Automatic data distribution and load balancing with space-filling curves: implementation in CONQUEST

IOP PUBLISHING JOURNAL OF PHYSICS: CONDENSED MATTER

J. Phys.: Condens. Matter **20** (2008) 275223 (9pp) doi:10.1088/0953-8984/20/27/275223

Automatic data distribution and load balancing with space-filling curves: implementation in CONQUEST

V Brázdová and D R Bowler

Department of Physics and Astronomy, University College London, Gower Street, London WC1E 6BT, UK,
London Centre for Nanotechnology, 17–19 Gordon Street, London WC1H 0AH, UK
and
Virtual Materials Laboratory, University College London, Gower Street, London WC1E 6BT, UK

E-mail: v.brazdova@ucl.ac.uk and david.bowler@ucl.ac.uk

Received 30 January 2008, in final form 18 April 2008
Published 4 June 2008
Online at stacks.iop.org/JPhysCM/20/275223

Abstract
We present an automatic, spatially local data distribution and load balancing scheme applicable to many-body problems running on parallel architectures. The particle distribution is based on spatial decomposition of the simulation cell. A one-dimensional Hilbert curve is mapped onto the three-dimensional real space cell, which reduces the dimensionality of the problem and provides a way to assign different spatially local parts of the cell to each processor. The scheme is independent of the number of processors. It can be used for both ordered and disordered structures and does not depend on the dimensionality or shape of the system. Details of implementation in the linear-scaling density functional code CONQUEST, as well as several case studies of systems of various complexity, containing up to 55 755 particles, are given.

1. Introduction

In software applications using parallel architectures, distribution of computational workload has a direct impact on the efficiency of the code. On MIMD (multiple instruction multiple data) architectures with distributed memory and hybrid systems with both distributed and shared memory, data has to be divided and assigned to individual processors which then communicate via messages. In partitioning the data we effectively distribute the computational workload and decide which information is stored on a particular processor and which has to be obtained from other processors. The former consumes local memory, the latter computational time. An efficient partitioning scheme must also achieve a balance between the size of messages passed (and hence the demand on local memory) and their number, which affects the sensitivity to latency of communication. The workload assigned to each processor should be approximately the same, interprocessor communication efficient and kept to a minimum and bookkeeping as simple as possible. With larger data sets load balancing needs to be done automatically, so as to provide the possibility of rebalancing on the fly if necessary.

Various data distribution schemes have been developed, including orthogonal recursive bisection, costzones [1], hashed oct-trees [2], and inverse space-filling partitioning [3]. The latter uses space-filling curves. Space-filling curves have also been used for achieving data-locality in sparse-blocked matrix multiplication [4] and data compression [5, 6], as well as in adaptive finite element methods [7].

Particle simulations are one example of multidimensional nonuniform problems that require parallelization and dynamical load balancing. Molecular dynamics codes, for example, routinely handle tens and hundreds of thousands of atoms. Among them are DL_POLY_3 [8], which uses domain decomposition based on the link-cell technique [9], AMBER [10] with the replicated data approach, and NAMD [11], using spatial decomposition of the unit cell, i.e., the unit cell is evenly divided and the parts are distributed over processors.

In electronic structure *ab initio* methods based on density functional theory (DFT) [12], the recent advances in linear-scaling ($\mathcal{O}(N)$, order–N) techniques [13, 14] have made modelling systems with thousands of atoms possible [15–23]. The data distribution strategies again vary; ONETEP [21] creates a Peano space-filling curve (see section 3) to connect atoms and determine their proximity in space [4], the simulation unit cell itself is divided into slabs. The code of Shimojo *et al* [24] uses spatial decomposition [25]. In this paper we present a partitioning scheme using Hilbert space-filling curves and its implementation in the linear-scaling DFT code CONQUEST [15, 16]. CONQUEST does have another, non-automatic load balancing algorithm, based on simulated annealing and a more detailed model of computation and communication costs for both atoms and grid points, which will be described in a forthcoming publication. However, the scheme detailed in this work is fully automatic, simple, and fast. Message passing in CONQUEST is done within the Message passing interface (MPI) standard [26, 27].

2. Data in many-body problems

In many-body simulations the data to be distributed among processors are tied to particles or grid points. These data are used for example to perform matrix multiplication for the former and summation over the latter in constructing overlap integrals. These operations also require information about the surroundings of the particle or grid point.

The particles in question in a DFT simulation are atoms and while we will deal with partitioning of atoms in this work we note that the same method can be applied to any other particles and indeed to grid points as well. We shall call the set of atoms for which a particular processor is responsible the *primary set* of atoms of that processor.

Most of the linear-scaling methods rely heavily on multiplication of sparse matrices and thus it is matrix multiplication that has to be parallelized efficiently [28]. Matrix elements are stored by rows in CONQUEST, so that for any atom i from a processor's primary set all matrix elements X_{ij} are stored on that processor (i and j index atoms). Therefore, in calculating the matrix product

$$C_{ij} = \sum_k A_{ik} B_{kj} \tag{1}$$

only elements B_{kj} need to be communicated between processors, because A_{ik} and C_{ij} are local to the processor.

In CONQUEST, a spatial cutoff on the charge density is imposed. That is, matrix elements of the charge density matrix and related matrices are set to zero for atoms that are further apart than a specified cutoff radius (see, e.g., [29]). It then follows that the more spatially compact the primary set is, the fewer B_{kj} elements need to be communicated between processors and the faster the matrix multiplication.

There is a trade-off between memory usage and latency in communicating the packets with B_{kj} elements. If many packets are sent, the latency will affect the overall speed of communication more than if fewer, larger packets are exchanged. In the latter case, on the other hand, the demand on memory increases. A good compromise is reached when the units of atoms contain between 5 and 20 atoms. However, the size of the primary set depends on the number of processors used. We therefore introduce an intermediate 'unit', which contains the required 5 to 20 atoms and does not depend on the number of processors. We will call such a group a *partition* [28]. A partition is defined in terms of its *real space* boundaries, a definition which ensures spatial locality of atoms in a partition. All atoms within a partition are assigned to one processor. A processor is generally responsible for more than one partition and these partitions form its primary set. The spatial compactness of the primary set therefore fully depends on the spatial compactness of the partitions assigned to each processor.

CONQUEST imposes some additional constraints in the way partitions are created: the partitions must be congruent (of the same size and shape) and have the same orientation.

We will assume that the amount of work related to any atom is about the same. This need not be the case if we have different elements with different numbers of basis functions. We show that we achieve good load balancing even treating all atoms as equivalent in terms of the workload. They could, however, easily be 'weighted' according to the number of their basis functions to improve the load balancing. The weighting can also be defined according to the number of neighbours or indeed any other costs affecting the computational speed and communication load.

With the partitions defined, we need a way to index them and to assign them to processors, creating primary sets as spatially compact as possible. We use space-filling curves for this purpose.

3. Space-filling curves

A one-dimensional interval can be continuously mapped onto n-dimensional space. In other words, a curve that passes through every point of an n-dimensional region can be constructed. Such curves are called space-filling curves, see [30] for details. In two dimensions, several curves, e.g., z-curve [31, 32], Grey-coded [33], Hilbert [34], and Peano [35] curves, map continuously onto a square. It has been shown both by analysing the mathematical properties of the curves [36] and by testing [37] that the Hilbert curve outperforms the z-curve and the Grey-coded curve in clustering objects close in the n-dimensional space along the one-dimensional curve, which in CONQUEST translates into creating the best localized primary sets.

As Hilbert [34] had shown, after dividing the interval into four congruent subintervals and the square into four congruent subsquares, each subinterval can be continuously mapped onto one of the subsquares. This procedure can be continued recursively and the number of intervals (and squares) thus created is 2^{2b}, b being the level of recursion. Note that b is always an integer. In more dimensions, the same algorithm applies with the bisection done in each dimension, i.e., in three dimension the space would be a cube and the bisection would be done in x, y, and z for each level of recursion, resulting in 2^{3b} sub-cubes. The difference in the number of sub-cubes between two levels of recursion in three dimensions is thus $2^{3(b+1)} - 2^{3b} = 7 \times 2^{3b}$. We will call these cubes *Hilbert cubes* and will use them as our partitions.

Peano curves in two dimensions are created by dissecting a square into 3×3 subsquares. In three dimensions, the difference in the number of sub-cubes between two levels of recursion, $3^{3(b+1)} - 3^{3b} = 26 \times 3^{3b}$, is too large to be useful for our purpose: if we have, for example, at a particular level of recursion, Hilbert cubes containing on average 40 atoms, after another division the new, smaller, Hilbert cubes would contain 5 atoms on average. This is because each Hilbert cube was divided into 8 smaller cubes and the number of atoms is now within the required range of 5–20 atoms per partition. But dividing one cube into 27 sub-cubes would result in maximum of 1–2 atoms in the smaller cubes. We would thus have either too many (40) atoms per partition, making even distribution of workload difficult, or too few (1 or 2) atoms per partition, which would lead to too many partitions and too few atoms for efficient packet communication.

The mapping of a space-filling curve onto space preserves spatial locality in the sense that points in adjacent subintervals are adjacent in the n-dimensional space. The opposite is not necessarily true. In three dimensions Hilbert cubes lying next to each other on the Hilbert curve share a face in real space (although cubes adjacent in real space are not always adjacent on the Hilbert curve). This property is exploited in this work to make the primary sets spatially compact.

The Hilbert cubes can be numbered using binary reflected Grey codes [38]. (This does not mean that a Hilbert curve is the same as a Grey-coded curve.) A sequence of Grey numbers is traversed by flipping one bit at a time. In three dimensions, a sequence of $3b$-bit binary Grey codes provides labels which can be used for the 2^{3b} Hilbert cubes. Moreover, such a label encodes the position of the Hilbert cube in the three-dimensional space as well as along the Hilbert curve. A Grey code can be converted to a binary number with the same value. Once converted to decimal, these numbers (*Hilbert integers*) provide convenient labels for each cube. Taking the Hilbert cubes in order of their number, we traverse the entire Hilbert curve, crossing each Hilbert cube once and only once. We thus have a particularly simple labelling scheme.

We use Skilling's code for generation of Hilbert curves and the corresponding Grey-codes and Hilbert integers [39].

4. Implementation in CONQUEST

In CONQUEST the number of partitions needed to distribute the given number of atoms is first estimated (section 4.1) and the cell is divided. The atoms are then assigned to partitions according to their real space coordinates (section 4.2) and the partitions are distributed to processors, taking into account their position along the Hilbert curve (section 4.3).

4.1. Creating partitions

A simulation unit cell can have empty regions. However, to keep advantage of the spatial locality of space-filling curves, we map the Hilbert curve onto the whole unit cell, instead of selecting only the occupied part. Because Hilbert curves require the same number of cubes along each cell side, the Hilbert 'cubes' in CONQUEST are not cubes unless the unit cell itself is cubic. Their cell parameters have the same ratio as the unit cell parameters. This is not a problem, as long as they fulfil the conditions listed in section 2. CONQUEST currently supports only orthorhombic cells, but the partitioning scheme would work also with non-orthorhombic cells, because we are interested in *relative* positions of the Hilbert cubes, not in their absolute coordinates in real space. The only difference would be in how to determine to which Hilbert cube an atom belongs.

Algorithm 1 shows how the initial estimate for the level of recursion of the Hilbert curve (and therefore the number of Hilbert cubes) is calculated. The initial level of recursion is determined by considering the *occupied* volume of the unit cell rather than the whole unit cell size. Hence the abbreviation occ in the names of the variables. First (lines 1–5), the minimum and maximum number of occupied partitions is calculated from the number of atoms and the maximum or minimum number of atoms in a partition, respectively. These are set to 5 and 20 in CONQUEST. The division on line 1 is an integer division and therefore we increment *min occ parts* by 1 to ensure that the minimum number of occupied partitions is not underestimated. For example, consider having between 21 and 39 atoms in the unit cell and a maximum of 20 atoms per partition. The integer division in line 1 would give a maximum of one occupied partition, when we clearly need at least two and thus the correction in line 1. On the other hand, we cannot increment *max occ parts* in the same way: the estimation of *min occ parts* and *max occ parts* has to ensure that the resulting minimal and maximal levels of recursion fall *inside* an integer interval that will give the correct level of recursion. For *max occ parts* we thus need a value less than or equal to the upper limit of this interval and this is achieved by the 'uncorrected' integer division in line 5.

In lines 2–4 we make sure that *max occ parts* is always greater than zero. *min occ parts* is always greater than zero due to the previous '+1' correction. A zero value of either of these variables would cause a zero argument in the logarithm on line 19.

The occupied volume is estimated using the difference between the largest and smallest atomic coordinates in all three dimensions (line 6). If the atoms are all in a plane perpendicular to one of the axes or along a line parallel to one of the coordinate axes, i.e., if the system has 'zero dimensionality' in one or two directions, the estimate is done only for two or one dimensions, respectively, rather than all three (lines 7–15 and 19 in Algorithm 1): in addition, the ratio of the occupied part of the unit cell with respect to the whole unit cell is calculated for each of the three directions x, y, and z and is set to 1 for any direction in which the system has 'zero dimensionality'. This ensures that *occ ratio* for 1D and 2D systems will not skew the estimates done in lines 16–19. In lines 16–18 we calculate the occupied volume as a fraction of the whole cell volume. Note that if the system is one- or two-dimensional, this volume is also effectively one- or two-dimensional, due to setting the corresponding *occ ratios* to 1. Line 19 in Algorithm 1 shows how the initial level of recursion is calculated for the required number of Hilbert cubes. It takes into account the dimensions of the system and the relation between the level of recursion and the number of Hilbert cubes, 2^{nb}.

It can theoretically happen that b_2, which was calculated as a maximum level of recursion, is actually smaller than b_1.

Algorithm 1 Determining initial level of recursion, b.

```
 1: integer division: min occ parts = #atoms / max atoms in partition + 1
 2: if #atoms < min atoms in partition then
 3:     min atoms in partition = #atoms
 4: end if
 5: integer division: max occ parts = #atoms / min atoms in partition
 6: determine the extent of occupied cell, occ cell
 7: dims = dimensions of the structure
 8: for directions $x$, $y$, $z$ do
 9:     if occupied cell < very small then
10:         occ ratio = 1
11:         dims = dims − 1
12:     else
13:         occ ratio = cell parameter / occ cell
14:     end if
15: end for
16: volume = occ ratio(1) × occ ratio(2) × occ ratio(3)
17: $\text{boundary}_1$ = volume× min occ parts
18: $\text{boundary}_2$ = volume× max occ parts
19: $b_i = \log(\text{boundary}_i)$ / log 2 / dims, $i = 1, 2$
20: integer $b_1$ = floor($b_1$)
21: integer $b_2$ = ceiling($b_2$)
22: $b$ = min(integer $b_1$, integer $b_2$)
23: if $b$ is 0 then
24:     $b = 1$
25: end if
```

Algorithm 2 Assigning atoms to Hilbert cubes.

```
 1: while refine do
 2:     refine = false
 3:     allocate all arrays dependent on $b$
 4:     for all atoms do
 5:         determine the real space partition
 6:         look up the Hilbert number of the partition OR calculate it (axes_to_transpose)
 7:     end for
 8:     for all Hilbert cubes do
 9:         sum atoms in the HC
10:         if number of atoms > global maximum then
11:             refine = true ⇒ refine partitioning
12:             deallocate all arrays dependent on $b$
13:         end if
14:     end for
15: end while
16: for all partitions do
17:     calculate the Hilbert number
18: end for
```

Such a situation could occur if the original values of maximum and minimum number of atoms per partition were close. For this reason a minimum of b_1 and b_2 is taken in line 22. We have not, however, come across such a situation in any of the test cases: b_2 is usually either equal to b_1 or is larger by one than b_1. However, to make sure the algorithm is general, we do leave the minimum here.

Lines 23 and 24 ensure that we always have more than one partition.

The initial level of recursion is then used to generate the partitions by spatial decomposition of the cell.

4.2. Assignment of atoms to partitions

For each atom its partition is determined according to its real space coordinates (Algorithm 2) and the position of the partition on the Hilbert curve is then found. This part of the code uses Skilling's `axes_to_transpose` routine [39], which takes relative real space coordinates and returns a set of 'Hilbert space coordinates', which can be converted to the corresponding Hilbert integer.

Each Hilbert cube would be divided into 8 sub-cubes in the next level of recursion. This somewhat limits the options for refining the partitioning, because we cannot always get the 5–20 atoms that would be optimal for the matrix multiplication routines. We impose a looser condition of the maximum number of atoms per partition instead. This number has been set to 34 after testing the scheme on different systems. However, it can be adjusted in the input of each calculation with a keyword (General.MaxAtomsPartition). If any partition contains more than the maximum number of atoms, the partitioning is repeated with the level of recursion increased by 1 ($b = b + 1$). With b estimated in the way described in Algorithm 1 and General.MaxAtomsPartition set to the default value, one refinement at most is usually needed.

An alternative way to assign atoms to partitions is to loop over all partitions and search for atoms that fall into each one. Here the necessity to refine the partitioning is identified much more quickly because we do not wait until all atoms are assigned to partitions to compare with General.MaxAtomsPartition. However, the atoms have to be sorted according to their coordinates in x, y, and z (i.e., in CONQUEST three calls of mergesort, which scales as $n \log n$) first. The parallelization is also more complicated. We therefore use the scheme in sequential runs, where looping over all atoms more than once cannot be shared by more processors.

It would seem that atoms that fall on partition boundaries can be assigned to either of the partitions in question, as long as the assignment is consistent throughout the cell. However, because fractional atomic coordinates in crystals are usually given in the range [0, 1) rather than (0, 1], it turns out that assigning atoms to partitions further away from the coordinate system origin gives more balanced results. Consider for example a one-dimensional system with atoms at positions 0.00, 0.25, 0.50, and 0.75. If we have four partitions along the line and assign atoms to partitions 'closer to 0', we will get two atoms in the first partition (0.00 and 0.25), one in each the second and third partition, and none in the fourth partition. Whereas assigning the atoms 'closer to 1' will put one atom into each of the partitions. The $(4 \times 4 \times 4)$ Si crystal (section 5) is a case in point: assigning atoms on boundaries to the partitions 'closer to 1' yields eight atoms in every partition and perfect load balancing. In contrast, assigning the boundary atoms to partitions 'closer to 0' gives between 1 and 10 atoms per partition.

4.3. Assignment of partitions to processors

Algorithm 3 details the way partitions are assigned to processors. First, an initial estimate of the number of atoms per processor is made (line 1). The Hilbert curve is then traversed and partitions are added to a processor until the number of atoms is greater than or equal to the initial estimate (lines 4–7). This can result in too many atoms on some processors and no atoms on the last ones. If this is the case, the initial estimate is decreased by 1 (lines 8–11) and the procedure is repeated. The same is done if the last processor has too few atoms compared to the rest of the processors (lines 12–15). If, however, the number of atoms on the last processor is significantly larger than on the previous ones (i.e., atoms are 'left over'), the code starts with the last processor and shifts partitions one by one from processor n to $n-1$ until the difference in the number of atoms on those two processors is less than or equal to the average number of atoms in an *occupied* partition. That is, it takes into account the average number of atoms that can be shifted by shifting whole partitions (lines 16–22).

Algorithm 3 Assigning partitions to processors.

```
 1: max atoms per processor = #atoms / #processors
 2: reassign = true
 3: while reassign do
 4:     for all partitions do
 5:         add partitions to a processor until #atoms on the
            processor ⩾ max atoms per processor
 6:     end for
 7:     reassign = false
 8:     if any processor has no atoms then
 9:         decrement max atoms per processor by 1
10:         reassign = true
11:     end if
12:     if too few atoms on last processor compared to the other
        processors then
13:         decrement max atoms per processor by 1
14:         reassign = true
15:     end if
16:     if too many atoms on last processor compared to the
        other processors then
17:         calculate average #atoms in occupied partition
            (correction)
18:         start with the last processor
19:         while #atoms on neighbouring processors differs by
            > correction do
20:             shift one partition to the previous processor
21:         end while
22:     end if
23:     if max atoms per processor has been decremented to 0
        then
24:         print error message: too many processors used
25:         exit program
26:     end if
27: end while
28: reshuffle empty partitions
```

If the maximum number of atoms per processor has been decremented to zero, it means that there are not enough occupied partitions to be distributed to all processors and the program exits with an error message, rather than leaving some processors empty (lines 23–26).

Finally, after a balanced distribution of atoms is obtained, the code searches along the Hilbert curve for strings of empty partitions that fall on the boundary of partitions assigned to different processors. If such a string is found and better balance in the number of partitions can be achieved, some of the empty partitions are shifted to the other processor (line 28).

Rebalancing during a calculation is not yet implemented in CONQUEST, however, the procedure is simple: the partitioner has to be rerun and the assignment of atoms to partitions checked against the original one. Apart from updating the assignment, information about atoms that have moved to a different partition has to be broadcast between processors only if the new partition is on a different processor. Here again the spatial locality of the primary sets, achieved with the help of the space-filling curve, keeps the interprocessor communication as low as possible.

5. Case studies

We ran our tests on an IBM p690 system consisting of Regatta nodes (logical partitions), which have 32 POWER4 1.3 GHz processors each and share 64 GB of memory. Tasks running on one node communicate via shared memory or message passing using shared memory. Tasks running on adjacent nodes communicate using the IBM High Performance Switches [40]. The system shows performance variations of about 10% and we therefore ran every calculation five to six times and took the mean of the recorded times. We always considered the CPU time on the processor that took the longest to complete its task. However, the variation in the CPU times across the processors is so small that plotting the mean or even the minimal CPU time does not make any difference. To ensure that no other jobs interfered with the test calculations, the whole node was always reserved, even for calculations using fewer than 32 processors. We use a single-ζ basis set for all atoms to cut the overall CPU time for the tests, but the partitioning scheme itself does not depend on the basis set used.

CONQUEST uses an overlap matrix S of localized orbitals $\phi_{i\alpha}(\mathbf{r})$:

$$S_{i\alpha,j\beta} = \int \mathrm{d}\mathbf{r}\, \phi_{i\alpha}(\mathbf{r})\phi_{j\beta}(\mathbf{r}), \qquad (2)$$

where i, j index the individual atoms and α, β the localized functions. We tested the time needed for a calculation of the inverse of the S matrix by Hotelling's method [41, 42] for different systems. Virtually all computational time in this algorithm is spent on matrix multiplication. We also calculated one energy minimization step for the $(4 \times 4 \times 4)$ Si crystal described below and compared the scaling to the scaling of the inverse matrix evaluation. The results are virtually the same (see figure 1, see below for discussion of S range) and we therefore use the inverse matrix calculation to test the overall scaling.

We use standard deviation defined as

$$\sigma = \left(1/(n-1)\sum_{i=1}^{n}(N_i - \bar{N})^2\right)^{1/2}$$

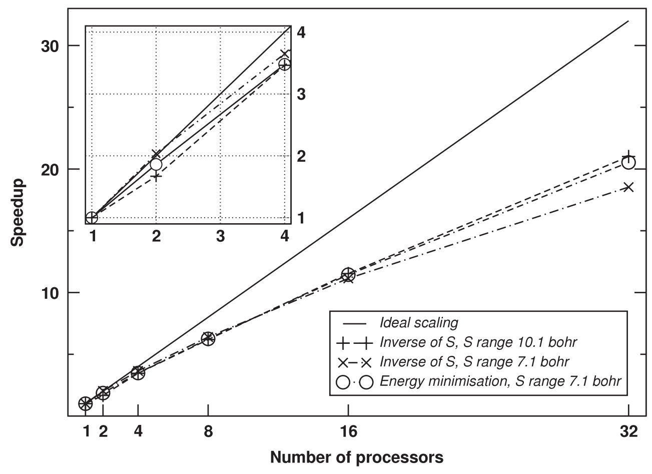

Figure 1. Bulk Si, 512 atoms. Speedup with respect to number of processors for one energy minimization loop and for calculation of an inverse matrix with two different L range values. The inset shows an enlarged part of the main graph.

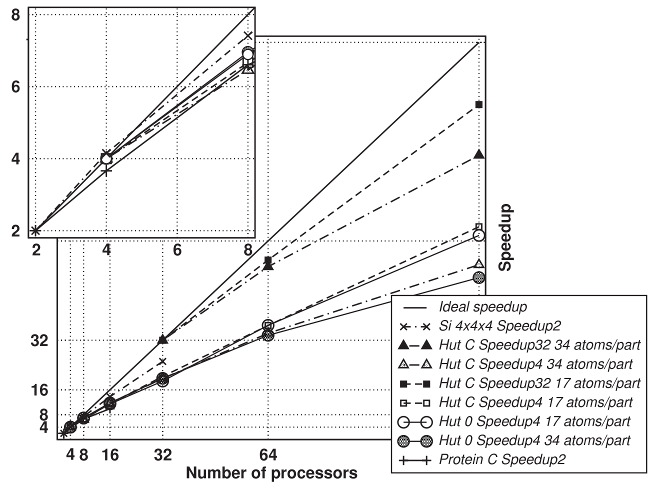

Figure 2. Speedup of the calculation of the inverse S matrix. Speedup2, 4, 32 is speedup with respect to CPU time on 2, 4, and 32 processors, respectively. Maximum number of atoms per partition is listed.

to assess the balance of distribution of atoms over processors. Here n is the number of processors, N_i the number of atoms on processor i, and $\bar{N}$ the mean number of atoms on a processor.

The first system used for the tests, the Si crystal, can be perfectly partitioned, i.e., all the partitions will have the same number of atoms and will be evenly divided on 2^n processors. Since all atoms in the unit cell are the same, they are described using the same basis and are equal in terms of workload. We use a $(4 \times 4 \times 4)$ Si unit cell that contains 512 atoms. The second test case, a Ge hut cluster on Si(001) surface, presents a system more complicated to partition because it is not a perfect crystal and the unit cell contains empty regions. The structure is, however, still quite regular, not only within the Si slab, but even in the Ge cluster. We use two models, one with the hut cluster positioned in the middle of the cell (*Hut C*) and one where the atoms are centred at the coordinate origin (*Hut 0*). The latter leads to vacuum in the centre of the unit cell and thus presents a slightly more complicated system to the partitioner. The cell is significantly larger than that of the Si crystal (86.9 Å × 86.9 Å × 40.7 Å and 21.7 Å × 21.7 Å × 21.7 Å, respectively) and contains 4363 atoms. With the same number of valence electrons for Ge and Si the workload associated with all atoms is the same.

The third system is a protein and partitioning it adds two new challenges. First, the system is spatially inhomogeneous compared to the well ordered Si crystal and even the hut on the Si surface. Second, the atoms are no longer equivalent in terms of workload and even with the number of atoms well balanced across the processors the scaling is expected to be less efficient. We chose two protein models, one with 323 atoms for testing the scaling and one with 55 755 atoms, for which we studied only the distribution of atoms over processors.

Si crystal

Figure 1 shows scaling for two different values of the spatial cutoff imposed on the S matrix. While the calculation with the larger, 10.1 bohr, cutoff is computationally more demanding,

Table 1. Distribution of atoms on processors for the Ge hut cluster, Hut 0.

Processors	Ideal[a]	Mean	Min[b]	Max[c]	Std. dev.[d]
4	1165.8	1165.8	1163	1170	3.10
8	582.9	582.9	572	600	10.36
16	291.4	291.4	276	308	11.41
32	145.7	145.7	124	159	11.22
32[e]	145.7	145.7	144	150	1.57
64	72.9	72.9	55	105	13.98

[a] Number of atoms/number of processors.
[b] Lowest number of atoms on a processor.
[c] Highest number of atoms on a processor.
[d] Standard deviation.
[e] Smaller partitions. See section *Ge hut cluster* for discussion.

6.7× than with a 7.1 bohr cutoff, it scales better, because each transmitted packet is longer and the transfer is more efficient.

Ge hut cluster

The Ge hut cluster on Si(001) surface we used contains 3072 Si and 1291 Ge atoms. As can be seen in figure 2, the scaling up to 32 processors is almost as good as for the perfect Si crystal for both Hut C and Hut 0. (When using 64 or 128 processors the calculation runs on two and four nodes, respectively.) Table 1 lists distribution of atoms over processors for Hut 0, i.e., the more complicated of the two. The average number of atoms on a processor matches exactly the ideal number of atoms per processor. The average number of atoms per occupied partition is 23.4, with a range of 1–34 atoms per partition (standard deviation 10.5). The range of number of atoms per processor on 8, 16, and 32 processors varies by about the average number of atoms per partition, which suggests that improvements in load balancing would have to achieve a smaller number of atoms per partition. This is indeed the case, as shown in figure 2 and in table 1. Specifically, in table 1 compare the

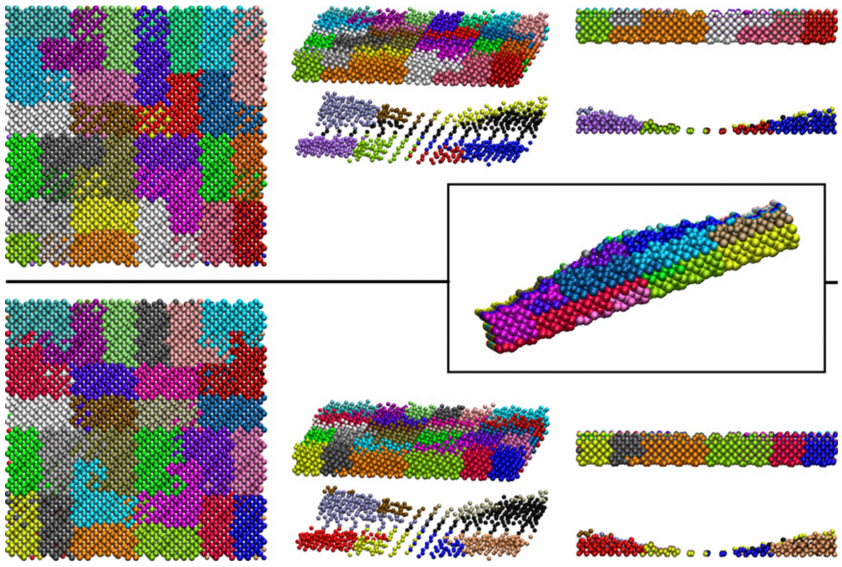

Figure 3. Primary sets of atoms in the Ge hut cluster divided among 32 processors. Atoms of the same colour belong to the same primary set. Top: Hut 0, maximum 34 atoms in a partition. Bottom: Hut 0, maximum 17 atoms in a partition. Framed in the middle: Hut C, maximum 17 atoms per partition.

(This figure is in colour only in the electronic version)

two lines showing distribution of atoms over 32 processors. With smaller partitions the minimum and maximum number of atoms per processor differs by only six atoms, compared to 35 with larger partitions. The standard deviation is also much smaller when using smaller partitions. Naturally, smaller partitions can increase the overall computational effort somewhat.

Using only four processors allows the atoms to be distributed over the processors very evenly, because of the four-fold symmetry of the hut. Each processor thus gets a quarter of the hut and that accounts for the small standard deviation, compared to distributions over more processors. For the other numbers of processors, the standard deviation remains fairly constant, which shows that our method is really independent on the number of processors.

Hut C scales slightly better than Hut 0, however, the main difference in scaling is not caused by the position of the atoms in the unit cell but rather by the size of the partitions, with smaller partitions leading to better scaling. This is due to more spatially compact primary sets that can be built up from smaller partitions, cf the top and bottom pictures in figure 3. Note that although Hut C and Hut 0 look different in figure 3, periodic boundary conditions are used in the calculations and therefore the only difference between Hut C and 0 is their absolute position in the unit cell, not the atomic structure. The original Hilbert curve can be partly traced in figure 3; notice the initial bisection in the x and y direction, visualized by the straight sharp boundaries of the primary sets in the middle of the cell of Hut 0 (figure 3 top left, bottom left). Likewise, the subsequent divisions can be seen in the primary set boundaries parallel to x and y. Similarly, the initial division in the z direction is visible in the side view of Hut C (figure 3, middle) as the horizontal boundary between the primary sets.

Figure 4 shows the CPU time summed over the number of processors, which represents the 'total' CPU time the

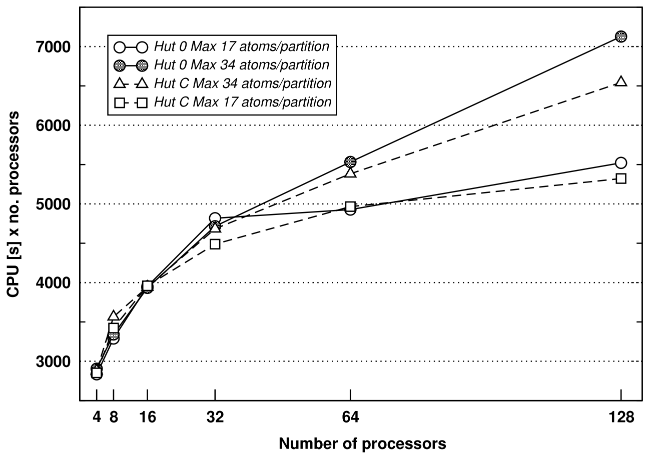

Figure 4. Ge hut clusters, 'total' CPU time: CPU time for a calculation summed over the number of processors used.

calculation took. It combines information about speedup and the absolute CPU time. Clearly the larger number of partitions does not slow the calculation down. Even though the total CPU time increases as we go from 4 to 32 processors, between 32 and 128 processors the increase is quite small for the calculations with more compact primary sets. Scaling of CONQUEST calculations on *large* numbers of processors is thus very efficient.

Small protein

Table 2 lists the distribution of atoms to processors for the smaller protein. The scaling is shown in figure 2. The maximum number of atoms in an occupied partition is 13. We note that with only 323 atoms, using 16 processors results in

Table 2. Distribution of atoms on processors for the small test protein.

Processors	Ideal[a]	Mean	Min[b]	Max[c]	Std. dev.[d]
4	80.8	71	68	75	3.16
8	40.4	35.5	32	40	3.16
16	20.2	17.8	12	29	4.96

[a] Number of atoms/number of processors.
[b] Lowest number of atoms on a processor.
[c] Highest number of atoms on a processor.
[d] Standard deviation.

too few atoms per processor and we get 14–29 atoms on a processor. The load levelling and thus the speedup might in this case be improved by 'weighting' atoms according to the number of their basis functions.

Large protein

We have also used the automatic partitioner on a protein structure containing 55 755 atoms, although we have not done scaling tests. The atom distribution is listed in table 3. There are, on average, 10.6 atoms in a partition, with 23 atoms in a partition at most. The standard deviation of the atom distribution is 5.11.

6. Discussion and conclusions

Specific applications of CONQUEST have not been presented in this paper as the main purpose is to describe the implementation of an automatic load balancing procedure as part of the code; we have other, optimizing approaches to load balancing which involve significant pre-processing and will be described elsewhere. While the parallelization which can be achieved with the optimizing schemes is better, it is significantly harder to re-assign atoms dynamically in this scheme (i.e., as the atoms are displaced); with the procedure described in this paper, molecular dynamics with large atomic displacements will not present any problems. The scheme described in this paper has been used for recent CONQUEST applications, and we give a brief overview here.

The enzyme dihydrofolate reductase (DHFR) catalyses the reduction of dihydrofolate (DHF) and is an important enzyme in metabolism. However, the mechanism by which DHFR catalyses DHF but not other precursors such as folate is unclear. Preliminary studies [43] using CONQUEST indicate that polarization of the DHF particularly by proteins on the Met20 loop are important. Other biological systems being studied with CONQUEST include ATPase, gramicidin-A and DNA. We have recently published a study showing that the electronic structure in a ten base-pair piece of DNA in water is remarkably localized, allowing excellent linear-scaling performance [44].

Semiconductor surfaces often show complex reconstructions, and strained heteroepitaxial growth can give rise to three-dimensional structures; a case in point is the 'hut' clusters which form when Ge is grown on Si(001). Using CONQUEST we have modelled the Ge(105) reconstruction (which forms on the faces of the huts) and found both that good accuracy is possible with relatively modest cutoffs and that there are several subtly different reconstructions which are possible on the surface [45]. The ultimate goal of these studies was to investigate the mechanisms behind formation of the hut clusters; using CONQUEST to model entire huts (as shown earlier in section 5) we have found that an energetic mechanism explains their formation [46]. The algorithm described in this paper will enable more studies like those described briefly to be performed, as well as more ambitious (e.g., long molecular dynamics runs).

Table 3. Distribution of atoms on processors for the large test protein.

Processors	Ideal[a]	Mean	Min[b]	Max[c]	Std. dev.[d]
4	13 938.8	13 938.8	13 934	13 943	4.92
8	6 969.4	6 969.4	6 963	6 978	4.98
16	3 484.7	3 484.7	3 479	3 496	4.61
32	1 742.3	1 742.3	1 732	1 762	5.99
64	871.2	871.2	860	890	6.13
128	435.6	435.6	425	448	4.92

[a] Number of atoms/number of processors.
[b] Lowest number of atoms on a processor.
[c] Highest number of atoms on a processor.
[d] Standard deviation.

In summary, we developed a simple and effective partitioning scheme using Hilbert space-filling curves and implemented it in the linear-scaling DFT code CONQUEST. The scheme is based on spatial decomposition of the simulation cell into even partitions. The particles are distributed according to their position in the partitions. Spatial locality of partitions and hence atoms on processors is achieved by using a Hilbert curve mapped on the simulation cell. The scheme is fully automatic. The load is balanced evenly for systems of varying size and complexity.

Acknowledgments

The tests were carried out on the IBM pSeries 690 system at Norddeutscher Verbund für Hoch-und Höchstleistungsrechnen (HLRN). We would like to thank Bernd Kallies for technical support.

DRB is funded by the Royal Society. We acknowledge useful discussions with Mike Gillan, Tsuyoshi Miyazaki and Antonio Torralba.

The work was supported by the grant 'Modelling Charge Transport in Conducting Polymers and Biological Systems' from the IRC in Nanotechnology funded by the EPSRC.

References

[1] Singh J P, Holt C, Totsuka T, Gupta A and Hennessy J 1995 *J. Parallel Distrb. Comput.* **27** 118–41
[2] Warren M S and Salmon J K 1995 *Comput. Phys. Commun.* **87** 266–90
[3] Pilkington J R and Baden S B 1996 *IEEE Trans. Parallel Distrib. Syst.* **7** 288–300
[4] Challacombe M 2000 *Comput. Phys. Commun.* **128** 93–107
[5] Bially T 1969 *IEEE Trans. Inf. Theory* **15** 658–64

[6] Omeltchenko A, Campbell T J, Kalia R K, Liu X, Nakano A and Vashishta P 2000 *Comput. Phys. Commun.* **131** 78–85
[7] Bauer A C and Patra A K 2004 *Int. J. Numer. Methods Eng.* **59** 337–64
[8] Todorov I and Smith W 2004 *Phil. Trans. R. Soc.* **362** 1835–52
[9] Pinches M R S, Tildesley D and Smith W 1991 *Mol. Simul.* **6** 51–87
[10] Case D A, Darden T A, Cheatham T E III, Simmerling C L, Wang J, Duke R E, Luo R, Merz K M, Pearlman D A, Crowley M, Walker R C, Zhang W, Wang B, Hayik S, Roitberg A, Seabra G, Wong K F, Paesani F, Wu S, Brozell S, Tsui V, Gohlke H, Yang L, Tan C, Mongan J, Hornak V, Cui G, Beroza P, Mathews D H, Schafmeister C, Ross W S and Kollman P A 2006 AMBER 9 http://amber.scripps.edu/
[11] Phillips J C, Braun R, Wang W, Gumbart J, Tajkhorshid E, Villa E, Chipot C, Skeel R D, Kale L and Schulten K 2005 *J. Comput. Chem.* **26** 1781–802
[12] Martin R 2004 *Electronic Structure: Basic Theory and Practical Methods* (Cambridge: Cambridge University Press)
[13] Goedecker S 1999 *Rev. Mod. Phys.* **71** 1085–123
[14] Shao Y, Molnar L F, Jung Y, Kussmann J, Ochsenfeld C, Brown S T, Gilbert A T B, Slipchenko L V, Levchenko S V, O'Neill D P Jr, DiStasio R A, Lochan R C, Wang T, Beran G J O, Besley N A, Herbert J M, Lin C Y, Van Voorhis T, Chien S H, Sodt A, Steele R P, Rassolov V A, Maslen P E, Korambath P P, Adamson R D, Austin B, Baker J, Byrd E F C, Dachsel H, Doerksen R J, Dreuw A, Dunietz B D, Dutoi A D, Furlani T R, Gwaltney S R, Heyden A, Hirata S, Hsu C P, Kedziora G, Khalliulin R Z, Klunzinger P, Lee A M, Lee M S, Liang W, Lotan I, Nair N, Peters B, Proynov E I, Pieniazek P A, Rhee Y M, Ritchie J, Rosta E, Sherrill C D, Simmonett A C, Subotnik J E III, Woodcock H L, Zhang W, Bell A T, Chakraborty A K, Chipman D M, Keil F J, Warshel A, Hehre W J III, Schaefer H F, Kong J, Krylov A I, Gill P M W and Head-Gordon M 2006 *Phys. Chem. Chem. Phys.* **8** 3172–91
[15] Bowler D R, Miyazaki T and Gillan M J 2002 *J. Phys.: Condens. Matter* **14** 2781–98
[16] Bowler D R, Choudhury R, Gillan M J and Miyazaki T 2006 *Phys. Status Solidi* b **243** 989–1000
[17] Soler J M, Artacho E, Gale J D, García A, Junquera J, Ordejón P and Sanchez-Portal D 2002 *J. Phys.: Condens. Matter* **14** 2745–79
[18] Ozaki T and Kino H 2005 *Phys. Rev.* B **72** 045121
[19] Ozaki T, Kino H, Yu J, Han M J, Kobayashi N, Ohfuti M and Ishii F COMPUTER CODE OPENMX http://www.openmx-square.org/
[20] Challacombe M 1999 *J. Chem. Phys.* **110** 2332–42
[21] Haynes P D, Skylaris C K, Mostofi A A and Payne M C 2006 *Phys. Status Solidi* b **243** 2489–99
[22] Shimojo F, Kalia R K, Nakano A and Vashishta P 2001 *Comput. Phys. Commun.* **140** 303–14
[23] Sodt A, Subotnik J E and Head-Gordon M 2006 *J. Chem. Phys.* **125** 194109
[24] Shimojo F, Kalia R K, Nakano A and Vashishta P 2005 *Comput. Phys. Commun.* **167** 151–64
[25] Nakano A, Kalia R K, Vashishta P, Campbell T J, Ogata S, Shimojo F and Saini S 2002 *Sci. Prog.* **10** 263–70
[26] 1993, 1994, 1995 *MPI: A Message-Passing Interface Standard* University of Tennessee, Knoxville, Tennessee
[27] 1995, 1996, 1997 *MPI-2: Extensions to the Message-Passing Interface* University of Tennessee, Knoxville, Tennessee
[28] Bowler D R, Miyazaki T and Gillan M J 2001 *Comput. Phys. Commun.* **137** 255–73
[29] Hernández E, Gillan M J and Goringe C M 1996 *Phys. Rev.* B **53** 7147
[30] Sagan H 1994 *Space-Filling Curves* (New York: Springer)
[31] Morton G M 1966 A computer oriented geodetic data base, and a new technique in file sequencing *Technical Report* IBM Canada, Ltd
[32] Orenstein J A 1986 Spatial query processing in an object-oriented database system *SIGMOD Conf.* pp 326–36
[33] Faloutsos C 1988 *IEEE Trans. Softw. Eng.* **14** 1281–393
[34] Hilbert D 1891 *Math. Ann.* **38** 459–60
[35] Peano G 1890 *Math. Ann.* **36** 157–60
[36] Moon B, Jagadish H V, Faloutsos C and Salz J 2001 *IEEE Trans. Knowl. Data Eng.* **13** 124–41
[37] Jagadish H V 1990 Linear clustering of objects with multiple atributes *SIGMOD Conf.* pp 332–42
[38] Gray F 1953 Pulse code communication *US Patent Specification* 2,632,058
[39] Skilling J 2004 *CP707, Bayesian Inference and Maximum Entropy Methods in Science and Engineering: 23rd Int. Workshop* ed G Erickson and Y Zhai (New York: American Institute of Physics) pp 381–7
[40] Kallies B and Vortisch W 2006 *LoadLeveler—The IBM SP Batch Queuing System* http://www.hlrn.de/doc/loadl/index.html
[41] Pan V and Reif J 1985 *STOC '85: Proc. 17th Annual ACM Symp. on Theory of Computing* (New York, NY: ACM Press) pp 143–52
[42] Press W H, Flannery B P, Teukolsky S A and Vetterling W T 1992 *Numerical Recipes in FORTRAN: The Art of Scientific Computing* 2nd edn (Cambridge: Cambridge University Press) p 49
[43] Gillan M J, Bowler D R, Torralba A S and Miyazaki T 2007 *Comput. Phys. Commun.* **177** 14
[44] Otsuka T, Miyazaki T, Ohno T, Bowler D R and Gillan M J 2008 *J. Phys.: Condens. Matter* **20** 272022
[45] Miyazaki T, Bowler D R, Choudhury R and Gillan M J 2007 *Phys. Rev.* B **76** 115327
[46] Miyazaki T, Ohno T, Bowler D R and Gillan M J 2008 *Phys. Rev. Lett.* submitted