dCSE Project on Improving Automatic Load Balancing and Molecular Dynamics in Conquest

Lianheng Tong

Wednesday, 2013/04/17

Abstract

This report describes the work done in the distributed Computational Science and Engineering (dCSE) project (6 months) aimed to improve the molecular dynamics capabilities of the linear scaling ab initio Density Functional Theory code Conquest. The work is split into two parts: the first concerns the initial load balancing, and the second concerns the implementation of dynamic reassignment of simulation data among the processors for molecular dynamics (MD). For the first part of this project, a flexible non-cubic Hilbert space-filling-curve has been successfully developed, which can generate compact 3D Hilbert curves with arbitrary and distinct number of recursions in (x, y, z) directions. A new automated partitioner and load balancing algorithm has been successfully developed and tests have shown significant improvements in initial load balancing over the existing implementation. For the second part of the project, all the required implementations for dynamically reassigning the indices and matrix elements have successfully been implemented. The new MD allows the reuse of simulation data from the previous steps, which reduced computational time by 40%–90%. However, tests have also shown that reusing the simulation data from previous steps introduced an energy drift during MD simulation, which has become the subject of further studies.

1 Introduction

The use of quantum mechanical techniques to simulate the structure and properties of systems has become a cornerstone of modern science. Commonly used DFT[5] codes like CASTEP and VASP, which form a significant proportion of HECToR use, all have a computational effort which scales with the cube of the number of atoms, N, (N^3 scaling) and memory requirements which scale with N^2. This scaling limits the size of system which can be modelled to a few thousand atoms at most, even on the largest HPC platforms, as well as limiting parallel scaling. Within biochemistry, it is common practice to embed small quantum mechanical simulations within classical force fields to circumvent this problem.

Over the last fifteen years, significant effort has gone into the development of linear scaling DFT codes, where the computational effort and memory requirements scale linearly with the system size. Conquest is one of the linear scaling code available on HECToR that has demonstrated the scalability necessary to go beyond 10000 atoms and several hundred cores efficiently[2].

This project focuses on improving the load balancing of Conquest, to enable its use for MD simulations on 10,000 to 100,000 atoms and beyond. This will involve two areas: the initial distribution of computational responsibility across the processors; and dynamically redistribution of load and data as the simulation progresses.

For initial load distribution, CONQUEST currently uses three methods: an automated scheme with CONQUEST, which uses space filling curves[3], an external python utility which divides the simulation cell and assigns system data to the processors according to user parameters, which is commonly used if the automated scheme was deemed insufficient, and another external code, based on simulated annealing, which is less commonly used. The automated scheme employed the use of Hilbert curves[7], which had the advantage of being compact in space. This means system data (such as atoms and grid points) close together in space (and thus be more likely to be involved in computation together) will be more likely assigned to the same processor, thus reducing amount of communications required. The standard Hilbert curve and space filling curve in general also imposes a rigid constraint on the automated load distributor in that all divisions of the simulations cell must be the same in all three directions (that is cubic), and the divisions must be in the power of 2. This constraint makes the automated scheme ill suited for systems with cells far from being cubic, such as slabs and thin columns. The proposed improvement to the automatic scheme is thus to find a way to relax the cubic constraint on the Hilbert curves, while still retaining the compactness of the curve.

Due to the complex matrix element indexing and storage systems used by CONQUEST for highly optimised linear scaling matrix operations, matrix storage formats and indices are dependent on the positon of atoms in the simulation cell. This makes implementation of molecular dynamics in CONQUEST a challenging problem. CONQUEST at the present implements a simple Born-Oppenheimer molecular dynamics algorithm which assumes the atoms remain assigned to the same partition (sub-division of the simulation cell) through-out the calculation. While this is fine for small atomic movements, the code is unable to address the situations when an atom has move sufficiently far to leave its original partition, as neighbour lists point to incorrect locations. Further more, due to the fact that atomic movements causes neighbour list changes, and the processor ownership of matrix elements are dependent on locations of the atoms, the sparse matrix formats and size on each processor change as simulation progresses. The current CONQUEST avoids this problem by resetting the matrices and restart the calculations from the start, treating every time step as an initial calculation. While this will always produce the correct Born-Oppenheimer results, it is not an efficient way to perform molecular dynamics calculations. This project is aimed to tackle these problems head-on, and implement dynamic reassignment of indices and reorder matrix elements at every molecular dynamics step, so that CONQUEST can perform stable molecular dynamics simulations irrespective of size of atomic movements, and be able to reuse calculated matrix data in the next simulation steps.

This work done in this project will be very likely to benefit wide range of codes available on HECToR and beyond. The principles used in the flexible space-filling-curve automated load balancing scheme can be easily transfered to other parallel codes which relies on domain decomposition to distribute data to processors. The implementation done on molecular dynamics for CONQUEST will add new capabilities in the code, and the ability to do ab initio quantum mechanical molecular dynamics on 10000+ atoms for up to 2–5 ps will open doors to new scientific studies which would not have been possible before.

In The rest of this report, methods used for implementing the two objectives stated above will be discussed, together with relevant test data.

2 Automatic Partitioning Scheme

CONQUEST divides the simulation cell into equal sized partitions, which forms the basic blocks for the assignment of atomic data to the processors[1]. If no user defined real space integration grid block distributions are given then the code distribute the real space grid point to the processors according to the partitions. This means the processor in charge of a particular partition will also be responsible for the set of grid points blocks inside that partition. A good partitioning scheme is therefore crucial for the load balance of CONQUEST.

The current implementation of CONQUEST estimates the partition cell sizes based on the maximum number of atoms $N_{a,\mathrm{part}}^{\mathrm{max}}$ allowed in a partition. $N_{a,\mathrm{part}}^{\mathrm{max}}$ is an user definable parameter. Starting from an initial guess assuming uniform distribution of atoms, the code gradually increases the number of partitions in each dimension, until the maximum number of atoms in partitions does not exceed $N_{a,\mathrm{part}}^{\mathrm{max}}$. Once the desirable partitions are found, a standard 3D Hilbert curve is used to "thread" through the partitions converting the three tuples `(ix, iy, iz)` of the partition coordinates into an one dimensional integer on the Hilbert curve. The 1D integer index is then used for assigning the partitions to each processors, weighted by the number of atoms in each partition—so that the processor gets roughly similar number of atoms each.

The use of the Hilbert curve to thread through the partitions has a distinct advantage that the Hilbert curve is compact in space. This means that irrespective to the number of divisions used for partitioning the cell, the partitions that are near to each other in space are near to each other on the Hilbert curve. This means, once the partitions are assigned to the processors, the group of atoms belonging to the same processor will likely be close in space, and group of atoms belonging to partitions that are further apart on the Hilbert curve—and thus assigned to different processors—will be likely to be further apart in space. This reduces the amount of MPI communications during calculation, as only interactions between atoms within a finite range are considered.

There is however also a disadvantage in using the standard 3D Hilbert curves for indexing the partitions. Due to the recursive nature of Hilbert curve generation, the number of partitions along each of the three cell dimensions must be the same, and are powers of 2. If the simulation cell is far from cubic, with one or two sides significantly longer than the rest, then too many partitions may have to be taken in the shortest dimension. At the best this creates many empty partitions which slows down the calculation; at worst, this causes significant load imbalances which eventually lead to the code crushing—examples of this happening is presented this report.

The aim of this part of the dCSE project is to find a way of construct a 3D Hilbert curve that allows different levels of recursions in different spacial dimensions.

2.1 Non-Cubic Hilbert Curve

The goal is to find a method to generate a non-cubic 3D Hilbert curve which allows the levels of recursions N_x, N_y and N_z in x, y and z direction which can take any distinct value in range $0, 1, 2, \cdots$. A solution has been successfully found, and is presented in this section.

The key to the solution is to break the non-cubic curve into three levels. The inner most level contains standard cubic 3D Hilbert curves with the level of recursion $N^C = \min(N_x, N_y, N_z)$; the middle level contains square 2D Hilbert curves with the level of recursion N^P equal to the second highest among N_x, N_y, N_z minus N^C; and the upper level contains a 1D Hilbert curve (a straight

[1] In this report, a "processor" is equivalent to a MPI process. An MPI process may create more than one threads and use one or more physical cores, but in this report a processor strictly means a MPI node,

line), with recursion $N^L = \max(N_x, N_y, N_z) - N^P - N^C$. Figure 1 illustrates the concept using an example of generation of non-cubic Hilbert curve with $N_x = 3$, $N_y = 2$ and $N_z = 1$. Within this approach, each partition cell will be a point on the N^L-th order cubic 3D Hilbert curve; each 3D curve will be a point on the N^P-th order 2D Hilbert curve; and each 2D curve will be a point on the N^L-th order 1D Hilbert curve. In total, the generated Hilbert curve will go through $2^{N_x} \times 2^{N_y} \times 2^{N_z}$ number of partition cells. If $N_x = N_y = N_z$ then the standard N_x-th order cubic 3D curve is obtained.

The 1D, 2D and 3D Hilbert curves are generated by doing recursions on the basic 1D, 2D and 3D unit Hilbert curves, with appropriate orientations. The unit curve are generated using reflective Gray code method[4].

The n-bit Gray codes represent the spacial coordinates of the points in the unit Hilbert curve—with coordinates have values 0 or 1—in n-dimensions. The transformation from Gray code to/from binary code thus corresponds to the mapping from/to the spacial coordinates to/from the sequencial indices in an unit Hilbert curve. This can be done very efficiently through bit-wise operations (see for example [8]).

The orientation of a unit Hilbert curve can be defined by specifying the start and end coordinates (Gray codes) of the Gray code sequence. For $n < 3$ dimensions, start and end coordinates uniquely define the Hilbert curve. For 3D, there are two distinct unit Hilbert curves for every pair of start and end points. For the problem associated with this project, finding either one of the 3D unit curve giving the desired travel is sufficient. The canonical n-bit reflective Gray code sequence starts from 0 and travels to 2^{n-1}. A more general Gray code sequence (i.e. unit Hilbert curve) may be generated by first rotating the bits in the canonical code sequence, so that the sequence travels from 0 to t, where t is the travel direction given as (start `XOR` end); then `XOR` the sequence with the desired start Gray code. The standard Hilbert curves can be generated by recursively generating unit curves, with the travel direction of the child unit curve at a given point on the parent curve equal to the direction of the next point on the parent curve in relation to the given point.

To generate the non-cubic Hilbert curve, both the integer sequencial index i of the coordinates on the path and the spacial coordinates (x, y, z) are decomposed into new representations:

$$i \to (l, p, c)$$
$$(x, y, z) \to (L, P, C)$$

where, $l = (b_1^l, b_2^l, \cdots, b_{N^L}^l)$ is the decomposed index bundle of the point on the outer 1D Hilbert curve, with b_i^l corresponding to the sequence indices of the the corresponding i-th recursion of the n-dimensional (1D for l) unit Hilbert curve. b_1^l is the index for outer most level, and $b_{N^L}^l$ innermost. So l corresponds to the $b_{N^L}^l$-th point of the inner-most child curve at the $b_{N^L-1}^l$-th point of its parent curve, which is at the $b_{N^L-2}^l$-th point of its parent curve and so on. The similar applies to $p = (b_1^p, b_2^p, \cdots, b_{N^P}^p)$ representing a point on the middle level 2D curve located at point l on the 1D curve; and $c = (b_1^c, b_2^c, \cdots, b_{N^C}^c)$ representing a point on the inner 3D curve, located at point p on the 2D curve. The same goes for the spacial indices, with $L = (g_1^l, g_2^l, \cdots, g_{N^L}^l)$, $P = (g_1^p, g_2^p, \cdots, g_{N^P}^p)$ and $C = (g_1^c, g_2^c, \cdots, g_{N^L}^c)$ corresponding to the Gray codes representing the spacial coordinates of the unit Hilbert curve at different levels of recursion.

The Hilbert curve is then generated from top level (starting from the 1D curve L) downwards, working out the correct rotation (start and end positions) of each child curve. Therefore, given Hilbert index i:

1. i is decomposed to (l, p, c)
2. (L, P, C) is worked out from top level downwards using binary to Gray code transformations
3. (L, P, C) is then transformed back into (x, y, z)

The code implemented is given below:

```
subroutine Hilbert3D_IntToCoords(ind, coords)
  implicit none
  ! Passed parameters
  integer, intent(in) :: ind
  integer, dimension(:), intent(out) :: coords
  ! Local variables
  type(unpacked) :: unpkd_ind, unpkd_coords
  integer :: index, c_start, c_end, start, end, ii
  integer :: l_ind, p_ind, c_ind
  ! ind goes starts from 1, transform to index which starts from 0
  index = max(0, ind - 1)
  ! unpack ind, unpkd_ind is allocated by UnpackIndex
  call UnpackIndex(index, unpkd_ind)
  ! allocate the unpacked coordinates with same dimension as unpkd_ind
  call AllocUnpacked(unpkd_coords, unpkd_ind%N_l_levels, &
                     unpkd_ind%N_p_levels, unpkd_ind%N_c_levels)
  ! zero the l, p and c levels for unpkd_coords
  unpkd_coords%l = 0
  unpkd_coords%p = 0
  unpkd_coords%c = 0
  ! move along the outer-mose level of the line
  start = 0
  end = 1
  if (DimLine > 1) then
     ! recursively generate the 1D fractal coordinates
     do ii = 1, unpkd_coords%N_l_levels
        l_ind = unpkd_ind%l(ii)
        unpkd_coords%l(ii) = GrayEncodeTravel(start, end, 1, l_ind)
        ! from parent start and end calculate child (c_)start and
        ! (c_)end at point l_ind of the parent curve
        call ChildStartEnd(start, end, 1, l_ind, c_start, c_end)
        start = c_start
        end = c_end
     end do
  end if
  ! move in the middle level squares, the highest level unit square
  ! always move along the direction of the higher level 1D line
  start = start * 2 ! need to convert from 1 to 10 (1D to 2D)
  end = end * 2
  if (DimSquare > 1) then
     do ii = 1, unpkd_coords%N_p_levels
        p_ind = unpkd_ind%p(ii)
        unpkd_coords%p(ii) = GrayEncodeTravel(start, end, 2, p_ind)
        call ChildStartEnd(start, end, 2, p_ind, c_start, c_end)
```

```
        start = c_start
        end = c_end
     end do
  end if
  ! move in the inner level cubes, the highest level unit cube
  ! follows the direction of the children of the upper level unit
  ! square, unless dimension of 2D curve is 1, in which case it
  ! follows the direction of the line
  start = start * 2 ! need to convert from 10 to 100 (2D to 3D)
  end = end * 2
  if (DimCube > 1) then
     do ii = 1, unpkd_coords%N_c_levels
        c_ind = unpkd_ind%c(ii)
        unpkd_coords%c(ii) = GrayEncodeTravel(start, end, 3, c_ind)
        call ChildStartEnd(start, end, 3, c_ind, c_start, c_end)
        start = c_start
        end = c_end
     end do
  end if
  ! pack unpkd_coords to give the cartesian coordinates
  call PackCoords(unpkd_coords, coords)
  ! deallocate memory
  call DeallocUnpacked(unpkd_ind)
  call DeallocUnpacked(unpkd_coords)
end subroutine Hilbert3D_IntToCoords
```

Similarly, given coordinates (x, y, z)

1. (x, y, z) is decomposed to (L, P, C)
2. Using Gray code to binary transformations, working from top level downwards, get the corresponding (l, p, c)
3. (l, p, c) is then transformed back into i

The code implemented is given below:

```
subroutine Hilbert3D_CoordsToInt(coords, ind)
  implicit none
  ! Passed parameters
  integer, dimension(:), intent(in) :: coords
  integer, intent(out) :: ind
  ! Local variables
  type(unpacked) :: unpkd_coords, unpkd_ind
  integer :: index, start, end, c_start, c_end, ii
  integer :: l_coords, p_coords, c_coords
  call UnpackCoords(coords, unpkd_coords)
  call AllocUnpacked(unpkd_ind, unpkd_coords%N_l_levels, &
                     unpkd_coords%N_p_levels, unpkd_coords%N_c_levels)
  ! zero the l, p and c levels for unpkd_ind
  unpkd_ind%l = 0
```

```
  unpkd_ind%p = 0
  unpkd_ind%c = 0
  ! move along the outer level 1D Hilbert curve
  start = 0
  end = 1
  if (DimLine > 1) then
     do ii = 1, unpkd_coords%N_l_levels
        l_coords = unpkd_coords%l(ii)
        unpkd_ind%l(ii) = GrayDecodeTravel(start, end, 1, l_coords)
        call ChildStartEnd(start, end, 1, unpkd_ind%l(ii), c_start, c_end)
        start = c_start
        end = c_end
     end do
  end if
  ! move in the middle level 2D Hilbert curve, the highest level
  ! square always has travel along direction of the line
  start = start * 2 ! need to convert from 1 to 10 (1D to 2D)
  end = end * 2
  if (DimSquare > 1) then
     do ii = 1, unpkd_coords%N_p_levels
        p_coords = unpkd_coords%p(ii)
        unpkd_ind%p(ii) = GrayDecodeTravel(start, end, 2, p_coords)
        call ChildStartEnd(start, end, 2, unpkd_ind%p(ii), c_start, c_end)
        start = c_start
        end = c_end
     end do
  end if
  ! move in the lower level 3D Hilbert curve, the highest level cube
  ! follows the direction of children of the 2D curve, unless
  ! dimension of 2D curve is 1, in which case it follows the
  ! direction of the 1D curve
  start = start * 2 ! need to convert from 10 to 100 (2D to 3D)
  end = end * 2
  if (DimCube > 1) then
     do ii = 1, unpkd_coords%N_c_levels
        c_coords = unpkd_coords%c(ii)
        unpkd_ind%c(ii) = GrayDecodeTravel(start, end, 3, c_coords)
        call ChildStartEnd(start, end, 3, unpkd_ind%c(ii), c_start, c_end)
        start = c_start
        end = c_end
     end do
  end if
  call PackIndex(unpkd_ind, index)
  ! index starts counting from 0, so correct for ind which counts
  ! from 1
  ind = index + 1
  ! deallocate memory
  call DeallocUnpacked(unpkd_ind)
  call DeallocUnpacked(unpkd_coords)
end subroutine Hilbert3D_CoordsToInt
```

2.2 Load Balancing

The original CONQUEST implementation assigns partitions to the processors and tries to balance the load by weighting on either

1. Number of partitions (`General.LoadBalance` = `partitions`)
2. Number of atoms (`General.LoadBalance` = `atoms` [default])

Generally speaking, load balancing by weighting on the number of partitions per processor is not recommended, unless the atoms are evenly distributed in the simulation cell. Even in that case, the number of atoms located inside each partition may vary, which results in poor load balance. A processor can also potentially get all empty partitions, in which case the code will fail to run.

For simulation systems with diverse number of species such as bio-molecules, weighting with respect to the number of atoms per processor may still lead to poor load balance, as the number of support functions—the actual computational load—differs significantly between different species.

The original partitioning algorithms in CONQUEST was written with the standard 3D Hilbert curve codes deeply embedded, and with the initial assumption that the level of recursions in all three dimensions are the same. This makes modifying the existing partitioning code to accommodate the new non-cubic Hilbert curves impractical. A completely new partitioning and load distribution module was therefore created in CONQUEST. Taking advantage of this opportunity, new features were added to the code so that:

- The shape of the system under consideration is now being taken into account in the automatic partitioner
- Load assignment to processors can now weight on the number of support functions per processor
- The user can now specify the number of partitions along which dimensions are to be set manually and which are to be determined automatically by the automatic partitioner. The original code only allows user to either let the code to determine the number of partitions fully automatically or set the partitions manually by supplying a partition map file generated from an external code

2.2.1 Determining System Shape

The simulation systems are classified into four categories, with the automatic partitioner behaving differently corresponding to the system type:

1. **Bulk**: where the atoms distribute nearly all parts of the simulation cell. The automatic partitioner will try to generate partitions that has edges of roughly equal lengths in all directions
2. **Slab**: where the atoms are distributed in a narrow slab either in the middle or at the two ends of the simulation cell. In the latter case, the system is assumed to be wrapped around by periodic boundary conditions. The automatic partitioner will not divide the simulation cell in the dimension perpendicular to the slab plane, so that a lot of empty partitions can be avoided, and try to generate the partitions so that each partition will have its edges parallel to the slab plane with roughly equal lengths.

3. **Chain**: where the atoms form a narrow column along either x, y or z directions in the cell. The column can either be in the middle of the cell, or be wrapped into 2 or 4 parts along the simulation cell edges due to periodic boundary conditions. In this case the automatic will generate partitions along the longitudinal direction of the chain. No divisions of the simulation cell will be made in the transverse directions.

4. **Molecule**: where the atoms are grouped together only in a small ($< 50\%$ in each directions) part of the cell. They can be either grouped as one part in the middle of the cell or become several parts spread around the simulation cell edges due to periodic boundary conditions. In this case the automatic partitioner will try to form partitions that has edges of roughly equal lengths in all directions, the same way as for the bulk systems.

The key to determination of the system type lies in the calculation of gaps in the cell where there are no atoms. The algorithm will try to determine the largest gap in the cell, and if the largest gap is in between atoms—that is, not touching the simulation edge in at least one of the 3 cartesian directions—then the atoms are assumed to be wrapped around by periodic boundary conditions. The system types are then determined as follows

- Treat each of the x, y and z directions separately, loop over the three directions:
  - If the largest gap is in the middle of the cell (in between atoms), and if the gap is greater or equal to 0.5 of the cell dimension in this direction, then treat the system as "hollow" in this direction
  - Else if the largest gap is on one side of the cell, then we calculate the extent of the atoms in this direction. If the atomic extent is less than 0.5 of the cell dimension in this direction, then the system is treated as "hollow" in this direction
- If no direction is "hollow" then the system is a bulk.
- Else if exactly one direction is "hollow" then the system is a slab
- else if exactly two directions are "hollow" then the system is a chain
- else if exactly three directions are "hollow" then the system is a molecule

Since atoms are points in space, in theory gaps will be everywhere. To make the calculation practical, along each direction, the system is divided into at most 10 segments. The minimum allowed width of the segment must not be lower than a user controllable parameter denoting the average atomic diameters (new input flag: `General.AverageAtomicDiameter`, defaults to 5.0 a0). The average atomic diameter parameter is used so that for small simulation cells, the segments do not become too small. The segments are then looped over along each directions, with the total number in the transverse plane counted. If no atoms were found, then the segment is assumed to be empty. One or more consecutive empty segments corresponds to a gap in the simulation cell. The actual gap width is then calculated by calculating the extent of atoms in within the limits of the segments just before or after the found gap:

- If the gap is not in the middle of the cell, then the gap is the distance between the minimum or maximum extent of the atom to the bottom or top edges of the simulation box along the given direction

- Else if the gap is in the middle of the cell, then gap is equal to the minimum extent of the atoms above the gap and the maximum extent of the atoms below the gap.

This way the extent, gap and type of the system under simulation can be quite robustly estimated, and this information are fed into the automatic partitioning subroutines so that the simulation cell is divided accordingly.

2.2.2 Initial Guess on Partition Cell Sizes

Following the approach used in the original CONQUEST implementation, the size of the partition cells are determined by limiting the number of atoms allowed to be contained in each partition.

The original approach uses the user defined parameter `General.MaxAtomsPartition` as the upper limit of the number of atoms allowed in each partition, In the new approach, the upper limit is determined by

```
max_natoms_part = ni_in_cell / numprocs
! ni_in_cell is total number of atoms in simulation cell
max_natoms_part = max(max_natoms_part, 1)
max_natoms_part = min(max_natoms_part, global_maxatomspart)
! global_maxatomspart is set by General.MaxAtomsPartition
```

so that the number of occupied partitions is never less than the number of processors. If this is not the case then some of the processors cannot be assigned atoms, and the code will fail.

The initial guess on the partition cell sizes are determined by:

1. If the number of partitions in particular directions is fixed and predetermined. If fixed then initial guess on the number of partitions in that direction is taken as the set value, no resizing will be performed in this direction. The number of partitions in a direction is fixed when either (or both) of the following happens:
   - The user sets it manually in `Conquest_input`
   - The system is either slab or chain, and number of partitions along this particular direction is set to 1.
2. For directions along which the number of partitions are not fixed, the partition sizes are determined by:
   - Calculate the volume of the occupied cell V_{occ}
   - Estimate volume of partition to be $V_{\text{part}} = V_{\text{occ}} \frac{\text{max\_natoms\_part}}{\text{ni\_in\_cell}}$
   - If three directions are not fixed then the sizes of the partition cell along all directions are estimated to be roughly equal: $r^x_{\text{part}} = r^y_{\text{part}} = r^z_{\text{part}} = \sqrt[3]{V_{\text{part}}}$
   - Else if two directions are not fixed, then partition sizes along these two directions u, v are estimated to be roughly equal: $r^u_{\text{part}} = r^v_{\text{part}} = \sqrt{\frac{V_{\text{occ}}}{\text{fixed\_part\_dim}}}$
   - Else if only one direction u is not fixed, then $r^u_{\text{part}} = \frac{V_{\text{occ}}}{\text{fixed\_part\_dim1}*\text{fixed\_part\_dim2}}$
3. The actual initial estimation of partition sizes must divide up the simulation cell exactly into 2^{N_x}, 2^{N_y} and 2^{N_z} parts along the three axes, therefore the partition sizes are recalculated using

```
! n_parts: number of partitions in 3 directions
! n_divs: number of divisions of simulation cell in 3 directions
! r_part: partition cell size in 3 directions
! FSC: the (fundamental) simulation cell
do ii = 1, 3
   n_parts(ii) = max(nint(FSC%dims(ii) / r_part(ii)), 1)
   n_divs(ii) = int_log2(n_parts(ii))
   n_parts(ii) = 2**n_divs(ii)
   r_part(ii) = FSC%dims(ii) / real(n_parts(ii),double)
end do
```

2.2.3 Refining Partition Cell Sizes

The initial estimation of partition sizes are based on the assumption that atoms are uniformly spread in the occupied region of the simulation cell. This is rarely the case in practice. Hence the partition sizes needs to be refined so that the number of atoms in each partition stay within the limit of `max_natoms_part`. The refinement process is given as follows:

1. Set the number of atoms and support functions in partitions to 0.
2. Loop over all atoms in the simulation cell, the (i_x, i_y, i_z) integer coordinates of the partition containing the particular atom can be obtained from

$$i_x = \text{floor}\left(r^x_{\text{atom}}/r^x_{\text{part}}\right) \tag{1}$$

$$i_y = \text{floor}\left(r^y_{\text{atom}}/r^y_{\text{part}}\right) \tag{2}$$

$$i_z = \text{floor}\left(r^z_{\text{atom}}/r^z_{\text{part}}\right) \tag{3}$$

   - From the (i_x, i_y, i_z) coordinates of the partition, work out the corresponding non-cubic Hilbert curve index i_{Hilbert}, using method presented in section 2.1.
   - Increment the atom count and the associated number of support functions in the partition i_{Hilbert}.
   - If the number of atoms in a particular partition is greater than `max_natoms_part`:
   - Refine partitions: increase the number of divisions—Hilbert curve recursion—N_u of the direction `(u)` that has the longest partition cell size: $r^u_{\text{part}} = \max(r^x_{\text{part}}, r^y_{\text{part}}, r^z_{\text{part}})$. The number of partitions in direction u is thus doubled to $2^{(N_u+1)}$.
   - exit the atom loop and return to step 1.

The process repeats until all partitions has number of atoms less than `max_natoms_part`. During this process, other important information regarding to the partitions are gathered, such as the Cartesian Composite (CC) indices of the partition:

$$i_{\text{CC}} = ix r^y_{\text{part}} r^z_{\text{part}} + iy r^z_{\text{part}} + iz$$

During the actual calculations, it is the CC indices rather than the Hilbert indices that are used to reference the partitions. Also from the CC indices calculated, work out the atomic index corresponding to the partitions (see reference [1]) and store the map between the partition atomic indices with global (simulation cell) atomic indices. These are important book-keeping information allowing each processor to identify local and remote atoms.

2.2.4 Assignment of Partitions to Processors

Once the simulation have been correctly partitioned and indexed on the Hilbert curve, the next step is to assign the partitions along the Hilbert curve to processors. The assignment is sequencial, in the sense that each processor always gets a chunk of continuous sequence of partitions on the Hilbert curve. This ensures the load assignment is taking advantage of the compactness of the Hilbert curves.

The aim of the assignment process is to let each processor to have roughly equal amount of data, either weighted in terms of the number of partitions, number of atoms or the number of support functions. The assignment process is best illustrated in figure 2.

- First, the minimum amount of data (partitions, atoms or support functions) per processor N/P is estimated to be the global average per processor. The partitions are assigned to the processors one by one, counting the amount of data being assigned, until the amount is greater or equal to the minimum required value.
- If the amount of data assigned to the last processor is less than the minimum required amount:
  - Reduce the minimum required amount by one, repeat the assignment process.
- The above is repeated until the last processor gets more data than the minimum required. At this point, work from the last processor to the first, shift the partition boundaries of the processors on the Hilbert curve and pass the partitions to the previous processor in the list until the shift of partition boundaries would cause the processor to contain fewer amount of data than the global average.
- Due to the assignment process, all the empty partitions could be at the end of each processor chunk. While empty partitions does not contain relevant data, they are still looped over during calculations, and of no user defined real space grid point assignments are defined, the processors owning these partitions would also own the real space grid points associated to the space these partitions occupy. Therefore it is important to balance the empty partitions among the processors as much as possible. This is done by going from the first processor to the last, and check if the last partition in the processor chunk is empty and if the processor contains more partitions than global average number of partitions per processor, if so, then move the partition boundaries so that the last partition is passed to the next processor. This repeats until the number of partitions in the processor is less or equal to the global average of if an occupied partition is encountered, and the process moves on to the next processor.

2.2.5 Significance of Rounding Errors

The reader may recall that whether an atom belongs to a partition is determined by equations (1), (2) and (3). This is generally okay if the atoms are not close to the partition boundaries. If an atom is on or very close (within numerical rounding error) to the partition boundaries, then rounding errors becomes significant.

Consider an example: an atom has x-coordinate 10.0 a.u. and the partitions have cell size in x of 2.0 a.u. then in theory $\text{floor}(10.0/2.0) = 5$, hence the atom should be in a partition with $i_x = 5$. However rounding errors can lead to $10.0/2.0 = 4.99999999$. This means $\text{floor}(4.99999999) = 4$. So during calculation, depending on the particular machine and compiled binary, the atom may or may

not be included in partition with $i_x = 5$. There are also the small rounding errors in the atomic coordinates that would cause similar effects (for example, $x = 9.99999999$ instead of 10.0). These rounding errors causes non even distribution of atoms in partitions even for what appears to be perfectly uniform systems. Indeed this problem has been encountered numerous times during test calculations on a perfect crystalline bulk Si structure (see 2.4 below). The problem is also present in the original CONQUEST implementation.

The solution to the rounding error problem is simple. The equations (1), (2) and (3) are rewritten as

$$\begin{aligned} i_x &= \text{floor}\left(r^x_{\text{atom}}/r^x_{\text{part}} + \delta\right) \\ i_y &= \text{floor}\left(r^y_{\text{atom}}/r^y_{\text{part}} + \delta\right) \\ i_z &= \text{floor}\left(r^z_{\text{atom}}/r^z_{\text{part}} + \delta\right) \end{aligned}$$

where $\delta = 10^{-8}$ is a very small number which adds a padding to the possible rounding error one may encounter. The rounding error only affects atoms on partition boundaries, and the addition of δ ensures atoms within δ a.u. to the partition boundaries are included in the correct partition. Tests have shown that this small modification to the code has a significant improvement on automatic load balancing of uniformly distributed systems.

2.3 Implementation in Conquest

New Hilbert curve automatic partitioning and load balancing routines have been created. All the subroutines and functions associated to automatic partitioning algorithms are grouped in two modules:

- `Hilbert3D` module, for all of the routines associated to the generation of non-cubic Hilbert space-filling curves. It is stored in file `Hilbert3D.f90`
- `sfc_partitions_module` module, for all of the routines associated to the automatic partitioning and load balancing tasks. It is stored in file `sfc_partitions_module.f90`.

The subroutine `sfc_partitions_to_processors` is used as a drop-in replacement of the old automatic partitioning subroutine `create_sfc_partitions`.

There are several new input flags added to CONQUEST, these are listed in the tabel below

- `General.NPartitionsX`: Number of partitions in x, if set to 0 the code will determine the number automatically. Default is 0
- `General.NPartitionsY`: Number of partitions in y, if set to 0 the code will determine the number automatically. Default is 0
- `General.NPartitionsA`: Number of partitions in z, if set to 0 the code will determine the number automatically. Default is 0
- `General.AverageAtomicDiameter`: The average atomic diameter used for estimation of gaps in the simulation cell. Default is 5.0 a.u.

The following input parameter has been modified:

- `General.LoadBalance`: A further option =”supportfunctions”= has been added to the option list. This flag controls the weighting function used for distributing partitions to the processors. Default is still =”atoms”=

Note that if the user entered a `General.NPartitionsX|Y|Z` value that is not a power of 2, then the code will use the smallest power of 2 integer greater than the user input value as the number of partions required in that direction.

The `sfc_partitions_module` also contains a checking subroutine and an information subroutine. The checking subroutine checks if data has been correctly assigned to the partitions and processors, such as if there are repeats in atomic indices, if the number of atoms in partitions match the total number of atoms in the simulation cell etc. The checks are run every time when the automatic partitioner is used, so that CONQUEST can be sure that a valid partition–processor map has been created and it can continue to perform actual calculations. The information subroutine depending on the user controlled global verbosity flag, prints out various statistical information on the partition and load assignments to processors, so the user can have an idea whether data had been evenly distributed among the MPI processes.

The user may use the new partitioning scheme without any modifications to his/her `Conquest_input` file used for the old CONQUEST binary, and should encounter minimal behaviour changes other than data being allocated differently (hopefully more optimised) internally to the processors.

2.4 Test Results

To test the new non-cubic Hilbert curve partitioner, the computation of the following systems performed by the updated CONQUEST had been compared with results from the old partitioning scheme:

- Bulk Si with 512 atoms, cell dimensions: (41.04 a.u. $\times$ 41.04 a.u. $\times$ 41.04 a.u.. Reference code: *BulkSi512C*
- Bulk Si with 512 atoms, cell dimensions: (82.09 a.u. $\times$ 82.09 a.u. $\times$ 10.26 a.u.). Reference code: *BulkSi512F*
- Bulk Si with 512 atoms, cell dimensions: (656.72 a.u. $\times$ 10.26 a.u. $\times$ 10.26 a.u.). Reference code: *BulkSi512L*
- Slab Si with 2048 atoms in xy plane, located in the middle of the cell, cell dimensions: (82.09 a.u. $\times$ 82.09 a.u. $\times$ 92.35 a.u.). Reference code: *SlabSi2048M*
- Slab Si with 2048 atoms in xy plane, located at the bottom of the cell, half of the atoms wrapped by periodic boundary condition to the top of the cell, cell dimensions: (82.09 a.u. $\times$ 82.09 a.u. $\times$ 92.35 a.u.). Reference code: *SlabSi2048W*
- Ge hut cluster on Si substrate (slab) with 5333 atoms, slab spans the xy-plane, and located at the bottom of the cell, cell dimensions: (164.18 a.u. $\times$ 205.22 a.u. $\times$ 51.31 a.u.). See figure 3. Reference code: *GeSi5333*
- Ge hut cluster on Si substrate (slab) with 22746 atoms, slab spans the xy-plane, and located at the top of the cell and wrapped around to the bottom of the cell, cell dimensions: (328.36 a.u. $\times$ 328.36 a.u. $\times$ 89.79 a.u.). See figure 4. Reference code: *GeSi22746*

- Ge hut cluster on Si substrate (slab) with 39130 atoms, slab spans the xy-plane, and located at the top of the cell and wrapped around to the bottom of the cell, cell dimensions: (328.36 a.u.$\times$ 328.36 a.u. $\times$ 97.48 a.u.). See figure 5. Reference code: *GeSi22746*
- DNA in water with 3439 atoms. This is used to test whether the load balance weighting by the support functions has any performance improvements over weighting by the atoms. Reference code: *DNAH2O3439*

All test results are based on non-self-consistent single point energy minimisation calculations, with auxiliary matrix tolerance of 10^{-4} Ha. All calculations were done on HECToR phase 3. The new non-cubic Hilbert partitioner is referred to in below as SFC-1.6 (revision 1.6 of the new space-filling-curve implementation); and the original CONQUEST is referred to as r162 (the 162-nd revision in the source code trunk—the most recent released version). Both SCF-1.6 and r162 were compiled with Cray compiler suit `PrgEnv-cray/4.0.46` with `xt-libsci/12.0.00`. The optimisation flag for the compiler was set at `-O3`.

2.4.1 Performance in Relation to Cell Shapes

The test results for the small (512 atoms) bulk Si systems with different shaped cells are presented below. All partition and load balancing are done automatically, and are weighted against the number of atoms. Weighting against support functions produces no benefit in this case as there is only one species present in the simulation.

		BulkSi512C	BulkSi512F	BulkSi512L
r162	Cores	32 (1×32)	32 (1×32)	32 (1×32)
	Partitions	$4 \times 4 \times 4$	$4 \times 4 \times 4$	–
	Total parts	64	64	–
	Occ parts	64	48	–
	Max atoms/proc	23	30	–
	Min atoms/proc	11	10	–
	Mean atoms/proc	16.0	16.0	–
	Std. atoms/proc	3.799671	4.138236	–
	N McW iterations	14	14	–
	N minE iterations	8	8	–
	Wall time (s)	562.023	728.443	–
SFC-1.6	Cores	32 (1×32)	32 (1×32)	32 (1×32)
	Partitions	$4 \times 4 \times 4$	$8 \times 8 \times 1$	$64 \times 1 \times 1$
	Total parts	64	64	64
	Occ parts	64	64	64
	Max atoms/proc	16	16	16
	Min atoms/proc	16	16	16
	Mean atoms/proc	16.0	16.0	16.0
	Std atoms/proc	0.0	0.0	0.0
	N McW iterations	14	14	14
	N minE iterations	8	8	8
	Wall time (s)	389.424	379.931	376.696

The r162 version failed to assign atoms to all processors for BulkSi512L, and stopped before any partition data can be printed. The extreme difference in the cell dimensions had broken the load balancing algorithm using the standard cubic Hilbert curves, as far too many partitions are being created in the transverse directions. Significant slow down can be observed going from the cubic cell BulkSi512C to the flat cell BulkSi512F, as empty partitions are being created due inflexibility of the cubic 3D Hilbert curve. The new partitioning scheme however produced optimal load balancing in all three cell shapes. The correction to the rounding error mentioned in section 2.2.5 is also evident when comparing the results for BulkSi512C, where both partitioning schemes used the same number of partitions and cubic Hilbert curve. The new scheme correctly corrected the placement of atoms in partitions, resulting significant improvements in performance.

For the more difficult Si slabs, the results are given below:

		SlabSi2048M	SlabSi2048W
r162	Cores	128 (4×32)	128 (4×32)
	Partitions	$8 \times 8 \times 8$	$8 \times 8 \times 8$
	Total parts	512	512
	Occ parts	320	256
	Max parts/proc	66	42
	Min parts/proc	1	1
	Mean parts/proc	4.0	4.0
	Std. parts/proc	7.280110	6.544320
	Max atoms/proc	26	34
	Min atoms/proc	9	8
	Mean atoms/proc	16.0	16.0
	Std. atoms/proc	3.818131	3.897114
	N McW iterations	–	19
	N minE iterations	–	31
	Wall time (s)	–	2278.342
SFC-1.6	Cores	128 (4×32)	128 (4×32)
	Partitions	$16 \times 16 \times 1$	$16 \times 16 \times 1$
	Total parts	256	256
	Occ parts	256	256
	Max parts/proc	2	2
	Min parts/proc	2	2
	Mean parts/proc	2.0	2.0
	Std. parts/proc	0.0	0.0
	Max atoms/proc	16	16
	Min atoms/proc	16	16
	Mean atoms/proc	16.0	16.0
	Std atoms/proc	0.0	0.0
	N McW iterations	19	19
	N minE iterations	34	34
	Wall time (s)	1131.634	1137.756

The new partitioner correctly determined the systems to be slabs, and constructed the partitions accordingly, resulting in significantly more optimised load balancing compared to r162. For

SlabSi2048M, r612 crashed due to OOM error on HECToR. Due to the fact that the phase 3 HECToR has only 1GB of memory per core, it is easy to go over the limits if the load balance is not sufficiently good. In this case, the problem did not come from too many atoms being allocated to one processor, but came from the fact that too many empty partitions were produced and some processors got too many empty partitions. This fact can be observed from the large disparity between the maximum and minimum number of partitions per processor (66 vs. 1). The result is too many integration block points are being assigned to a particular processor, resulting in OOM error.

2.4.2 Ge Hut Clusters

For the Ge hut clusters on Si substrates, r162 with automatic partitioner failed to provide adequate load balancing such the corresponding calculations for all 3 system sizes failed due to OOM error on Conquest. The number of processors to atoms ratio was already set to be quite high, averaging about 20 atoms per processor. Requesting more processors caused the r162 partitioner to fail due to it being unable to assign atoms to all processors. The old automatic partitioner has been demonstrated to work on an IBM p690 system (see [3]), however HECToR phase 3 has a very different architecture, and there are far fewer amount of memory available per core on HECToR. The new SFC-1.6 version however ran successfully for all systems due to improved partitioning and load balancing.

Never-the-less CONQUEST versions r162 and below had been demonstrated to make efficient calculations on large Ge hut clusters. In all those cases user defined partition files generated from external utilities were provided in place of the automatic Hilbert partitioner. It would be interesting to compare the performance CONQUEST with the new automatic partitioner against the manually produced partition files. The Manual partitions produced for this test were just simple divisions of the simulation cell based on visual inspections.

The results are given below: r162M denotes r162 with manual partitioning.

		GeSi5333	GeSi22746	GeSi39130
r162M	Cores	256 (8×32)	1024 (32×32)	1792 (56×32)
	Partitions	$16 \times 16 \times 1$	$32 \times 32 \times 1$	$56 \times 32 \times 1$
	Total parts	256	1024	1792
	Occ parts	256	1024	1792
	N McW iterations	15	14	13
	N minE iterations	12	8	8
	Wall time (s)	224.020	278.708	350.602
SFC-1.6	Cores	256 (8×32)	1024 (32×32)	1792 (56×32)
	Partitions	$16 \times 32 \times 1$	$64 \times 64 \times 1$	$64 \times 64 \times 1$
	Total parts	512	4096	4096
	Occ parts	512	4096	4096
	Max atoms/proc	32	32	38
	Min atoms/proc	16	19	17
	Mean atoms/proc	20.832031	22.212891	21.835938
	Std atoms/proc	3.802640	2.508888	3.869651
	N McW iterations	15	14	13
	N minE iterations	12	8	8
	Wall time (s)	219.705	251.927	328.123

As the results show the new automatic partitioner also slightly out performs the manually set partitions (albeit just simple divisions of the cell, and non-optimised). This can be attributed to the use of Hilbert curve for indexing the partitions during load assignment in SFC-1.6, which is not the case for the manually set. The compactness of the Hilbert curve helps with reducing the amount of MPI communications between the processors.

2.4.3 DNA In Water

DNA in water has 6 pieces including Hydrogen, and is a typical biological system where load balancing with respect to the number of atoms in processors may not produce optimal performances. Tests were carried out for SFC-1.6 running with `General.LoadBalance` set to `atoms` and `supportfunctions`. The results are given below:

	`atoms`	`supportfunctions`
Cores	256 (8×32)	256 (8×32)
Partitions	$8 \times 8 \times 8$	$8 \times 8 \times 8$
Total parts	512	512
Occ parts	512	512
Max atoms/proc	22	23
Min atoms/proc	9	6
Mean atoms/proc	13.433594	13.433594
Std atoms/proc	2.866067	2.668443
Max sf/proc	63	49
Min sf/proc	15	20
Mean sf/proc	29.089844	29.089844
Std sf/proc	8.394784	6.045057
N McW iterations	26	26
N minE iterations	4	4
Wall time (s)	492.558	445.334

The use support functions to weight the assignment of partitions to processors thus produced 9.59% of performance improvement.

3 Molecular Dynamics with Dynamic Reassignment

In the original Conquest implementation, the once the atoms are assigned to the partitions, they are assumed to be in the partition through out the course of simulation. While each atom has a global index in the simulation cell, the actual book keeping of which processor owns which atom, and where should the data related to a remote neighbour atom be fetched are all done based on the index system based on the partitions and halos, which are collections of partitions which contains at least an atom within a given interaction range of an atom in the primary set. The primary set of a MPI process is the set of all atoms owned by the process. If during molecular dynamics (MD), some of the atoms get out of their partitions into a different partition, or cross the simulation cell boundaries, the book keeping on them are never updated. Therefore, while the new atomic positions

tells the code they belong to the new partitions[2], the new partitions have no record of them, and the old partition still treats the atoms as if they have never left. This produces errors in neighbour lists, and contradictions in sizes of sending and receiving arrays for values on the integration grids. Typically a MD calculation may need to be restarted (rerun of the code reading from updated atomic coordinates) every few time-steps, whenever an atom moves out of its own partition. This makes CONQUEST in its original version impractical for performing large scale MD simulations.

Further more, CONQUEST at the beginning of every MD step starts afresh with McWeeny initialisation for the auxiliary matrix **L**. The auxiliary matrix is the variable used in energy minimisation process. If the atoms does not move too far in every MD step, then it may be more efficient to reuse the auxiliary **L** matrix at the start of the next MD step, skipping the McWeeny **L** initialisation process all together.

The work for improving molecular dynamics in CONQUEST can therefore be done in two parts. The first is to to add the ability for CONQUEST to update its partition and bundle[3] data dynamically during a MD run, so that the code does not need to be restarted every time an atom exists or enters a partition. The second is to implement ways of transfer of matrix rows corresponding to the moved atoms from their old owners to new owners, the reconstruct **L** matrix calculated from the previous MD step, which can be used directly in the energy minimisation loops in the new step.

The work was done in collaboration with Michiaki Arita and Tsuyoshi Miyazaki from National Institute for Materials Science (NIMS), Japan.

3.1 Over view of MD implementation

Figure 6 shows a simplified picture of the Born-Oppenheimer molecular dynamics algorithm implemented in CONQUEST. The red path and box denotes the original implementation, in this case only the atomic neighbour list and the list of neighbouring partitions are up dated. The covering set is just a collection of all partitions containing a neighbour of an atom in the primary set. The number of atoms (members) in the partitions are not changed. In the updated approach (black path), the members information in the partitions are updated according to the new atomic coordinates, and the matrix rows associated to the moved atoms are then transfered to the corresponding partition/processors.

For the current implementation, the number and size of partitions are assumed to be fixed through out an MD simulation session. One can have different number of partitions if restarting a CONQUEST MD run based on the previous MD data.

After the atoms have been moved and velocities calculated, the following is carried out (corresponding to the blue boxe in figure 6):

1. Using the new atomic coordinates, reassign atoms to partitions, and update the halos and covering set, and reconstruct the neighbour lists.

2. Use a (new) flag to tell if a primary set atom has data stored locally. If not then:

   - Issue `MPI_send` and `MPI_irecv` to transfer the relevant **L** matrix rows and the basis set coefficients from the old owners to the new owners

[2] In CONQUEST, all periodic images of atoms in the simulation cell are treated as if they are real atoms in a much larger super cell. The data associated to these images are taken from their respective images in the simulation cell, and thus belongs to the corresponding partition and processor which own the atom in the simulation cell.

[3] A collection of all partitions owned by a MPI process is referred to as a bundle in CONQUEST.

3. Reorder the local rows (not including the remote rows being fetched in step 2.) of **L** according to the new indices calculated from step 1. This can be done in parallel to the MPI communications initiated in step 2.

4. After the MPI communications have finished, the reindexed **L** matrix part on the processor is reconstructed from the reordered local rows and received received rows.

Once the **L** matrix has been reconstructed with up to date neighbour lists, it can then be fed into the DFT routines for the next round of MD loop.

3.2 Update Members Information

After the calculation of DFT ground states and movement of atoms during MD step n, each processor knows:

- The updated coordinates of atoms in its old primary set
- The updated covering set (neighbour lists) corresponding to the old primary set
- The displacement of atoms in the primary set (from step $n-1$ to n)

There are situation where an atom has wondered out of the simulation cell, and therefore, becomes an "periodic image". It must be folded back into the simulation cell according to periodic boundary conditions, and be assigned to the new partition accordingly.

To update the member information, each processor must do the following:

- Loop over all atoms in the simulation cell, recalculate which partition does each atom belong to, any atom that have existed the simulation cell are assigned partitions corresponding to its periodic image in the cell
- Reconstruct the partitions, and update the local atom sequence mappings for all processors. The atom sequences are arranged in the order of bundles (set of partitions owned by a processor)–partitions–atoms
- From the new processor-partition-atom maps, reconstruct the global atomic index to bundle atomic index mappings. At the same time, remember the old mappings, as it will be used for rearranging various data arrays whose elements are still ordered according to the old maps
- Rearrange the atomic coordinates, velocity and species arrays according to the new atomic bundle indices.
- Update the new primary set
- Update the neighbour list and the covering set

After the member information is updated, the code can either continue onto a new molecular dynamics step, with **L** rebuilt from scratch, and pass through the McWeeny initialisation and early energy minimisation (preparation) steps; or more perhaps more efficiently reconstruct the **L** matrix from the current MD step using the updated member information, and use it as the initial input for the energy minimisation steps in the next MD step. This way, both McWeeny iteration and early energy minimisation preparations may be skipped, thus improving the running time of the MD simulation. The section below describes how **L** may be rebuilt.

3.3 Reconstruct L Matrix

After the member information have been updated, in theory all of the required **L** matrix data has already been computed by the previous MD step. The **L** matrix is piece-wisely stored on all participating processors, with format dependent on the primary and covering set information. As the primary and covering set has changed, the way **L** matrix is stored must also change to correspond to the new member informations for the code to run correctly. This section gives a brief explanation on how is done.

The **L** matrix is stored in rows on the processors. The row are indexed by atoms and their corresponding support functions. Only the matrix elements with column index being the neighbours of i are stored in each row.

Each processor must first work out which of its own rows (atoms) now belongs to the primary sets of other processors, and which atom now in its primary set is was a member of the primary set on another processor. The processors not only has to send the relevant rows of **L**, it also has to send the information on the global atomic index of the row (i.e. which exact row in the global **L**) it is sending, how many rows, and global atomic indices of neighbours j in each row. Otherwise unlike in the original CONQUEST implementation where the member data never changes, the processors would not be able to find out the identity of the remote matrix data it received because the member data has changed while the data received was still arranged using the old format. Therefore in order to reorder for each processor to the matrix rows correctly, the exact format information about the **L** matrix needs to be sent and received before the matrix rows are being sent, which records:

- The number of atoms in the old primary set corresponding to the **L** matrix before its reordering
- The α indices (support function indices) corresponding to each atom i, indexing the matrix element $L_{i\alpha j\beta}$
- The number of (old) neighbours j of each atom i, which corresponds to the length of each row in **L**
- The global ids of all the neighbours j of atom i in the simulation
- The length of each $L_{i\alpha j\beta}$ row (the number of combined index $j\beta$)
- The beginning location in the **L** matrix array for each row, indexed by i
- The support function β indices of each of neighbours j of atom i
- The displacements between atoms i and the neighbours j of the **L** matrix not yet been re-ordered, and with the atoms indexed in the covering set-partition-atom indices of the previous MD step. This information is important for reordering the matrix elements for the current MD step.

The above is grouped into a structure type `InfoL`, and will be sent to and received by the relevant processor before the matrix data is sent. After this the relevant matrix rows are sent, using `MPI_irecv` and `MPI_send` pair. The part of **L** is first reordered using the updated member information, and once the relevant data from the remote matrix row has been received they are added to the local matrix to complete the reconstruction of **L**.

To reorder the local **L** matrix, the goal is to identify the row atoms and their neighbours stored in the old matrix format and correspond them to the atoms in the new covering set. The following steps are taken:

- Loop over the old primary atoms i_{old}, rows of local **L** (number from `InfoL`)
  - Loop over the current primary atoms i_{new} and match via global id which i_{new} corresponds to i_{old} and get the the new bundle index for this atom: i_{new}
  - Get the atomic displacement of i_{new} from the previous step, which are stored during velocity-verlet routine when moving atoms.
  - Loop over the old neighbours j_{old} (number from `InfoL`), at the moment these atoms are unidentified, the goal of this loop is to find their id in the updated neighbour list.
    - Get the atomic displacement of j_{old} from data stored in velocity-verlet routine
    - Get the current position of j_new, calculated using the current position of i_{new}, the displacement of i_{old} during MD and the old atomic displacements between i_{old} and j_{old} stored in `InfoL`
    - From the position of j_{new} find the partition in the (current) covering set this atom belongs to
    - Loop over the atoms in that partition, and use the global id of j_{old} the current bundle labeling for j_{new}
    - Reorder the **L** matrix elements

The remote matrix elements are constructed into the new formatted **L** matrix in the same way, by identifying the atoms associated to the row and column indices of the received matrix rows in the new covering set, and then assign the matrix rows to their appropriate places accordingly

3.4 Test Results

The tests were carried out to find out the stability of the implemented MD routines when running on HECToR. The code was again compiled with Cray compiler suit `PrgEnv-cray/4.0.46` with `xt-libsci/12.0.00`. The optimisation flag for the compiler was set at `-O3`.

The test systems are water (ice) boxes of various sizes. The molecular dynamics settings were always set with:

- Initial ionic temperature: 300 K
- Time step-size: 0.5 fs
- DFT Functional: PW92 (LDA)
- Self-consistency: Mixed energy minimisation and charge self-consistency scheme

3.4.1 Stability

First the stability of the MD algorithm was tested. CONQUEST was run on 32 cores (1 node) on HECToR phase 3, for MD simulation of a 768 atoms water box. The first run without reusing the **L** matrix, starting from McWeeny initialisations at every time step. This is the way the original CONQUEST implements MD, with the only difference being the member data is now updated

after every MD step, so the code would not stop if an atom crosses a partition boundary. The results were then compared with the MD calculation run that does reuse the **L** matrix, and skips the entire McWeeny initialisation steps and early energy minimisation preparations. The **L** matrix tolerance, the criteria for finishing the energy minimisation procedure was set to 10^{-3}. Originally the simulation time was set to be 800 iterations (400 fs). However, due to the slow speed of the calculation with using McWeeny iterations for every time step, only 136 MD steps were completed in 12 hours. 32 cores were already quite large number of cores for a 768 atoms system, and increasing the number of cores beyond 64 atoms would result in poor load balancing as well as large increases in communication-to-computation ratio. In any case the speed up of calculation required for completing 400+ step computations will need to be 4 to 5 times, this cannot be achieved by increasing the number of cores before the atoms in the simulation cell run out. Therefore, a long test run on HECToR for the MD calculation with McWeeny initialisation at every time step was considered to be impractical. Never-the-less long MD runs for a small 8 atom Si cell with the same CONQUEST settings were performed on the NIMS Simulator 1 (Intel Xeon processor Nehalem-EP (2.8 GHz), 4 cores/ node, 2.85GB per core) in Japan, the results of which are also presented in this section below.

Figure 7 shows the total energy vs. simulation time results of the MD runs on the 768 atoms water box carried out using McWeeny initialisation and that reusing the **L** matrix at each step. The simulation was carried out for 100 steps. Significant oscillations in total energy was observed for the both simulation. For both simulations the amplitude of the oscillation decreased over simulation time, with the amplitude dropping faster for the calculation with reused **L**. The mean energy of the "McWeeny" calculation stayed constant, while the "reuseL" result showed a clear drift, which gradually turned constant at a lower energy.

The oscillations observed for the "McWeeny" calculations seemed to be dependent on the type of systems. The results shown in figure 8 corresponds to that of the test calculations performed on Si 8 atoms cell by collaborators from NIMS Japan using exactly the same code demonstrated that the energy of MD simulation with McWeeny initialisation being used at every step remained largely constant, with only very small amount oscillations about the mean value. However the energy drift in "reuseL" results follow the same trend as the corresponding results obtained for the water box on HECToR.

3.4.2 Increasing L Tolerances

Figure 9 shows the energy vs.\ simulation time results obtained by increasing the **L** tolerances for the "reuseL" calculations. The "reuseL" calculations were all simulated for 400 fs (800 MD steps). Large oscillations where observed in all calculations, which gradually went away. Apart of the initial oscillations the result for the 768 atoms water box followed the same trend as that was found in the 8 atom Si cell. As the **L** tolerances becomes stricter, the energy drift in the "reuseL" calculations became smaller.

The nature of the large oscillations found at the initial stages of MD simulation for the water box is not well understood. The oscillations should not have been physical, and they suggest possible poor initial conditions present in the simulation. Significant time has been spent in trying to find a possible bug in the implementation, however no such bugs were found so far. The damping of the oscillations as the simulation goes on showed that despite poor initial conditions the MD algorithm is largely stable and the calculation did not diverge.

The energy drifts found in the "reuseL" calculations may be related to the accumulation of errors

during successive MD steps. The resetting of **L** matrix for the

While this dCSE project has completed, the investigation into the problems in MD simulations is on going by the collaborative development team of CONQUEST.

3.4.3 Computational Costs

The tables below shows the computational cost used by the MD simulations performed on HECToR. The system is water (ice) boxes of different number of atoms,

Method	**L** Tol.	N Atoms	Cores	MD Steps	N McWeeny Iter.	N minE/SC Iter.	Wall time (s)
McWeeny	10^{-3}	768	32	100	1776	307	4047.732
ReuseL	10^{-3}	768	32	100	18	130	2451.367
ReuseL	10^{-4}	768	32	100	18	366	4246.005
ReuseL	10^{-5}	768	32	100	18	656	6211.387
ReuseL	10^{-6}	768	32	100	18	1030	8814.671
McWeeny	10^{-3}	1536	64	100	1800	307	5826.146
ReuseL	10^{-3}	1536	64	100	18	130	3531.344

If the "ReuseL" method was set to use the same **L** tolerance as the "McWeeny" calculations, then for the water box system tested the "ReuseL" calculation is about 40% faster than the "McWeeny" calculation, mainly due to the significantly reduced number of McWeeny iterations required in total. For loose tolerances, the mixed self-consistent energy minimisation step converge within 1 or 2 steps. If "ReuseL" calculations are done using stricter tolerances to tackle the energy drifting problem, then the number of iterations in the mixed self-consistent energy minimisation step to increase, and this are reflected it the amount of wall time those calculation used. Therefore, while stricter tolerances may reduce the amount of drift in the total energy, they become less efficient for a given MD time step. Also worth noting from the above table is that each McWeeny iteration takes considerable less computational cost compared with a mixed self consistent energy minimisation iteration.

3.4.4 Implemented Subroutine and User Input Flags

The implemented subroutines are arranged in 4 different modules. These are listed below:

- `atom_dispenser_module`: mapping atoms to partitions
  - `atom2part`
  - `allatom2part`
- `UpdateMember_module`: member updates
  - `group_update_mparts`
  - `deallocate_PSmember`
  - `allocate_PSmember`
  - `allocate_Psmember`
  - `primary_update_mparts`

  - `deallocate_CSmember`
  - `allocate_Csmamber`
  - `cover_update_mparts`
  - `updateMembers`

- `UpdateInfo_module`: rebuild **L** matrices
  - `Lmatrix_CommRebuild`
  - `make_glob_to_node`
  - `sort_recv_node`
  - `alloc_send_array`
  - `CommLmat_send_size`
  - `CommLmat_send_neig`
  - `CommLmat_send_data`
  - `alloc_recv_arra`
  - `CommLmat_irecv_data`
  - `UpdateLmatrix_loca`
  - `UpdateLmatrix_remot`
  - `deallocate_CommLmatArrays`
- `io_module2`: for input and output
  - `dump_matrixL`
  - `grab_matrixL`
  - `dump_InfoGlobal`
  - `grab_InfoGlobal`
  - `dump_idglob_old`
  - `grab_idglob_old`
  - `deallocate_InfoLmatrix_File`

The following new input flags is added

Input Flag	Purpose	Default
`General.UseOLDConquest`	Whether to turn on the new MD implementation	F
`AtomMove.ReuseL`	Whether to reuse **L** matrix from previous step	F
`AtomMove.McWeenyFreq`	Number of MD iteration before resetting **L**	1
`AtomMove.SkipEarlyDM`	Whether to skip EarlyDM preparation	F

3.5 Extended Lagrangian Born-Oppenheimer MD

As the test have show that despite a dramatic increase in speed when reusing **L** matrix for lower tolerances, the stability of the calculations required a stricter **L** matrix tolerance, which reduced the effectiveness in performance improvements when reusing **L**.

One possible solution was found by extending the existing CONQUEST Bon-Oppenheimer MD to use the extended Lagrangian formalism[6], which incorporates additional electronic degrees of freedom into the Born-Oppenheimer Lagrangian. **L** matrix is reused at every MD step, and no McWeeny initialisation iterations are required for the extended Lagrangian Born-Oppenheimer MD.

Result of a preliminary test calculation using CONQUEST with the extended Lagrangian formalism performed on an 8 atom Si cell is shown in figure 10. The results show that extended Lagrangian formalism produced virtually no energy drift when using a loose tolerance of 10^{-3}.

The computational costs of the extend Lagrangian Born-Oppenheimer MD for the 8 atoms Si cell is compared with the "McWeeny" and "ReuseL" methods in the table below:

Method	**L** Tol.	MD Steps	N McWeeny Iter.	N minE/SC Iter.
Ex. Lag.	$1o^{-3}$	2000	17	4766
McWeeny	10^{-3}	2000	34000	$>$ 20000
ReuseL	10^{-3}	2000	17	$>$2000
ReuseL	10^{-4}	2000	17	$\approx$ 4000
ReuseL	10^{-5}	2000	17	7476
ReuseL	10^{-6}	2000	17	16677
ReuseL	10^{-7}	2000	17	28298

The results indicate that while the Extended Lagrangian formalism is more complicated (hence more expensive per MD step than "ReuseL" method) and takes more iterations to for the energy to converge than "ReuseL" method using the same **L** tolerance, the extended Lagrangian method offered much more superior performance in stability of the MD simulation. The new method is still significantly faster than the method involving resetting **L** with McWeeny initialisation at every MD step.

The research in this topic is on-going, and it is beyond the scope of this dCSE project.

4 Miscellaneous Remarks

During running tests on the Hilbert automatic partitioner implementation on HECToR, errors were encountered which was traced to be caused by the optimisation settings during compilation of CONQUEST using Cray compiler (`PrgEnv-cray/4.0.46`). Setting optimiser flags to either `-O2` and `-O3` caused error in the atomic indices, while turning off optimisation made the error disappear. Further analyst showed this may have been caused by the optimiser trying to merge or swap several lines of code—inserting a `print` statement between the incident lines fixed the bug (with the optimiser set to `-O3`).

The code was properly fixed eventually by rewriting the part of code that was causing the problem in a different way (while doing the same logical functions).

5 Summary and Conclusion

The goal of this dCSE project is to:

1. Implement a more flexible Hilbert partitioning algorithm in CONQUEST, in order to improve the user friendliness of the code in general and load balancing when running on HECToR.

2. Change the existing molecular dynamics code in CONQUEST to include ability to dynamically reassign atoms to partitions, thus allowing the code to perform molecular dynamics simulations without needing to restart in frequent intervals. Furthermore, allow the code to use the calculated **L** matrix from the previous MD step in order to speed up the simulation.

Objective 1 has been successfully achieved. A solution for generating flexible non-cubic Hilbert curves has been found, and the associated automatic partitioning algorithms have been implemented in CONQUEST. The new automatic partitioning algorithm shows significant improvement in both load balancing performances and functionality over the original implementation. It allowed users to manually set partitions in any given direction at will. Tests have shown that due to the use of Hilbert curves the partitioning algorithm out-performed the external utility that was traditionally used by users to manually set partitions and distribute data to processors, and thus can potentially become a replacement over the existing external utility. The partitioner in full-auto mode also performed well compared with the original implementation. In particular the new automatic partitioning scheme allowed calculations to be performed efficiently for awkward systems on HECToR with minimal user input, where the original automatic partitioner would fail.

Objective 2 has been achieved with partial success. All of the proposed implementations related to MD has been successfully implemented. Most importantly CONQUEST can now perform stable MD simulations for indefinite number of steps without the need to restart the job. In terms of performance improvements, however, while reusing **L** reduces the computation cost from 40%(water)-90%(Si) compared with initialising the matrix at every time step, significant energy drift where observed. Tightening up computational tolerances during MD simulations helped to reduce the energy drift, but at a great cost to overall computational time. Large initial oscillations in total energy was also observed during doing test calculations on water boxes. The investigation on this problem is still on-going by the CONQUEST development team. Due to the energy drift and oscillation issues with the current MD implementation, the MD simulation of large GramicidinA embedded lipid bilayer originally proposed for the project was not performed.

The recent works indicated that solution to the problems discovered in MD may be solved by employing the extended Lagrangian Born-Oppenheimer formalism. While the investigation on this topic is still on-going, preliminary studies have shown positive results.

The work on the new Hilbert partitioner has been submitted into CONQUEST trunk, and is available to all users of CONQUEST.

6 Acknowledgement

The Author wish to thank Michiaki Arita and Tsuyoshi Miyazaki for their contribution to the work on implementing the improved molecular dynamics algorithms in CONQUEST.

This project was funded under the HECToR Distributed Computational Science and Engineering (CSE) Service operated by NAG Ltd. HECToR—A Research Councils UK High End Computing

Service—is the UK's national supercomputing service, managed by EPSRC on behalf of the participating Research Councils. Its mission is to support capability science and engineering in UK academia. The HECToR supercomputers are managed by UoE HPCx Ltd and the CSE Support Service is provided by NAG Ltd. http://www.hector.ac.uk

References

[1] D. R. Bowler, T. Miyazaki, and M. J. Gillan. Parallel sparse matrix multiplication for linear scaling electronic structure calculations. *Comput. Phys. Commun.*, 137(2):255–273, June 2001.

[2] D. R. Bowler, T. Miyazaki, and M. J. Gillan. Recent progress in linear scaling ab initio electronic structure techniques. *J. Phys.: Condens. Matter*, 14(11):2781, 2002.

[3] V. Brázdová and D. R. Bowler. Automatic data distribution and load balancing with space-filling curves: implementation in CONQUEST. *J. Phys.: Condens. Matter*, 20(27):275223, 2008.

[4] F. Gray. Pulse code communication, 1953.

[5] W. Kohn and L. J. Sham. Self-consistent equations including exchange and correlation effects. *Phys. Rev.*, 140(4A):A1133, 1965.

[6] A. M. N. Niklasson. Extended born-oppenheimer molecular dynamics. *Phys. Rev. Lett.*, 100(12):123004–, Mar. 2008.

[7] H. Sagan. *Space Filling Curves*. Springer, 1994.

[8] J. Skilling. In G. Erickson and Y. Zhai, editors, *Bayesian Inference and Maximum Entropy Methods in Science and Engineering: 23rd Int. Workshop*, number CP707, page 381–7, New York, 2004. American Institute of Physics.

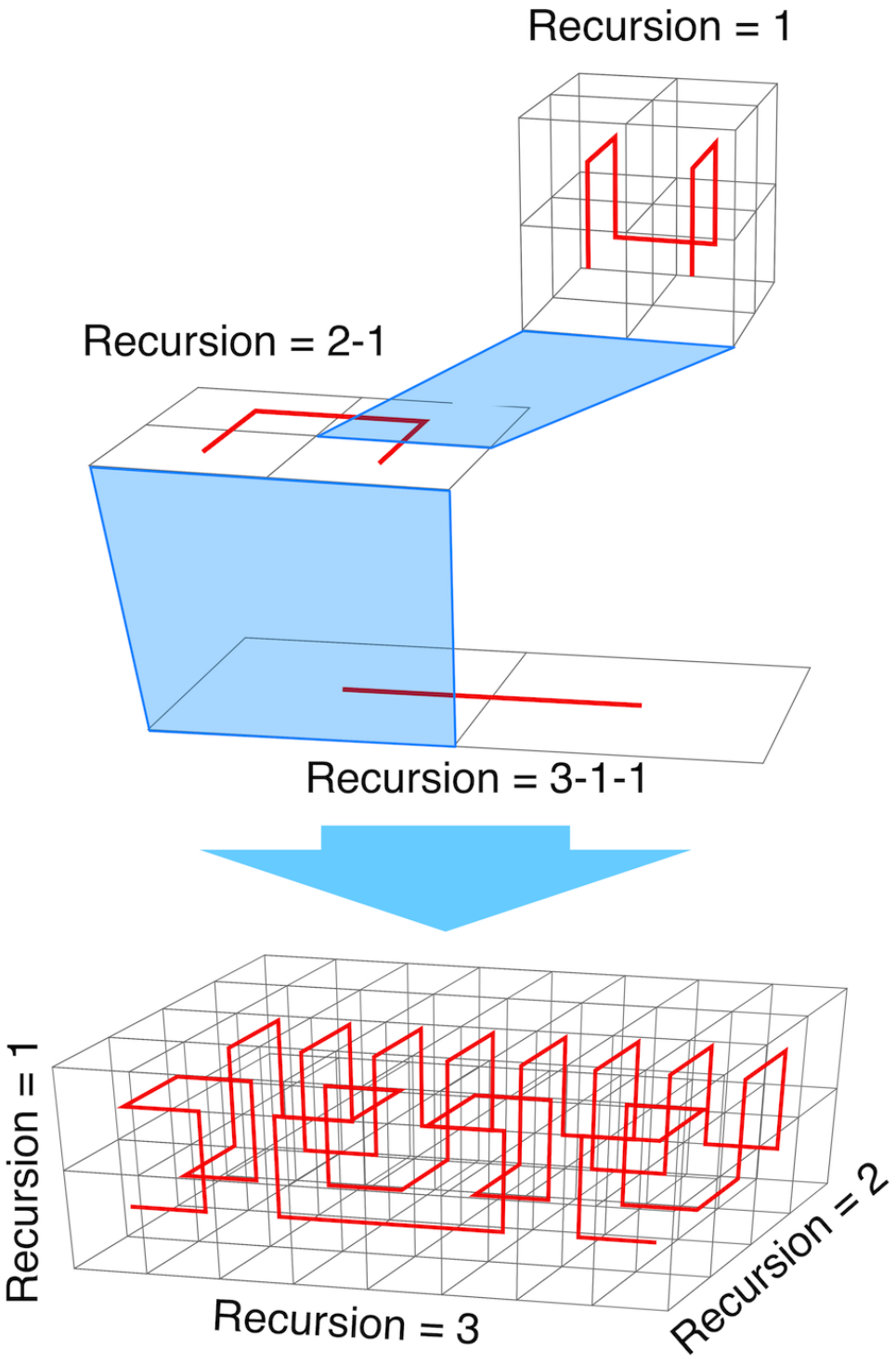

Figure 1: Construction of Hilbert curve with $N_x = 3$, $N_y = 2$, $N_z = 1$

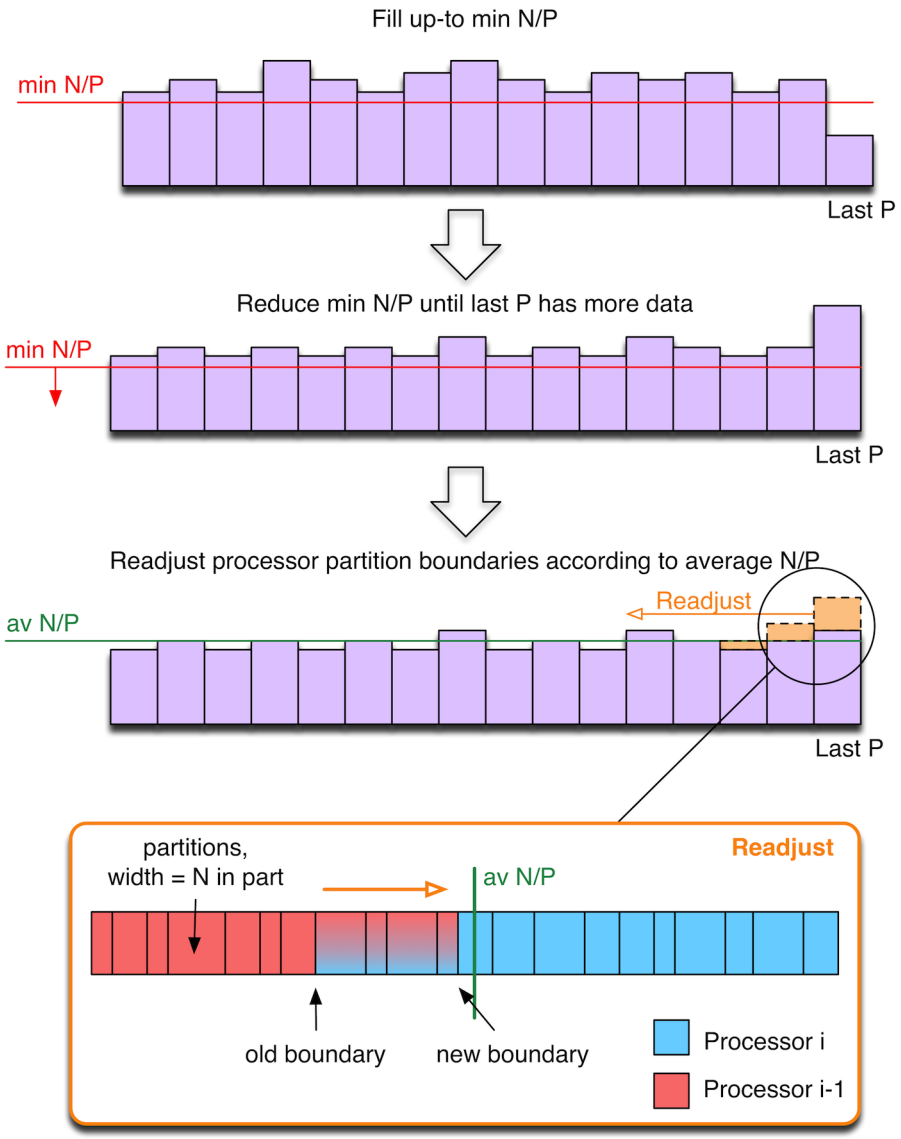

Figure 2: Assignment of partitions to processors

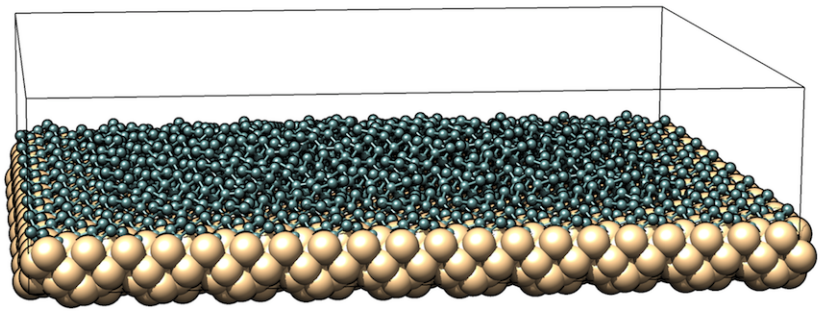

Figure 3: Ge Hut Cluster on Si Substrate, 5333 atoms

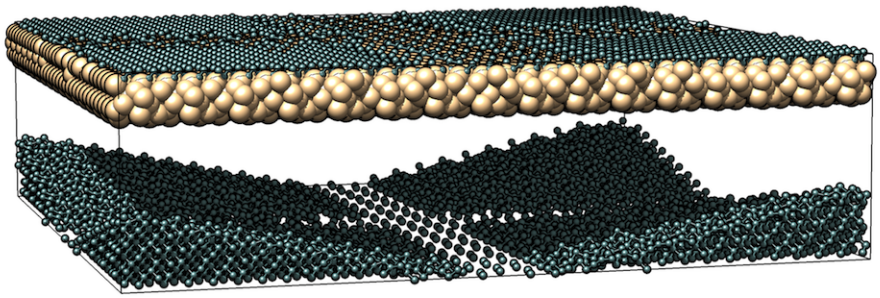

Figure 4: Ge Hut Cluster on Si Substrate, 22746 atoms

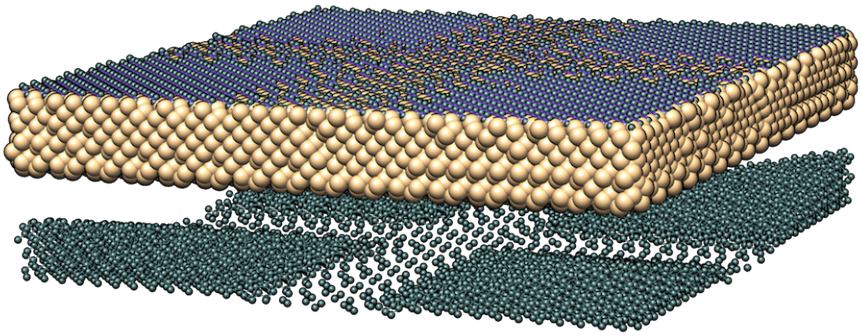

Figure 5: Ge Hut Cluster on Si Substrate, 39130 atoms

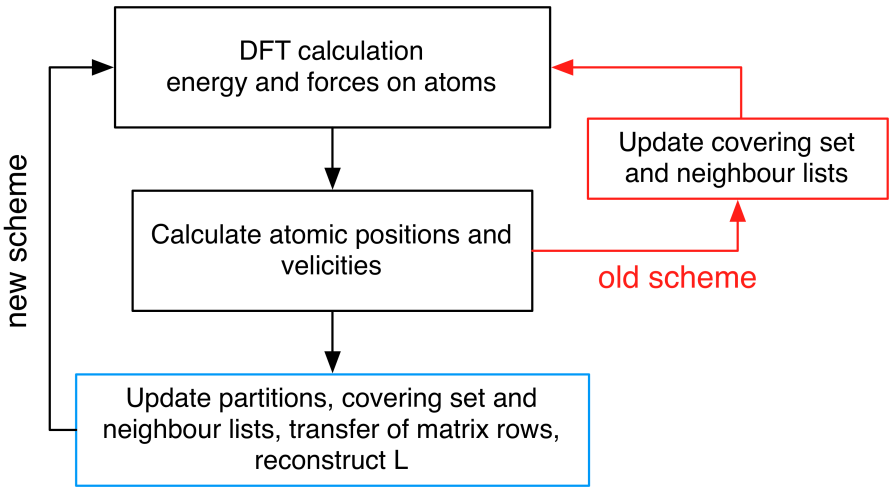

Figure 6: Over view of MD scheme implemented in Conquest

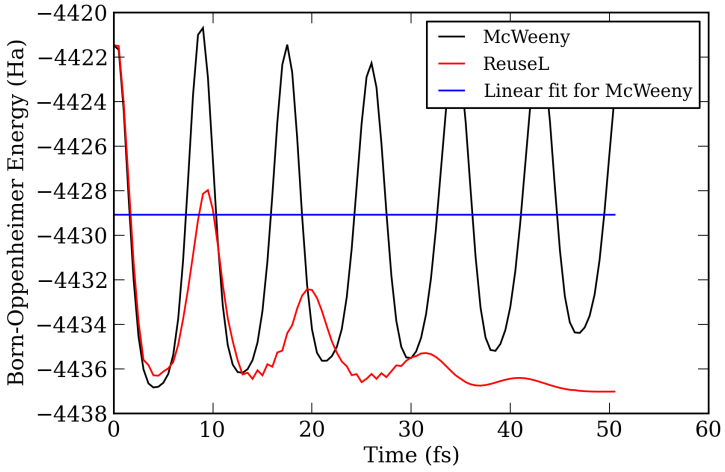

Figure 7: Total Born Oppenheimer energy vs the simulation time. The black curve labelled "McWeeny" corresponds to the results obtained from MD runs which reconstructs **L** matrix at every step with McWeeny initialisation; the red curve labelled "reuseL" corresponds to the results obtained from MD runs which reuses the **L** matrix at each step

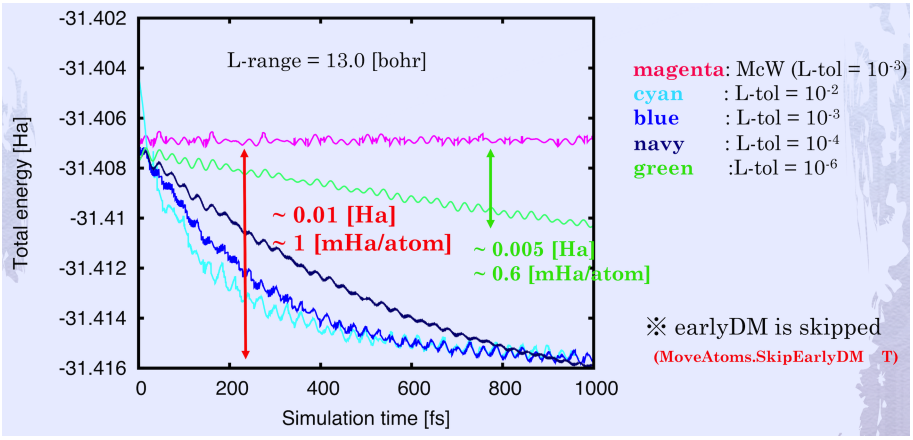

Figure 8: MD simulation results for 8 atoms bulk Si cell. The time-step was 0.5 fs, with the simulation running for total of 2000 MD steps. The calculation was performed by Michiaki Arita on NIMS simulater 1. The magenta line corresponds to the calculation with **L** reconstructed from scratch using McWeeny initialisation; the cyan, blue, navy and green lines corresponds to the MD simulation with **L** being reused, and with **L** tolerances set to 1e-2, 1e-3, 1e-4 and 1e-6 respectively

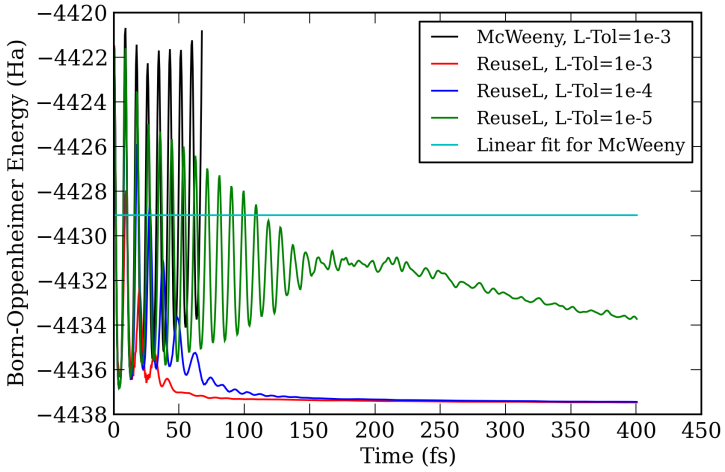

Figure 9: Total Born Oppenheimer energy vs the simulation time for various **L** tolerances. The black curve labelled "McWeeny" corresponds to the results obtained from MD runs which reconstructs **L** matrix at every step with McWeeny initialisation; The rest of the curves corresponds to the results obtained from MD runs which reuses the **L** matrix at every step, with different **L** tolerances

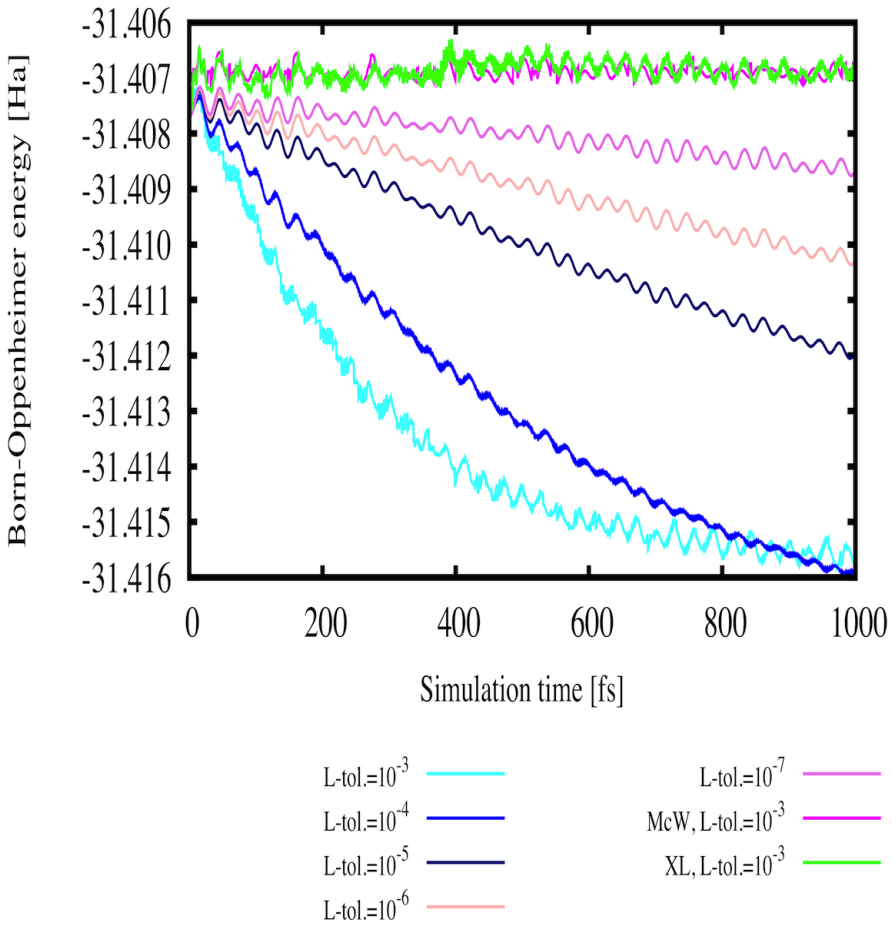

Figure 10: Energy vs. simulation time for MD simulation of 8 atoms Si cell. The calculations were performed by Michiaki Arita using NIMS simulator 1 in Japan. The magenta curve corresponds to the results obtained using the "McWeeny" approach with **L** tolerance set to 1e-3; the green curve corresponds to the results obtained using extended Lagrangian MD method , with **L** tolerance set to 1e-3; the rest of the curves correspond to the results obtained using "reuseL" method, with various **L** tolerances