we store only values for positive l, m and n. One can note that storing all values of C_{lmnj} and S_{lmnj} would require $2N(2l_{\max}m_{\max} + l_{\max} + m_{\max})(2n_{\max} + 1)$ double precision words.

The long range potential computed in the Metalwalls implementation corresponds to

$$\tilde{V}_i^{\mathrm{lr}} = \frac{8\pi}{abc} \sum_{|k_{lmn}|<k_{\max}} (C_{lmni}C_{lmn} + S_{lmni}S_{lmn}) \frac{e^{-\frac{|k_{lmn}|^2}{4\alpha^2}}}{|k_{lmn}|^2} \tag{3}$$

The listing of the source code corresponding to this implementation is shown in Figure 1. The kernel consists in two loop levels. The outer loop runs on the k-points and two inner loops run on the number of electrode atoms. The first inner loop corresponds to a reduction on the variables `Sk_cos` and `Sk_sin`. The second inner loop uses these values to add the contribution of the current k-point to each electrode atom potential. The two inner loops are executed only if the k-point satisfies the cut-off criterion. The subroutine `compute_kmode_index` computes the (l, m, n) triplet corresponding to a given k-point index.

As an optimization, blocking has been introduced in order to reuse data already loaded into the cache. Indeed, when the k-point index, `imode`, increases, the n index increases the fastest while the l index increases the slowest. Therefore the data loaded from `cos_kx_elec(j,l)`, `sin_kx_elec(j,l)`, `cos_ky_elec(j,m)`, `sin_ky_elec(j,m)` can be reused. Figure 2 shows the listing of the source code corresponding to this optimization. In order, to save some space, parts of the code which are identical to the one presented in Figure 1 has been replaced by ellipses. As can be seen, the kernel is now decomposed into two loop nests of three-level. The outer level corresponds to the blocking, the second level is on the k-point and the innermost level is on the electrode atoms. The size of the innermost loop is controled by a constant named `block_vector_size`. We also have to introduce the two 1D arrays `Sk_cos(:)` and `Sk_sin(:)` of size `num_kpoints` to be able to use the values computed in the first loop nest in the second one.

The computation of the contribution from one k-point is completely independent from the other k-points. The MPI parallelisation strategy is straightforward. Data is replicated on each process and work is distributed. Each process is assigned a range of k-points to work on and a call to `MPI_Allreduce` is made at the end of the kernel to sum up all the k-points contribution into the vector `V(:)`. During the setup phase, k-points which satisfy the cut-off criteria are assigned a weight of 1 and k-points which don't satisfy the cut-off criteria are assigned a weight of 0. The k-points are distributed consecutively to each processes such that all processes work on an equal amount of weighted k-points. Figure 3 shows the skeleton of the implementation for the MPI implementation of this kernel. The loop on the k-point index now runs only the local range, from `imode_start` to `imode_end`. The local and global rep-

Figure 1: Long range potential kernel fortran implementation

```
! Loop on all k-points : num_kpoints = (2*n_max+1)*(2*l_max*m_max+l_max+m_max)
do imode = 1, num_kpoints
  ! Setup the current k-point (l,m,n) triplet
  call compute_kmode_index(imode, l, m, n)

  ! cos_ky/sin_ky are stored only for m>= 0
  mabs = abs(m)
  sign_m = real(sign(1,m), wp)

  ! cos_kz/sin_kz are stored only for n>= 0
  nabs = abs(n)
  sign_n = real(sign(1,n), wp)

  ! Compute the norm squared of the k-point
  kx = real(l,wp) * twopi / a
  ky = real(m,wp) * twopi / b
  kz = real(n,wp) * twopi / c
  knorm2 = kx*kx + ky*ky + kz*kz

  if (knorm2 <= knorm2_max) then

    ! Compute Sk_cos/Sk_sin for this (l,m,n) k-point
    Sk_cos = 0.0_wp
    Sk_sin = 0.0_wp
    do j = 1, num_atoms
      ! load precomputed cos/sin values
      cos_kx = cos_kx_elec(j,l)
      sin_kx = sin_kx_elec(j,l)
      cos_ky = cos_ky_elec(j,mabs)
      sin_ky = sin_ky_elec(j,mabs) * sign_m
      cos_kz = cos_kz_elec(j,nabs)
      sin_kz = sin_kz_elec(j,nabs) * sign_n

      ! Compute cos/sin values using trigonometric rules
      cos_kxky = cos_kx * cos_ky - sin_kx * sin_ky
      sin_kxky = sin_kx * cos_ky + cos_kx * sin_ky
      cos_kxkykz = cos_kxky * cos_kz - sin_kxky * sin_kz
      sin_kxkykz = sin_kxky * cos_kz + cos_kxky * sin_kz

      Sk_cos = Sk_cos + q_elec(j)*cos_kxkykz
      Sk_sin = Sk_sin + q_elec(j)*sin_kxkykz
    end do

    ! For each atom, compute the contribution of this k-point to the potential
    Sk_alpha = 2.0_wp * (4.0_wp*pi/(a*b*c)) * exp(-knorm2/(4*alphasq))/knorm2
    do i = 1, num_atoms
      ! Load precomputed cos/sin values
      cos_kx = cos_kx_elec(i,l)
      sin_kx = sin_kx_elec(i,l)
      cos_ky = cos_ky_elec(i,mabs)
      sin_ky = sin_ky_elec(i,mabs) * sign_m
      cos_kz = cos_kz_elec(i,nabs)
      sin_kz = sin_kz_elec(i,nabs) * sign_n

      ! Compute cos/sin values using trigonometric rules
      cos_kxky = cos_kx * cos_ky - sin_kx * sin_ky
      sin_kxky = sin_kx * cos_ky + cos_kx * sin_ky
      cos_kxkykz = cos_kxky * cos_kz - sin_kxky * sin_kz
      sin_kxkykz = sin_kxky * cos_kz + cos_kxky * sin_kz

      V(i) = V(i) + Sk_alpha * (Sk_cos * cos_kxkykz + Sk_sin * sin_kxkykz)
    end do
  end if
end do
```

Figure 2: Long range potential kernel fortran implementation with blocking

```
Sk_cos(:) = 0.0_wp
Sk_sin(:) = 0.0_wp

num_blocks = (num_atoms-1) / block_vector_size + 1

! Compute Sk_cos/Sk_sin for each k-point
do iblock = 1, num_blocks
  jstart_block = (iblock-1) * block_vector_size + 1
  jend_block = max(jstart_block + block_vector_size - 1, num_atoms)

  ! Loop on all k-points : num_kpoints = (2*n_max+1)*(2*l_max*m_max+l_max+m_max)
  do imode = 1, num_kpoints
    (...)
    if (knorm2 <= knorm2_max) then

      ! Compute block contribution to Sk_cos/Sk_sin for this (l,m,n) k-point
      do j = jstart_block, jend_block
        ! load precomputed cos/sin values
        (...)
        ! Compute cos/sin values using trigonometric rules
        (...)
        Sk_cos(imode) = Sk_cos(imode) + q_elec(j)*cos_kxkykz
        Sk_sin(imode) = Sk_sin(imode) + q_elec(j)*sin_kxkykz
      end do
    end if
  end do
end do

! Compute potential on each electrode atom
do iblock = 1, num_blocks
  istart_block = (iblock-1) * block_vector_size + 1
  iend_block = max(jstart_block + block_vector_size - 1, num_atoms)

  ! Loop on all k-points : num_kpoints = (2*n_max+1)*(2*l_max*m_max+l_max+m_max)
  do imode = 1, num_kpoints
    (...)
    if (knorm2 <= knorm2_max) then
      ! For each atom, compute the contribution of this k-point to the potential
      (...)
      do i = istart_block, iend_block
        ! Load precomputed cos/sin values
        (...)
        ! Compute cos/sin values using trigonometric rules
        (...)
        V(i) = V(i) + Sk_alpha * (Sk_cos(imode)*cos_kxkykz &
                                + Sk_sin(imode)*sin_kxkykz)
      end do
    end if
  end do
end do
```

Figure 3: Long range potential kernel fortran implementation with MPI parallelization

```
Sk_cos(:) = 0.0_wp
Sk_sin(:) = 0.0_wp

num_blocks = (num_atoms-1) / block_vector_size + 1

! Compute Sk_cos/Sk_sin for each k-point
do iblock = 1, num_blocks
  (...)
  ! Loop on local k-points
  do imode = imode_start, imode_end
    (...)
    if (knorm2 <= knorm2_max) then
      ! Compute block contribution to Sk_cos/Sk_sin for this (l,m,n) k-point
      do j = jstart_block, jend_block
        (...)
        Sk_cos(imode) = Sk_cos(imode) + q_elec(j)*cos_kxkykz
        Sk_sin(imode) = Sk_sin(imode) + q_elec(j)*sin_kxkykz
      end do
    end if
  end do
end do

! Compute potential on each electrode atom
do iblock = 1, num_blocks
  (...)
  ! Loop on local k-points
  do imode = imode_start, imode_end
    (...)
    if (knorm2 <= knorm2_max) then
      ! For each atom, compute the contribution of this k-point to the potential
      (...)
      do i = istart_block, iend_block
        (...)
        V_local(i) = V_local(i) + Sk_alpha * (Sk_cos(imode)*cos_kxkykz &
                                            + Sk_sin(imode)*sin_kxkykz)
      end do
    end if
  end do
end do

call MPI_allreduce(V_local(:),V_global(:), num_atoms, &
                   MPI_DOUBLE_PRECISION, MPI_SUM, MPI_COMM_WORLD, ierr)
```

resentation of the potential values `V(:)` is explicitely shown, however one can avoid the allocation of the extra storage by using the `MPI_IN_PLACE` tag in the call to `MPI_Allreduce`. The `Sk_cos(:)` and `Sk_sin(:)` arrays are local to each process and thus they only need to be allocated for the range `(imode_start:imode_end)`.

Figure 4: $k = 0$ potential kernel fortran serial implementation

```
do i = 1, num_atoms
  do j = 1, i
    zij = z(j) - z(i)
    zijsq = zij * zij
    pot_ij = volfactor * (sqrpialpha * exp(-zijsq*alphasq) &
                          + pi * zij * erf(zij*alpha))
    V(j) = V(j) - q_elec(i) * pot_ij
    V(i) = V(i) - q_elec(j) * pot_ij
  end do
end do
```

2 $k = 0$ potential kernel

The $k = 0$ potential equation is derived from the energy equation as $V_i^{k=0} = \frac{\partial U_c^{\mathrm{lr},0}}{\partial qi}$. Using the formula given in the Ewald summation document, this yields:

$$V_i^{k=0} = -\frac{2\sqrt{\pi}}{ab} \sum_j Q_j \left(\frac{e^{-z_{ij}^2 \alpha^2}}{\alpha} + \sqrt{\pi} |z_{ij}| \operatorname{erf}(\alpha |z_{ij}|) \right) \tag{4}$$

The $k = 0$ potential on an electrode atom is the sum of the contribution from all other atoms in the system. One can note that for a pair of atoms $\frac{V_{ij}^{k=0}}{Q_j} = \frac{V_{ji}^{k=0}}{Q_i}$ and therefore the computation of the potential on each electrode atom can be implemented as a triangular loop as shown in Figure 4.

In order to improve the data locality and the vectorization potential, the triangular loop on atom pairs has been reformulated into a loop of block of atoms pairs. The block size parameter is a constant parameter chosen to be a multiple of the vector length. Figure 5 shows the order used to compute pair interaction in the triangular loop, while Figure 6 shows the ordering used in the blocked version. All blocks are treated fully, even blocks on the diagonal for which we don't use the symmetry of the interaction. For small values of the block size parameter, the extra work is compensated by an increase in vectorization potential. The corresponding implementation is shown in Figure 7.

From this point, parallelisation in MPI is straightforward. Using a data replication strategy, the work is distributed by blocks. Each process computes a contribution from a subset of the blocks and a call to `MPI_Allreduce` is used to add the contribution from all blocks to the potential. The implementation is shown in Figure 8.

Figure 5: Pair ordering for triangular loop nest (`num_atoms`=12)

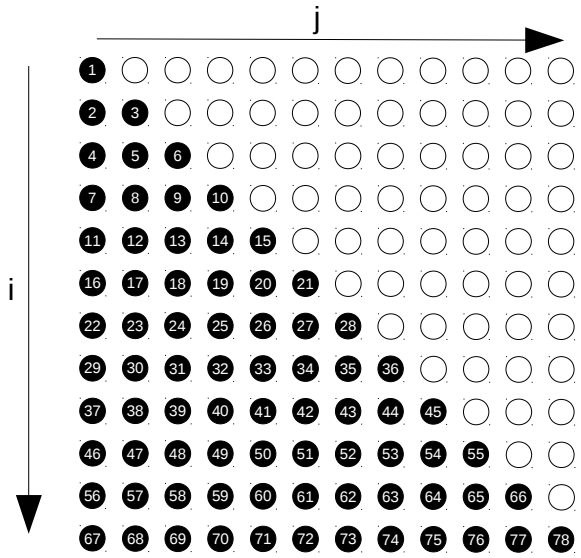

Figure 6: Pair ordering for block loop nest

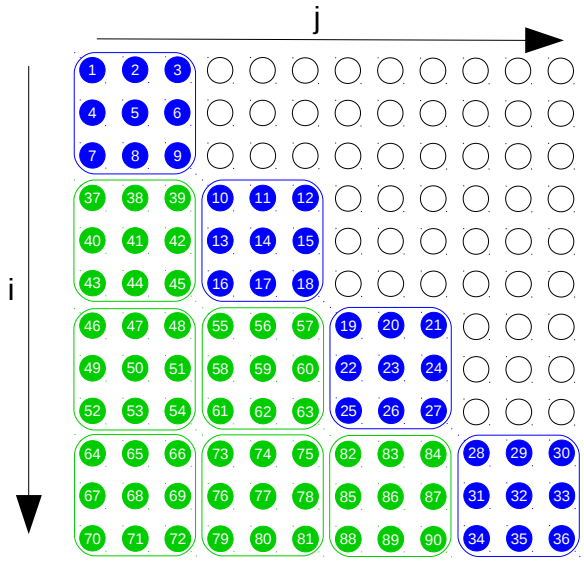

Figure 7: $k = 0$ potential kernel fortran blocked implementation

```
! Compute contribution from blocks on the diagonal
do iblock = 1, num_block_diag
  call compute_diag_block_boundaries(iblock, istart, iend)
  do i = istart, iend
    do j = istart, iend
      zij = z(j) - z(i)
      zijsq = zij * zij
      pot_ij = volfactor * (sqrpialpha * exp(-zijsq*alphasq) &
                           + pi * zij * erf(zij*alpha))
      V(i) = V(i) - q_elec(j) * pot_ij
    end do
  end do
end do

! Compute contribution from blocks below the diagonal
do iblock = 1, num_block_full
  call update_tri_block_boundaries(iblock, istart, iend, jstart, jend)
    do i = istart, iend
      do j = jstart, jend
        zij = z(j) - z(i)
        zijsq = zij * zij
        pot_ij = volfactor * (sqrpialpha * exp(-zijsq*alphasq) &
                             + pi * zij * erf(zij*alpha))
        V(j) = V(j) - q_elec(i) * pot_ij
        V(i) = V(i) - q_elec(j) * pot_ij
      end do
    end do
  end do
end do
```

Figure 8: $k = 0$ potential kernel fortran parallel implementation

```
! Compute contribution from blocks on the diagonal
do iblock = istart_block_diag, iend_block_diag
  call compute_diag_block_boundaries(iblock, istart, iend)
  do i = istart, iend
    do j = istart, iend
      (...)
      V_local(i) = V_local(i) - q_elec(j) * pot_ij
    end do
  end do
end do

! Compute contribution from blocks below the diagonal
do iblock = istart_block_full, iend_block_full
  call update_tri_block_boundaries(iblock, istart, iend, jstart, jend)
    do i = istart, iend
      do j = jstart, jend
        (...)
        V_local(j) = V_local(j) - q_elec(i) * pot_ij
        V_local(i) = V_local(i) - q_elec(j) * pot_ij
      end do
    end do
  end do
end do

call MPI_Allreduce(V_local(:), V_global(:), num_atoms, &
                   MPI_DOUBLE_PRECISION, MPI_SUM, MPI_COMM_WORLD, ierr)
```

3 Short Range potential kernel

The short range potential equation is derived from the energy equation as $V_i^{\mathrm{sr}} = \frac{\partial U_c^{\mathrm{sr}}}{\partial qi}$. Using the formula given in the Ewald summation document this yields:

$$V_i^{\mathrm{sr}} = \sum_j Q_j \sum_{\boldsymbol{n}}{}' |\boldsymbol{r_{ij}} + \boldsymbol{n}|^{-1} \left(\mathrm{erfc}(\alpha|\boldsymbol{r_{ij}} + \boldsymbol{n}|) - \mathrm{erfc}(\eta_{ij}|\boldsymbol{r_{ij}} + \boldsymbol{n}|)\right) \qquad (5)$$

In Equation (5), the sums run on all particles and all of their images. The $'$ indicates that the self-interaction term should be ommited. However, we cannot compute infinite sums on a computer. A cut-off distance distance r_{cut} is imposed and the minimum image distance convention is used.

The structure of the short range potential kernel is very close to the $k = 0$ potential. The interaction are symmetric and thus involve a triangular loop. The main difference, apart from the actual value, is the presence of a cut-off radius and the absence of self-interaction which may hinder vectorization. However the same blocking and parallelisation strategy is used as in the other kernel. The Figure 9 shows the fortran implementation of the kernel.

Knowing that electrode atoms have a constant position during the whole simulation. It is possible to distribute only blocks for which we know, from a setup phase computation, where there will be some interactions between particles in those blocks.

Figure 9: Short-range potential kernel fortran parallel implementation

```
! Compute contribution from blocks on the diagonal
do iblock = istart_block_diag, iend_block_diag
  call compute_diag_block_boundaries(iblock, istart, iend)
  do i = istart, iend
    do j = istart, iend
      call minimum_image_distance(x(i), y(i), z(i), x(j), y(j), z(j), drnorm2)
      if (drnorm2 < rcut_sq) then
        drnorm = sqrt(drnorm2)
        pot_ij = (erfc(alpha*drnorm) - erfc(eta*drnorm)) / drnorm
        V_local(i) = V_local(i) + q_elec(j) * pot_ij
      end if
    end do
  end do
end do

! Compute contribution from blocks below the diagonal
do iblock = 1, num_block_full
  call update_tri_block_boundaries(iblock, istart, iend, jstart, jend)
    do i = istart, iend
      do j = jstart, jend
        call minimum_image_distance(x(i), y(i), z(i), x(j), y(j), z(j), drnorm2)
        if (drnorm2 < rcut_sq) then
          drnorm = sqrt(drnorm2)
          pot_ij = (erfc(alpha*drnorm) - erfc(eta*drnorm)) / drnorm
          V_local(j) = V_local(j) + q_elec(i) * pot_ij
          V_local(i) = V_local(i) + q_elec(j) * pot_ij
        end if
      end do
    end do
  end do
end do

call MPI_Allreduce(V_local(:), V_global(:), num_atoms, &
                   MPI_DOUBLE_PRECISION, MPI_SUM, MPI_COMM_WORLD, ierr)
```

4 Self potential kernel

The short range potential equation is derived from the energy equation as $V_i^{\mathrm{sr}} = \frac{\partial U_c^{\mathrm{self}}}{\partial qi}$. Using the formula given in the Ewald summation document this yields:

$$V_i^{\mathrm{self}} = \frac{2}{\sqrt{\pi}} Q_i \left(\frac{\eta_i}{\sqrt{2}} - \alpha \right) \tag{6}$$

The self potential kernel is straightforward to implement. It is simply a vector scaling. It will not be discussed further in this document.