Quantum Molecular Dynamics: O(N) Tight-Binding on the T3D

A. Canning, Cray Research, Switzerland, PSE, EPFL, and *G. Galli*, *F. Mauri*, *A. De Vita*, and *R. Car*, IRRMA (Institut Romand de Recherche Numérique en Physique des Matériaux, EPFL, Lausanne, Switzerland

Abstract

We discuss the implementation of an O(N) tight-binding molecular dynamics code on the Cray T3D parallel computer. The localisation introduces a sparse nature to the orbital data and Hamiltonian matrix, greatly changing the coding on parallel machines compared to non-localised systems. The data distribution, communication routines and dynamic load-balancing scheme of the program are presented in detail together with the speed and scaling of the code on various homogeneous and inhomogeneous physical systems. Performance results will be presented for systems of 2048 to 32768 atoms on 32 to 512 processors. We discuss the relevance to quantum molecular dynamics simulations with localised orbitals, of techniques used for programming short-range classical molecular dynamics simulations on parallel machines. The absence of global communications and the localised nature of the orbitals makes these algorithms extremely scalable in terms of memory and speed on parallel systems with fast communications.

1 Introduction

The evaluation of the Energy in Density Functional Theory (DFT) calculations using the Local Density Approximation (LDA), such as those used in Car-Parrinello type Molecular Dynamics (MD) simulations [1], requires of order N^3 operations where N is the number of atoms in the system. This limits the size of systems that can be treated with these first-principles approaches to a few hundred, even with the most powerful modern computers [2]. Tight-Binding (TB) models, while still presenting a quantum mechanical approach to molecular dynamics, have a much simpler Hamiltonian and smaller basis set, greatly reducing the computational cost of an MD step. However, the scaling of the computational cost is still of O(N^3) even though the prefactor is much smaller than for first-principles calculations. This limits the number of atoms that can be studied in TB MD simulations to about a thousand, on the most powerful modern supercomputers. In order to extend the application of quantum MD to larger systems, many new (so-called O(N)) techniques have been introduced in recent years, O(N) meaning that their computational cost grows only linearly with the system size. Some of these approaches are based on an orbital formulation of the electronic properties [3, 4, 5, 6, 7] while others are based on the direct calculation of either the one-electron Green's function [8, 9] or the density matrix [10, 11]. The basic ingredients of the orbital based O(N) approaches are the spatial localisation of the orbitals and a novel energy functional which does not involve explicit orthogonalisation of the orbitals (ϕ_I), or inversion of the overlap matrix ($S_{IJ} =< \phi_I|\phi_J >$). Most methods for finding the ground state, without explicit orthogonalisation of the orbitals, require the inversion of the overlap matrix in order to ensure orthogonality of the wavefunctions at the minimum of the energy functional.

In this paper we will discuss our approach to the parallel programming of the O(N) energy functional proposed independently in references [4, 12] and [7] and extended to a non-orthogonal representation of the orbitals in reference [13]. A lot of the techniques presented in this paper are applicable to other O(N) algorithms in the context of a TB model and to LDA implementations of O(N) methods. In particular the density matrix approach to O(N) TB [10, 11] could be programmed on parallel machines in the same way as our orbital approach. We will not describe in detail the calculation of all the different terms for our MD simulation but will rather present an overview illustrated with a few typical examples such as the calculation of the overlap matrix. We adopt the TB Hamiltonian of Xu *et al.* [15] in the context of an MD simulation. This TB O(N) scheme has already been used with a serial code to study fullerene impacts on a semiconducting surface [16] and one of the target problems for our parallel code is an inhomogeneous system of fullerenes deposited on a diamond surface.

In the next section of this paper we will discuss the implementation of the O(N) TB MD in a serial code. In particular we will discuss how the energy functional and its derivative are calculated in the context of sparse matrices. In the third section we will look at techniques used in programming classical MD simulations on parallel machines that are relevant to our quantum simulations. In the fourth section we will present the data distribution and communication routines used in the parallel code. In section five some optimisation techniques which are relevant to this code and applicable to RISC processors of the type found in the Cray T3D will be presented. In section six we will discuss dynamic load-balancing and its application to inhomogeneous systems such as fullerenes arriving on a diamond surface. In the last section we will give results for the speed and scaling of the code on various systems.

2 O(N) Tight-Binding MD Code

Our code uses the conjugate gradient (CG) method to minimise the energy functional for the electronic degrees of freedom and the Verlet algorithm [18] to update the atomic positions in an MD simulation. The basic structure of the code is two large nested loops with the inner loop performing the minimisation to the ground state of the wavefunctions and the outer loop the atomic dynamics. The minimisation of the energy functional in O(N) calculations typically takes more CG steps than in non-localised methods where the CG steps already take the largest percentage of the calculation time in an MD run. Therefore, in this paper, we will devote most of the discussion of parallelisation and optimisation to the CG part of the code rather than the atomic update. However, most of the calculations for the atomic update are very similar in nature to the CG step so that most of the discussion of the programming techniques for the CG step is also relevant to the atomic update.

In the TB model the wavefunctions are expanded on the basis set of atomic orbitals at all the atomic sites. In the context of the TB O(N) algorithm a localisation region (LR) for the orbital representation of the wavefunctions can then be identified with a set of atomic sites centered around a given atom. In our code we define the localisation region as including up to the N_hth nearest neighbour of the center atom where a cut-off distance defines, whether or not, an atom is a nearest neighbour. In this way, by varying the value of N_h, we vary the size of the localisation region of the orbital. The localised orbitals can then be represented by a set of coefficients c_{im}^{jk} such that

$$|\phi_i^m> = \sum_{j \in LR_i} \sum_k c_{im}^{jk} |\Phi_j^k>, \qquad (1)$$

where i is the atomic site on which the orbital is localised and m is the index for the orbitals localised at site i; j is summed over the atomic sites $j = 1, \ldots, N$, but c_{im}^{jk} is only non-zero in the localisation region LR_i of site i i.e. $j \in LR_i$; k is summed over the set of basis functions ($k = 1, 4$ the s,p_x,p_y and p_z atomic orbitals in our case where $|\Phi_j^k>$ is the k'th basis function at site j). In the program the orbitals are stored as sparse coded matrices in the sense that we only store the non-zero values of c_{im}^{jk}.

The complete set of orbitals for the system is stored as a four dimensional sparse coded matrix of the form $C(k, m, j_s, i)$ and a two dimensional matrix $NEIG(j_s, i)$ lists the true site numbers for each C value. We use the notation j_s to denote that this is the sparse coded dimension in the matrix. Thus $C(k, m, j_s, i)$ is the value of C at site $j = NEIG(j_s, i)$ in the localisation region centered on atom i. A further vector $NNEIG(i)$, contains the number of sites in each localisation region (i.e. $j_s = 1, \ldots, NNEIG(i)$ since there are $NNEIG(i)$ sites in the localisation region LR_i). With these definitions equation 1 now becomes,

$$|\phi_i^m> = \sum_{j_s=1}^{NNEIG(i)} \sum_k C(k, m, j_s, i)|\Phi^k_{NEIG(j_s,i)}> . \quad (2)$$

It would be prohibitive, in terms of memory, and computationally inefficient to work with the full C matrix containing all the zero values of the C's outside the localisation regions. It should also be noted that while for non-localised problems the overlap matrix, Hamiltonian etc. are usually calculated with Basic Linear Algebra Subprograms (BLAS), typically written in assembly language and highly machine-optimised, we now have to hand code the sparse matrix multiplications. This leads to codes which typically have a MFLOP speed lower than non-localised codes. This will be discussed later in more detail in the section on optimisation.

In this paper we will discuss results for a code using the energy functional for non-orthogonal localised orbitals presented in reference [13]. This is essentially the same as the energy functional for localised orbitals presented in [4, 12] except that the number of orbitals is chosen to be larger than the number of occupied electronic states. All the coding techniques we will discuss apply to both functionals. For more discussion on the physics behind these energy functionals the reader should consult these papers and the references therein. The energy functional used in our program is given by

$$E(\{\phi_i^m\}, \eta) = 2 \sum_{i,j,m,n} Q_{ij}^{mn} < \phi_j^m|H - \eta|\phi_i^n > + \eta\mathcal{N}, \quad (3)$$

where $\mathcal{N}$ is the number of electrons in the system. The number of orbitals is chosen to be larger than $\mathcal{N}/2$, the number of occupied states which is half the number of electrons $\mathcal{N}$, due to spin degeneracy. In all our calculations we choose the number of orbitals to be $\frac{3\mathcal{N}}{4}$ corresponding to three orbitals for each of the N localisation regions. Thus in equation 3 the sum over orbitals is split into a sum over the sites $i = 1, \ldots, N$ and a sum $m = 1, 2, 3$ over the orbitals centered on the same site. The number of sites is $N = \frac{\mathcal{N}}{4}$ since we are using the TB Hamiltonian of Xu *et al.* [15] for carbon based systems and the number of valence electrons for carbon is four. η is a scalar parameter (playing the role of the chemical potential) which is adjusted to give the correct number of electrons and Q is the first order expansion of the inverse of the overlap matrix

$$S^{-1} \approx Q = 2I - S, \quad (4)$$

where S is the overlap matrix ($S_{ij}^{mn} =< \phi_i^m|\phi_j^n >$) and I is the identity matrix. It was pointed out in reference [15] that in order to control unphysical charge transfer for certain systems studied with TB models e.g. systems with dangling bonds, it is often useful to add a Hubbard like term to the energy functional. This term is present in our program and can be switched on and off depending on what type of system is under study. It is given by $\frac{1}{2}U(q_i - 4)^2$ where q_i is the total charge at atomic site i and U is a scalar parameter typically chosen to be from 4 to 8eV for carbon systems [15, 16].

The TB Hamiltonian of Xu *et al.* [15] is an empirical TB model where the Hamiltonian matrix elements are parametrised as a function of distance between the atomic sites and atomic orbital type. Thus the different terms in the Hamiltonian can be calculated, knowing only the inter-atomic distances and direction cosines of the vectors between the atoms connected by hopping terms. The orbital products on the same site are trivial to calculate. The hopping terms lead to products of orbitals on neighbouring sites so to keep track of these terms we must have an enlarged neighbourhood list $NEIGH(j_s, i)$ which contains all the sites in a given localisation region plus the ones connected to it via the hopping terms. If the Hamiltonian operates on an orbital localised at site i it will produce a new orbital which is localised in the larger region defined by the sites $NEIGH(j_s, i)$. We define this new orbital to be

$$|\psi_i^m> = H|\phi_i^m> = \sum_{j \in LRH_i} \sum_k b_{im}^{jk} |\Phi_j^k>, \qquad (5)$$

where LRH_i is the localisation region defined by the neighbourhood list $NEIGH(j_s, i)$ in which b_{im}^{jk} is non-zero. In the program we precalculate $|\psi_i^m>$, then terms of the form $<\phi_i^m|H|\phi_j^n> = <\phi_i^m|\psi_j^n>$ can be calculated in the same way as S_{ij}^{mn} is calculated.

A third site index matrix $NEIGS(j_s, i)$ is required for the sparse coding of S and lists the pairs of orbitals having non-zero overlap i.e. the element $S(m, n, j_s, i)$ is the overlap between an orbital centered at site i and an orbital centered at site $NEIGS(j_s, i)$. Thus we have three index matrices $NEIG(j_s, i)$,$NEIGH(j_s, i)$ and $NEIGS(j_s, i)$ which define the structure of all the sparse data matrices in the program.

The minimisation of the energy functional of equation 3 was performed using the conjugate gradient method. In this method at each step t, the gradient, at the current position C_t, is calculated from

$$\begin{aligned} \frac{\delta E}{\delta \phi_i^m} &= 4 \sum_{j,n} [(H - \eta)|\phi_j^n> (2\delta_{ij}^{mn} - S_{ij}^{mn}) \\ &\quad - |\phi_j^n><\phi_j^n|H - \eta|\phi_i^m>] \qquad (6) \end{aligned}$$

and is used to construct the minimisation direction D_t, which is conjugate to all the former directions of search [17]. The energy is then minimised along the line $C = C_t + \lambda D_t$. The energy is a fourth order polynomial in λ (8th order if the Hubbard term is included) whose coefficients can be calculated explicitly by evaluating the energy functional in equation 3 at $C_t + \lambda D_t$. Once $\bar{\lambda}$, the value of λ at the minimum, has been determined the next point in the conjugate gradient process can then be calculated ($C_{t+1} = C_t + \bar{\lambda} D_t$). This whole process can be repeated until the required tolerance on the energy is reached.

The majority of calculations to evaluate the quartic polynomial and the derivative of the energy functional involve sparse matrix products. We will take as a typical example the calculation of S. As previously noted S is stored as a four dimensional array where the first two indices correspond to the orbitals centered on the same sites so that, if there was no sparse coding, S would be given by,

$$S(m, n, j, i) = \sum_l \sum_k C(k, m, l, j) * C(k, n, l, i) \qquad (7)$$

where $m = 1,3$ and $n = 1,3$ are the indices for the 3 orbitals on each site; $k = 1,4$ is the index for the basis set of four orbitals and l,i and j are site indices. The sum over l is only non-zero on the sites where the two orbitals overlap. $C(k, m, l, i)$ is sparse coded only on the l index while $S(m, n, j, i)$ is sparse coded only on the j index. Thus in the calculation of S the indices m, n, k and i are *direct indices* in the sense that they run over all their possible values while j and l are *indirect indices* running over only the non-zero values of the matrices. This makes our calculation different from standard sparse matrix multiplications which are typically of the form $Z_{ji} = \sum_k X_{jk} Y_{ki}$ where only i is a *direct index*. In our calculation we are essentially pulling in two 3×4 matrices ($C(k, m, l, i); k = 1,4; m = 1,3$) from indirectly indexed memory locations and multiplying them together to give a 3×3 matrix ($S(m, n, j, i); m = 1,3; n = 1,3$). Therefore in sparse notation S is given by,

$$\begin{aligned} S(m, n, j_s, i) &= \sum_{a,b;NEIG(a,j)=NEIG(b,i)} \sum_k \\ &\quad C(k, m, a, NEIGS(j_s, i)) * C(k, n, b, i), \qquad (8) \end{aligned}$$

where $j = NEIGS(j_s, i)$. At each MD step we precalculate index lists giving the couples of values, a and b, for which $NEIG(a, j) = NEIG(b, i)$ required for the outermost summation. Since we pull in small matrices rather than single values for each indirect memory reference these calculations run much faster on RISC type architectures with cache (such as the Cray T3D) than a standard sparse matrix multiplication. We will discuss this in more detail in the section on optimisation. The calculation of $< \phi_j^m|H|\phi_i^n >$ once we have constructed $H|\phi_i^n >$ is of the same form as the calculation of S given in equation 8. The calculation of the quartic polynomial for the line minimisation requires the further calculation of $< \phi_i^m|D_j^n >, < D_i^m|D_j^n >, < \phi_i^m|H|D_j^n >$ and $< D_i^m|H|D_j^n >$ where D is the conjugate gradient direction. All these calculations are of the same form as equation 8, the calculation of the overlap matrix. In total, for a typical run, about 65 % of the time in the CG step is taken up by sparse matrix multiplications while the remaining 35 % is mainly vector matrix products (to calculate the derivative) which are also sparse coded. To perform the MD step we must calculate the derivative with respect to ionic positions, of the total energy E (including the ionic potential and kinetic energy). E is given by

$$E(\{\phi_i^m\},\{R_I\}) = \frac{1}{2}\sum_I M\dot{R}_I^2 - E_{TB}(\{\phi_i^m\}) - \mu_{TB}(\{R_I\}), \tag{9}$$

where E_{TB} is the band structure energy which takes the form $2\sum_{i,j,m,n} Q_{ij}^{mn} < \phi_i^m|H|\phi_j^n >$; μ_{TB} is the interatomic potential of the TB model [15] and R_I is the atomic position of atom I. The forces on the ions are then given by

$$M_I\ddot{R}_I = -\frac{d\bar{E}(\{R_I\})}{dR_I}; \quad \bar{E} = min_{\{\phi_i^m\}}E(\{\phi_i^m\},\{R_I\}), \tag{10}$$

and the ionic positions are updated using the Verlet algorithm [18]. Calculating the contribution from the electronic band structure term E_{TB} takes the largest percentage of the time and, as in the CG step, this calculation mainly involves sparse matrix multiplications. As part of the MD step we must also update the index lists $NEIG(j_s,i)$,$NEIGH(j_s,i)$ and $NEIGS(j_s,i)$ since the sites within the different localisation regions change. The direction cosines which occur in the matrix elements of the Hamiltonian must also be updated [15]. In a typical MD run where 15 CG steps were performed for each MD step the CG steps took 85% to 90% of the total run time.

3 Molecular Dynamics on Parallel Computers

It may be thought that the localisation of the orbitals in our Quantum MD calculation would lead to a system which is more similar in nature, from a programming point of view, to short-range classical MD than standard Quantum MD simulations. In this section we will discuss some of the techniques used in programming classical MD with short-range interactions which are relevant to our localised quantum MD calculations and we will also point out some of the major differences between these two types of simulations.

In short-ranged classical MD simulations on parallel machines there are two main ways to distribute the data among the processors, often referred to as particle and spatial distribution. In the case of a spatial distribution the cell in which we carry out the simulation is divided spatially into blocks which are then allocated to different processors. Each processor then stores all the data associated with the particles within its block (which may change during the simulation) and calculates the trajectories for these particles. In the case of a particle type distribution all the particles are divided among the processors (typically each processor dealing with the same number of particles) and throughout the simulation each processor will store the data associated with its set of particles (which do not change) and calculate the trajectories for its set of particles. With this type of distribution, if the particles are moving rapidly during the simulation (e.g. in the case of a liquid or gas), there is no spatial correlation between the particles on the processors and their physical positions in the simulation. This means that when the data of a physically neighbouring particle is required there is a high probability it will not be on the same processor and not even on a processor which is physically close in the sense of the communication topology of the parallel computer. The particle data distribution can therefore lead to a high cost in communication between processors although it has the advantage that it is easy to load-balance. On the other hand the spatial decomposition method can lead to large load imbalances in the calculations on each processor if the relative number of particles in each spatial region varies greatly. However, due to the spatial correlation between the particles on the processors and the true physical positions of the particles in the simulation it is highly probable that the data of neighbouring particles required to update a given particle will be on the same processor or at worst a physically close neighbouring processor. Thus a spatial distribution of the particles tends to be very efficient from a communications point of view but bad from a load-balancing point of view. The best algorithms typically perform a combination of these two types of distribution giving roughly equal numbers of particles to each processor which have spatial locality and redistributing the particles to the processors, every few MD steps, if the particles move too much and lose there spatial locality. In this way both the constraints of load-balancing and minimising communication can be partially satisfied. It is an approach similar to this that we have followed for our Quantum MD program.

While in classical MD simulations particles are point like, in quantum MD simulations the wavefunctions or orbitals describing the electrons have a spatial extent. This means we can have a second approach to a spatial distribution of the data. In this case the system would be divided spatially but now each processor would deal with the calculations for the parts of the orbitals which lie in its spatial region. In the case of plane-wave codes the Fourier components of the plane-waves are also distributed over the processors. This is the most adaptable and commonly used approach for quantum systems where there is no localisation, and the orbitals are extended over the whole cell [19, 20]. It allows good load-balancing and minimises communication in the orthogonalisation step compared to a particle type distribution where complete orbitals are given to each processing element (PE). The localisation of the orbitals introduces the concept of closeness and spatial locality between the orbitals closer to classical systems where the particle data is, in a sense, localised at a point.

While we have drawn some analogies between classical short-range MD and quantum MD with localised orbitals there are several major differences which make our parallel implementation rather different from techniques used in classical MD. In particular, due to the optimisation of the electronic degrees of freedom, the amount of calculation required to update an atom at each MD step in quantum simulations is very much larger than the update of an atomic position for classical MD. This has implications for the load-balancing and also for the techniques used to calculate the neighbourhood lists.

In short-range classical MD the time taken to construct the neighbourhood lists; the particles within the cut-off radius of the interaction for each site, can become a significant percentage of the program time if not done efficiently. The simplest way to construct these lists is to calculate, for each particle, its distances from all the other particles and then compare them against the cut-off distance. This approach requires the calculation of distances between all possible pairs of particles and leads to an algorithm which scales as $O(N^2)$ which can quickly dominate in classical algorithms whose numerical calculations scale as $O(N)$. To obtain an $O(N)$ approach it is necessary to first bin the atoms into three dimensional cells of side r_c where r_c is the cut-off radius for the interaction. To find an atom's neighbours we then only have to search through 27 bins: the bin the atom is in and the 26 surrounding ones. Both the steps of binning and then searching the local areas for the neighbours are $O(N)$. This approach is often called the link-cell method [22, 23]. Another approach is to calculate an oversized neighbourhood list with cut-off $r_c + \delta r$ for each atom such that even after a given number of MD steps this list will still contain all the true neighbours ($r < r_c$), since there is a maximum distance an atom can move at each MD step [22, 23]. In this way we only have to search this oversized list for the true neighbours, at each MD step, and update the oversized list after a given number of MD steps. The most sophisticated algorithms are a combination of these approaches, using the link-cell binning every few MD steps to construct the oversized neighbourhood lists which are searched every MD step for the true neighbours [22, 23]. However, the optimal algorithm depends not only on the physical system under study but also on the architecture of the computer used as well as available memory.

In localised quantum MD, since the computational cost for each MD step is very large, we can only work with a relatively small number of atoms compared to classical short-range MD where simulations are performed on millions of atoms [23]. We have found that in our typical TB simulations, where we have 3000 to 4000 atoms, we can use the simple exhaustive search, $O(N^2)$ algorithm to calculate the neighbourhood lists. This approach only takes a few percent of the total computation time. Once we have more than 5000 atoms the $O(N^2)$ scaling can become important and for this size of system we used a link-cell method of the simplest kind where we performed the binning at each MD step. Again this was found to take a few percent of the program time and scaled in the same way as the calculations i.e. $O(N)$. Both of these algorithms were programmed in parallel with each processor having a complete list of all the atomic positions from which it calculated the neighbourhood lists for the sites it was allocated (see next section) and performed the binning procedure on all the sites. Since we are dealing with a relatively small number of sites we can store and communicate all the atomic positions to all processors but with classical MD simulations, which sometimes deal with millions of particles, this approach would be prohibitive in terms of memory and communication time. In a general sense the sophisticated techniques for calculating the neighbourhood lists used in classical MD are not necessary in Quantum MD with localised orbitals due to the computational intensity of the calculations at each MD step which dominates even inefficient methods for constructing the neighbourhood lists.

4 Parallel Tight-binding Code

In our program we used a particle/orbital distribution of the data among the processors so that each processor has complete orbitals. In the section on load-balancing we will discuss how we implemented spatial locality in the context of a particle like distribution for an inhomogeneous system. The reader may assume, in this section, that the code is written for a solid with periodic boundary conditions (such as bulk diamond) with the system being divided up into identical blocks, each processor dealing with the orbitals centered within this block and the atomic motion of the atoms in that block. We also assume the atoms never cross the boundaries of any of these blocks. For this system we have no load-balancing problems and a spatial locality of the mapping of the orbitals onto the processors. We chose a particle/orbital distribution (processors having complete orbitals) rather than a spatial type distribution (single orbitals being distributed between processors) for the following reasons.(i) It is the simplest to program as the parallel code loop structures remain essentially the same as the serial code with most of the subroutines remaining almost identical. (ii) It gives faster calculation speeds on RISC type chips with cache than other methods [25]. (iii) The communication cost of passing the orbital data between processors is modest due to the localised nature of the orbitals.

In our localised TB formulation the center of the localisation region is an atomic site so we can associate orbitals, as well as the atoms, to atomic sites. Therefore, each processor has a subset of atomic sites allocated to it and it will have stored in its memory all the data for the complete orbitals centered on that site plus all the atomic data for its sites. At each conjugate gradient step we require the values of the orbitals on other processors which are overlapping with the orbitals on our processor. We used a technique similar to ghost cells used in solving differential equations on parallel machines in that at each CG step each processor copies into a dummy array all the off-PE orbital elements that are required to update its own orbitals. Since during the CG steps the particles are not moving we can, at the start of each MD step, make a list of the processor numbers and memory locations of all the required off-PE orbitals. This list can then be used at each CG step to copy in the required data. It should be noted that during the conjugate gradient step, $H|\phi_i^m >$ plus the conjugate gradient directions $|D_i^m >$ must also be communicated between the processors. Our program is written in a message passing format using Fortran77 and PVM while the communication routines for the orbitals were written using the faster SHared MEMory (SHMEM) library routines which are native to the Cray T3D.

In addition to carrying all the local information for its set of particles and orbitals, each processor has global lists running over all the particles in the system which tell it which processor deals with a given particle and what is the local address of that particle. Using these global lists and its local neighbourhood lists each processor can determine the location of all its neighbouring orbitals. In addition each processor has a complete list of all the atomic sites so that the new neighbourhood lists can be calculated in parallel when the particles move.

Many of the matrices calculated in this program, such as the overlap matrix, are symmetric so we only need to calculate half the elements. This leads to a load imbalance problem if standard techniques are used. To overcome this problem we use a checkerboard type mask to distribute out the elements to the processors. This technique is very similar to the one used in classical MD calculations, for load-balancing the calculation of the anti-symmetric force matrix [22]. In general using a checkerboard mask to calculate non-sparse symmetric matrices on a parallel machine is very bad from the point of view of memory access since we are continually skipping through memory rather than going through with stride one. This greatly effects the MFLOP speed for RISC type chips with cache. Since our matrices are in fact sparse this is not an issue as we are already skipping through memory in a more or less random way so that to add another memory skip in the calculation makes very little difference to the MFLOP speed.

5 Optimisation

Sparse matrix multiplications can run extremely slowly on certain types of machine due to bad memory access so in this section we will briefly discuss how to program these types of calculations on machines such as the Cray T3D to get the best performance. Most modern parallel machines, such as the Cray T3D, are constructed using RISC type processors with a small, local fast memory (cache) and a slower, larger DRAM memory. RISC processors typically have very fast clock speeds and correspondingly high peak performance. The DEC (Digital Equipment Corporation) Alpha chip used in the Cray T3D has a clock frequency of 150 MHz and a peak speed of 150 MFLOPs but speeds close to this can only be achieved if the data can be read from memory quickly enough to feed the processor. If most data is being read directly from the cache rather than the slower DRAM memory it is possible to achieve high MFLOP speeds. A matrix multiply, such as the BLAS-3 routines, can run at over 100 MFLOPs since for $N \times N$ matrices each value is reused N times and therefore, it is typically in cache when required. Most calculations that do not significantly reuse the same values will run much slower. When we require a number from the DRAM memory a whole cache line (4×8 bytes for the Alpha chip, i.e. 4 double precision numbers) is loaded from the DRAM into the cache. It is then very important to use all or most of these values in the cache to obtain high MFLOP speeds.

Typically this means memory strides should be one, corresponding in a Fortran code to having the first index of the arrays striding by one in the innermost loops. It should be noted that this is very different from vector type machines (e.g. Cray YMP and C90) which do not have any cache and for which the MFLOP speed is much less sensitive to the size of the stride in loops.

In the case of sparse matrix multiplications we are typically reading each value from a random memory location so that we may have to read from DRAM each time we need a value and the 3 neighbouring values loaded into the cache are not used. What gives us reasonably high MFLOP speeds for our sparse matrix multiplications in the TB code is that each atomic orbital has 4 elements and we have typically 3 orbitals per site. Thus for each indirect memory address we will perform 3 DRAM memory reads each pulling in 4 values to the cache which are all used. It is therefore extremely important to have the correct order of the indices for the orbitals i.e. $C(k,m,l,i)$ where $k = 1,4$ for the basis set and $m = 1,3$ for the number of orbitals centered on each site. Any nested do loops should then be constructed so that the innermost loop strides through the k index then the m index so that one of the outer loops contains the indirectly addressed l index. In calculating the overlap matrix S we are therefore pulling in two 4×3 matrices into the cache and each of these values is reused 3 times in the matrix multiply so that for the 2nd and 3rd usage it will be read from cache rather than the DRAM memory. Using this data layout the calculation of S achieved about 30-33 MFLOPs[1] per processor depending on the physical system under study and the size of the localisation region. The other sparse matrix multiplications in the code such as the calculation of $< \phi_j^m | H | \phi_i^n >$ ran at similar speeds. Other calculations in the code such as vector matrix products ran a little slower at around 25 MFLOPs.

Another feature of our code, which exploits the fact that the data layout does not change during the CG steps, is the precalculation, at each MD step, of index lists for the memory location of the atomic orbital values required for calculating each element of S (see section 2). This avoids searching through the indirect indices of overlapping orbitals, at each CG step, to find the values corresponding to the atomic orbitals on the same site. This searching procedure can be quite costly since it involves many memory reads through indirect indices. Similar index lists can be calculated for the Hamiltonian and other orbital multiplications required in the CG step. A disadvantage of this approach is that if the localisation regions are very large these index lists can become very long and take up a significant percentage of the memory on each processor. These index lists were found to speed-up the calculation in our applications by about 20% to 30%.

[1] version 6.2.0.9 of the Cray Fortran77 compiler (cf77) was used with the options; aggress and readahead

6 Dynamic Load-Balancing

There are now many articles on different load-balancing schemes in short-range classical molecular dynamics applied to a range of different physical problems [22, 23, 24]. Load-balancing becomes an issue for systems where the amount of time required to calculate the new atomic positions varies from particle to particle. Giving out the same number of particles to each processor would then cause processors to idle while waiting for the processor with the most work to finish. As discussed in previous sections the main problem in developing a load-balancing algorithm to divide the calculations among the processors is to satisfy the, often conflicting, constraints of spatial locality of the data on the processors and load-balancing of the calculations between processors. If particles/orbitals are rapidly moving during the MD simulation any load-balancing algorithm which does not take into account spatial locality will lead to increased communications during the run. The most sophisticated load-balancing algorithms for classical MD simulations typically involve division of the system into a number of different sized regions equal to the number of processors such that the calculations, for all the particles in each region, take roughly the same time [21]. These regions should then have a surface area to volume ratio as low as possible in order to minimise the amount of communications. Algorithms to perform this type of load-balancing are often difficult to implement in three dimensions and can have numerical instabilities and bad convergence. In our load-balancing algorithm we will weakly relax the constraint of spatial locality while strongly satisfying the load-balancing constraint on the calculations. In our typical simulations the communications only take about 10% of the time so that even if the spatial locality was strongly satisfied we would only expect a few percent difference in the program run time between our solution and the optimal solution. On the other hand the calculations take about 90% of the run time therefore good load-balancing is extremely important.

The first step in our load-balancing algorithm is to spatially divide the system into three dimensional blocks. Typically the number of blocks is chosen to be equal to the number of processors although this is not a necessary condition and some systems may give better performance with other forms of spatial division. In our program we read in three parameters which correspond to the number of blocks in the x,y and z directions. We then construct a one dimensional list of all the sites by (i) ordering the blocks as a one dimensional list with x,y,z ordering and (ii) writing the sites in each block as a one dimensional list with z,y,x ordering. It is important that the ordering of the sites for each block is the reverse of the block ordering to give good spatial locality. In this way we construct a one dimensional list that has three dimensional spatial locality. The choice of the x coordinate in the physical system is, of course, arbitrary and can be chosen to given the best spatial mapping of the system onto the processors. We now wish to load-balance the list distributing sites to processors such that each processor does the same amount of work. In essentially all the subroutines in the program the outermost loop is over the sites allocated to each processor. It should be noted that in all our calculations we are only using two neighbourhood lists i.e. the one for the wavefunctions and the one for the Hamiltonian, and the relative amount of calculations between sites for the subroutines associated with these lists is essentially the same. In the program we time the calculation of S and $< \phi_j^m|H|\phi_i^n >$ associated with each site by adding a subroutine timing call within the loop over the sites. Averaging over these times for each site this gives a time t_i associated with the calculations for each site where $i = 1, 2, \ldots, N$, the order of sites being the one dimensional list we have constructed with spatial locality. We now calculate the time each processor would take to do the calculations if the system was perfectly load-balanced i.e.

$$T_{av} = \frac{\sum_i^N t_i}{n_{procs}}, \tag{11}$$

where n_{procs} is the number of processors. We then hand out contiguous strips of sites to each processor from our one dimensional list such that the sum of the times on each processor are as close as possible to T_{av}. In this way we achieve an essentially optimal load-balancing while still having a high degree of spatial locality of the mapping of the sites onto the processors. This means m sites, $j+1, j+2, \ldots, j+m$ are allocated to a given processor such that,

$$\sum_{i=j+1}^{j+m} t_i \simeq T_{av}. \tag{12}$$

An example of our load-balancing scheme applied to a two dimensional system is shown in figure one. It should be noted that sometimes in constructing the one dimensional list the communication cost may be reduced by exploiting the structure of the system. In a system of fullerenes we found a slight reduction in the communication cost by (i) ordering the complete fullerenes into a list with x,y,z ordering and (ii) each fullerene is written as a one dimensional list with approximately z,y,x ordering. Since the neighbourhood of each atom in the fullerene includes other atoms in the same fullerene the communications are

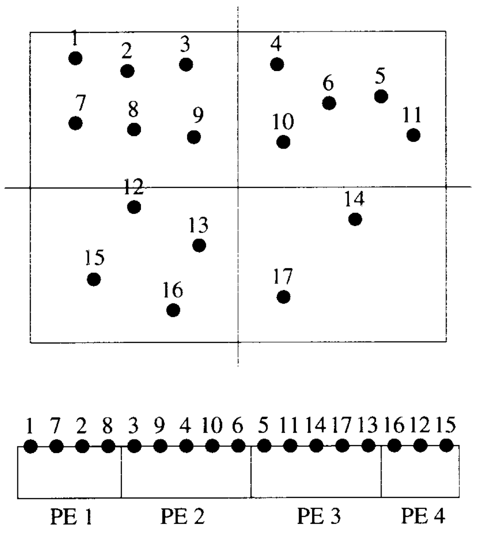

Figure 1: An example of a possible load-balancing scheme, for 4 processors. The two dimensional system of 17 atoms is divided up into 4 two dimensional blocks $B1$, $B2$, $B3$, $B4$. These blocks are then arranged in a one dimensional manner with the x coordinate of the block varying most rapidly. This means that if we associated coordinates with each block such that $B1$ is (1,1); $B2$ is (2,1); $B3$ is (1,2) and $B4$ is (2,2) then we order the blocks as (1,1),(2,1),(2,2),(1,2) i.e. $B1$,$B2$,$B4$,$B3$. In this example we have reversed the order for the second row i.e. $B1$,$B2$,$B4$,$B3$ and not $B1$,$B2$,$B3$,$B4$. This gives better spatial locality for non-periodic systems where blocks $B2$ and $B3$ are not spatially close but for periodic systems it is less necessary to do this. The atomic sites within each block are then written as a one dimensional list with the y index varying the most rapidly. In this way we construct a one dimensional list for the whole system (shown in the lower part of the diagram) which has two dimensional spatial locality. We then perform our load-balancing on this one dimensional list such that each processor has the same amount of work to do i.e. processor 1 works on sites 1,7,2,8,3,9 etc. We are assuming here that the amount of work to update each atom varies from atom to atom (see text for more details).

reduced slightly if our one dimensional list contains contiguous fullerenes rather than cutting them up with three dimensional blocks.

Providing the the ratio of the number of atoms to the number of processors is not too small we have found this load-balancing scheme to work extremely well. In a typical simulation where we had ~ 40 sites per processor we achieved load-balancing down to a few percent. We perform the load-balancing dynamically during our MD run every 10 to 20 MD steps depending on how rapidly the atoms are moving in our simulation. All the data for the orbitals and the atoms is reorganised in memory to correspond to the new mapping of sites to processors. Due to the computationally intensive nature of quantum MD the load-balancing typically takes less than one percent of the run time if it is carried out every 20 MD steps. This should be compared with classical MD where the load-balancing can easily become a significant percentage of the run time. It should be noted that our load-balancing algorithm is very general in that the same program can run any geometry of system, even systems of different dimensionality, on any configuration of processors.

7 Performance

In this section we will present performance results for an idealised bulk diamond system and for the inhomogeneous system of fullerenes deposited on a reconstructed diamond surface. The bulk diamond system used in the tests has periodic boundary conditions in the x,y and z directions. As mentioned in the section on the parallel tight-binding code each processor deals with a geometrically equivalent sub-block of the diamond system. During the MD runs on bulk diamond the neighbourhoods defining the localisation regions remained unchanged. The calculation times for constructing the index lists at each MD step are still included in the performance figures to make them more representative of an MD run on a real system in a research environment. Due to the static and homogeneous nature of

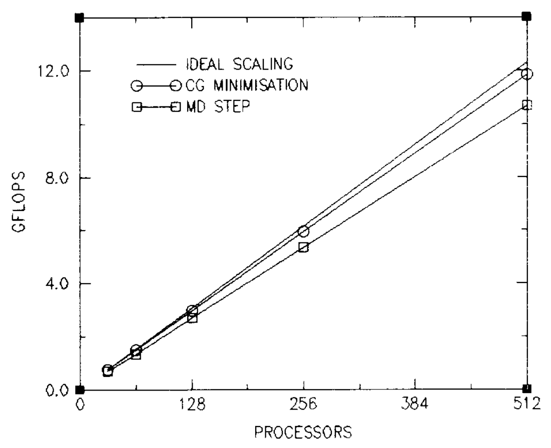

Figure 2: "Weak scaling" of the code for a bulk diamond system of 2048 to 32768 atoms running on 32 to 512 processors where the localisation region for each orbital contains 45 sites. The curve of circles shows the code performance for the conjugate gradient minimisation and the curve of squares shows the code performance for a full molecular dynamics run. The ideal scaling is shown for the CG curve.

this problem no load-balancing was required or performed during these runs. The inhomogeneous system used for these tests consisted of 52 fullerenes, each of 28 carbon atoms, deposited on a diamond slab (3072 atoms) consisting of 12 double layers with a reconstructed (111) surface and periodic boundary conditions in the x and y directions. The performance figures were taken for a typical dynamical run, with dynamic load-balancing every 20 MD steps, where 5 fullerenes were deposited on top of 47 fullerenes already bonded to the diamond slab.

In figure two we show the results for the "weak scaling" of our code where the size of the system is increased proportionally to the number of processors. The results shown are for a 2048 atom bulk diamond system on 32 processors up to a 32768 atom system on 512 processors. The localisation region is taken up to the 3rd nearest neighbour giving 45 atomic sites in each localisation region. This is a localisation region which is sufficiently large to give good convergence and very small errors (compared to a non-localised algorithm) for carbon insulators. The results for the MD runs are given with 15 CG steps per MD step which, in a former study, was found sufficient to achieve convergence [12, 16] for this type of system. As can be seen from the graph the speed-up is extremely linear and very close to ideal. The full MD simulation MFLOP speed is slightly lower than for the CG step due to the index lists that are constructed at each MD step. Also, some of the calculations for the atomic update run at a slightly lower MFLOP speed than the CG step. We present the results in GFLOPs rather than seconds per MD step as the smaller systems have wraparound effects on the localisation regions which means that proportionally less time is required for the MD step than larger systems. It should be noted that the algorithm used is strictly O(N) in terms of the number of calculations per MD step, so that provided there are no wraparound effects on the localisation regions the speed-up in GFLOPs translates directly to an identical increase in the number of atoms updated per second. All of these simulations spent about 14% of the time in communications and 86% doing calculations. An MD step, including 15 CG minimisation steps of the orbitals, took about 50s for the 32768 atom system on 512 processors running at above 11 GFLOPs. The weak scaling curves for other ratios of particles to processors and different localisation regions are also essentially linear with nearly ideal speedup. Due to the localised nature of the problem the communications remain local and the ratio of communications to calculations remains the same as we scale up the problem. The GFLOP speed for the inhomogeneous system on 128 processors is $\sim$ 8% lower than for the bulk diamond system. The lower GFLOP speed has different causes, the first of which is the barrier wait time introduced into the problem (even after load-balancing) by the inhomogeneous nature of the system and the motion of the particles. The second most important factor is the increase in communications due to the less spatially structured nature of the problem which makes the spatial mapping of the system to the processors less direct than the bulk diamond system. A third less important factor is the extra time required to run the

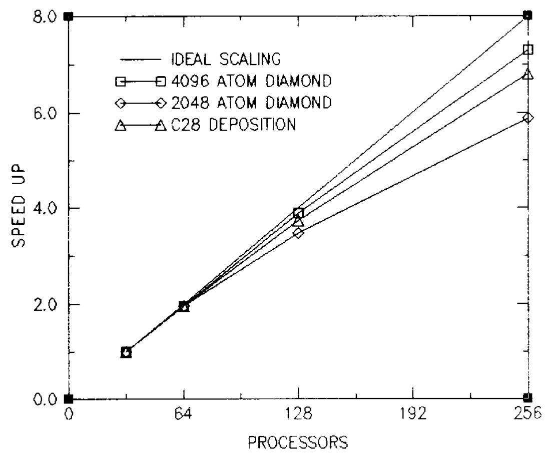

Figure 3: "Strong scaling" of the code for two bulk diamond systems of 2048 and 4096 atoms, and the inhomogeneous system of fullerenes deposited on a diamond slab. All orbital localisation regions are taken up to the third nearest neighbour.

load-balancing algorithm every 20 MD steps although this is only about 1% of the program time. Without dynamic load-balancing our test run is about twice as slow since when the fullerenes arrive at the surface the number of sites in the localisation regions of their atoms can increase by factors of 2 to 3.

These results should be compared with our vector code that runs at about 270 MFLOPs on one processor of a Cray C90 for the bulk diamond system. The number of floating point operations per MD step is, to within a few percent, the same for the vector and serial code. It should be noted that the structure of the loops in the vector code is very different from that of the parallel code. The innermost loop in the vector code must be over sites, since this is the only loop long enough to efficiently exploit the vector processing, while in the parallel code we parallelise over the site loop which must therefore be the outermost loop. Figure three shows the "strong scaling" performance for the code where the size of the system is kept constant and the number of processors is increased. Results are shown for 32 to 256 processors for the 2048 and 4096 atom bulk diamond system and also the inhomogeneous system. The choices of system are meant to be typical of the size of problem that would be run in a research environment on a computer of 32 to 128 processors. The speed-up is very linear and in the case of 4096 atoms of diamond there is a 7.3 speed-up in going from 32 to 256 processors. The speed-up is lower for the smaller 2048 atom system since, when we are in the regime where the number of atoms on each processor is very small (8 atoms per processor for 256 processors), the amount of time spent in communications becomes more important. This is because the fewer atomic sites we have on each processor the more probable it is that the orbitals, required to perform the calculation of the overlap matrix elements etc. for these sites, will not be on that processor and will have to be communicated. Typically the data will also have to be transmitted via more nodes of the communication network when there are fewer atoms on each processor thus the communication times will be larger. In order to reduce the communication cost of studying small systems on a large number of processors it may be more efficient to use a different data distribution where the data for each orbital is spatially distributed over the processors. However, it is not clear that the overall speed would be faster as the MFLOP speed for the calculations may decrease. This type of data distribution was implemented in reference [25] on a CM5 but the single PE performance is much lower than we obtained for our implementation. Load-balancing, while not discussed in this reference, may also become more difficult as it is not possible to simply load-balance over sites. This is because the calculations associated with the MD step for each individual atom are distributed over processors. Our production runs to date have never been in the regime where there were a small number of sites on each processor. This would only be useful for studying small systems over very long timescales.

8 Conclusions

In this paper we have presented a particle type data distribution scheme for the parallel implementation of an O(N)

TB molecular dynamics scheme on the Cray T3D. We have shown that the local nature of orbitals translates directly into a parallel algorithm that has only local communications between processors. While the amount of interprocessor communications in our program is significant the locality of these communications leads to an algorithm whose ratio of communications to particle number remains constant as we scale up the processor number and system size. We have also presented a simple and robust load-balancing scheme to handle inhomogeneous systems or systems with rapidly moving atoms. Parallel machines with fast communications, like the Cray T3D, are very cost-effective for these types of simulation, as well as allowing the possibility to scale to larger systems than could be studied on conventional supercomputing platforms (e.g. vector machines). Our program has now been used to study many different problems e.g. fullerene deposition, in a variety of carbon based systems. The results of these simulations will be published elsewhere [26].

9 Acknowledgements

This work was done as part of the PATP (Parallel Applications Technology Program) joint project between the EPFL and Cray Research. Support is also acknowledged from the Swiss National Science Foundation. In the course of this work we have benefited from useful discussions with other PATP collaborators; staff at Cray Research, Eagan, USA and C.F. Baillie at the University of Colorado.

References

[1] R. Car and M. Parrinello, Phys. Rev. 55 (1985) 2471.

[2] A. De Vita, G. Galli, A. Canning and R. Car, Nature, (to be published), (1996).

[3] G. Galli and M. Parrinello, Phys. Rev. Lett. 69 (1992) 3547.

[4] F. Mauri, G. Galli, and R. Car, Phys. Rev. B 47 (1993) 9973.

[5] W. L. Wang and M. Teter, Phys. Rev. B 46 (1992) 12798.

[6] W. Kohn, Chem. Phys. Lett. 208 (1993) 167.

[7] P. Ordejón, D. Drabold, M. Grunbach and R. Martin, Phys. Rev. B 48 (1993) 14646.

[8] S. Baroni and P. Giannozzi, Europhys. Lett. 17 (1991) 547.

[9] M. Aoki, Phys. Rev. Lett. 71 (1993) 3842.

[10] X.-P. Li, R. Nunes, and D. Vanderbilt, Phys. Rev. B 47 (1993) 10891.

[11] M. S. Daw, Phys. Rev. B 47 (1993) 10895.

[12] F. Mauri and G. Galli, Phys. Rev. B 50 (1994) 4316.

[13] J. Kim, F. Mauri and G. Galli, Phys. Rev. B 52 (1995) 1640.

[14] E. I. Blount, Solid State Physics, 13 (1962) 305.

[15] C. Xu, C. Wang, C. Chan and K. Ho, J. Phys.: Condens. Matter 4 (1992) 6047.

[16] G. Galli and F. Mauri, Phys. Rev. Lett. 73 (1994) 3471.

[17] W. H. Press, B. P. Flannery, S. A. Teukolsky and W. T. Vetterlin, *Numerical Recipes: The Art of Scientific Computing*, (Cambridge University Press, Cambridge, England), (1989) 301.

[18] L. Verlet, Phys. Rev. 159 (1967) 98.

[19] L. J. Clarke, I. Štich and M. C. Payne, Comp. Phys. Commun. 72 (1992) 14.

[20] A. Canning, A. De Vita, G. Galli, F. Gygi, F. Mauri and R. Car, Procs. of the 94 Fall, Cray User Group Conf. (Fine Point Editorial Services, Sheppherdstown, WV 25443), (1995) 18.

[21] Y. Deng, R. A. McCoy, R. B. Marr and R. F. Peierls, Procs. of the Seventh SIAM Conf., (edited by D. H. Bailey *et al.*), (1995) 605.

[22] S. J. Plimpton, J. Comp. Phys. 117 (1995) 1.

[23] D. C. Rapaport, Comp. Phys. Commun. 62 (1991) 198; D. C. Rapaport, Comp. Phys. Commun. 62 (1991) 217.

[24] W. Smith, Comp. Phys. Commun. 62 (1991) 229.

[25] S. Itoh, P. Ordejón and R. M. Martin, Comp. Phys. Commun. 88 (1995) 173.

[26] A. Canning, G. Galli, F. Mauri and R. Car, (In preparation).