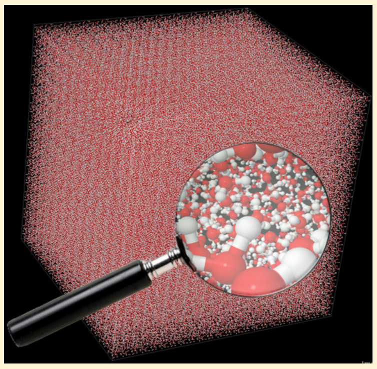
A Sparse Self-Consistent Field Algorithm and Its Parallel Implementation: Application to Density-Functional-Based Tight Binding

Anthony Scemama,[†] Nicolas Renon,[‡] and Mathias Rapacioli*,[†]

[†]Laboratoire de Chimie et Physique Quantiques, Université de Toulouse−CNRS−IRSAMC, 31062 Toulouse Cedex 04, France

[‡]CALMIP, Université de Toulouse−CNRS−INPT−INSA−UPS, F-31062 Toulouse Cedex 4, France

ABSTRACT: We present an algorithm and its parallel implementation for solving a self-consistent problem as encountered in Hartree−Fock or density functional theory. The algorithm takes advantage of the sparsity of matrices through the use of local molecular orbitals. The implementation allows one to exploit efficiently modern symmetric multiprocessing (SMP) computer architectures. As a first application, the algorithm is used within the density-functional-based tight binding method, for which most of the computational time is spent in the linear algebra routines (diagonalization of the Fock/Kohn−Sham matrix). We show that with this algorithm (i) single point calculations on very large systems (millions of atoms) can be performed on large SMP machines, (ii) calculations involving intermediate size systems (1000−100 000 atoms) are also strongly accelerated and can run efficiently on standard servers, and (iii) the error on the total energy due to the use of a cutoff in the molecular orbital coefficients can be controlled such that it remains smaller than the SCF convergence criterion.



1. INTRODUCTION

Density functional theory (DFT),[1] as routinely used with the practical formalism of Kohn and Sham,[2] is a very efficient approach to access physical and chemical properties for a large number of systems. Apart from the programs working with a grid to express the electron density, Kohn−Sham determinants are usually expressed on molecular orbitals (MOs) expanded on a basis set of N atomic orbitals (AOs), and the electronic problem is solved iteratively until a self-consistent solution is found. Technically, the naïve implementation involves the storage of several matrices containing $\mathcal{O}(N^2)$ elements and the computation of several diagonalization steps, each with a $\mathcal{O}(N^3)$ computational complexity. Hence, such an implementation of DFT can hardly address systems containing more than several hundreds of atoms with current computational facilities. However, the physical intuition tells us that it is possible to achieve linear scaling with the size of the system, as the interparticle interactions are essentially local.

In the past decades, there have been multiple developments of linear-scaling techniques, reducing both the required memory and the computational cost. Those imply inevitably the use of sparse matrices: matrices transformed such that almost all of the information is packed into a small number of matrix elements (the nonzero elements). The computational cost is also reduced, since operations involving zeros are avoided.

The electronic problem in DFT consists of minimizing the electronic energy with respect to the parameters of the density. These parameters can be expressed whether as molecular orbital coefficients or as density matrix elements. The density matrix has the property to be local, so the density matrix is naturally sparse, and this feature is exploited in density-matrix-based algorithms. Density matrix schemes usually rely either on a polynomial expansion of the density matrix, including several purification schemes,[3−11] or on the direct minimization of the energy with gradient or conjugated gradient schemes.[7,8,12−17] In these iterative approaches, a single step consists of performing a limited number of sparse matrix multiplications. Then, the use of sparse linear algebra libraries usually leads to linear scaling codes for very large systems. Among these schemes, the matrix sign function or the orbital free framework have recently reported linear scaling opening the route to the calculation of millions of atoms at the DFT level.[18,19] In schemes implying the expression of the MOs, the linear scaling regime can be reached by first localizing the MOs in the three-dimensional space and then using advanced diagonalization techniques, usually in a divide and conquer framework.[20,21]

The computational cost of the whole self-consistent field (SCF) process can be seen as the sum of the time spent in the building of the Fock/Kohn−Sham matrix and the time spent in the linear algebra, that is, the multiple matrix diagonalizations. In this work, we present a general algorithm to solve the linear algebra SCF problem by taking advantage of the sparsity of the systems and of the modern shared-memory symmetric multiprocessing (SMP) architectures. This algorithm is general and can be in principle applied to any single-determinant-based method (DFT, Hartree−Fock, and their semiempirical counterparts). We have implemented this algorithm in the deMon−

Received: February 12, 2014
Published: April 23, 2014

Nano code,[22] a density-functional-based tight binding (DFTB) program. DFTB[23−28] uses an approximate DFT scheme in which the elements of the Kohn−Sham matrix are computed efficiently from precalculated parameter tables. As in DFTB most of the computational time is spent on the SCF linear algebra, it is a good benchmark for the proposed implementation. In addition, DFTB is very well suited to linear scaling schemes, since it uses minimal and compact atomic basis sets, which are very favorable to having a very large fraction of zeros in the matrices. DFTB has already been improved in the literature by taking advantage of the sparsity of the matrices to compute properties[29] or by combining the approach with the divide and conquer scheme[27,30] or with the matrix sign function scheme to achieve very large scale computations.[19] Recently Giese et al. performed fast calculations on huge water clusters describing intramolecular interactions at the DFTB level and intermolecular interactions in a quantum mechanical scheme.[31]

Modern computers are very good at doing simple arithmetic operations (additions and multiplications) in the CPU core, but are much less efficient at moving data in the main memory. The efficiency of the data transfers is also not constant: when the distance in between two data access increases, the latency increases accordingly because of the multiple levels of cache and because the data may be located on a memory module attached to another CPU [cache-coherent non-uniform memory access (cc-NUMA) architecture]. Dense linear algebra can exploit well such architectures because the computational complexity of a matrix product or of a diagonalization is $\mathcal{O}(N^3)$, whereas the required storage is $\mathcal{O}(N^2)$: the cost of the data movement becomes negligible compared to the cost of the arithmetic operations and the processors are always fed with data quickly enough. In linear scaling methods, both the computational complexity and the data storage are $\mathcal{O}(N)$, and the arithmetic intensity (the number of operation per loaded or stored byte) is often not high enough to have CPU-bound implementations. These algorithms are then memory-bound. In addition, the data access patterns are also not as regular as in the dense matrix algorithms, where all the elements of the arrays can be explored contiguously. In that case, the hardware mechanisms that prefetch data into the caches to hide the large memory latencies will not work as efficiently, and these algorithms become *memory-latency-bound*. Moreover, the irregular access patterns may disable the possibility of using vector instructions, reducing the performance of arithmetic operations and data movement. As the efficiency of memory accesses decreases with the size of the data, the wall time curves are not expected to be linear in "linear-scaling" implementations. Linear wall time curves appear in two situations. The first situation is when the data access is always in the worst condition (with the largest possible latency). The second situation is when the cost of arithmetic operations is sufficiently high to dominate the wall time. This happens when the arithmetic intensity is high (the same data can be reused for a large number of operations) or when the involved arithmetic operations are expensive (trigonometric functions, exponentials, divisions, impossible to use vector instructions, etc.). Therefore, implementing an efficient linear scaling algorithm requires one to carefully investigate the data structures to optimize as much as possible the data transfers. The sole reduction of the number of arithmetic operations is not at all a guarantee to obtain a faster program: doing some additional operations can accelerate the program if it can cure a bad data access problem. A final remark is that linear scaling is *asymptotic*. In practice, we make simulations in a finite range, and it makes sense to find the fastest possible implementation in this particular range. For instance, if an $\mathcal{O}(N^2)$ algorithm is able to exploit the hardware such that the $\mathcal{O}(N)$ and $\mathcal{O}(N^2)$ curves intersect in a domain where N is much larger than the practical range, the quadratic algorithm is preferable. So, in this work, we did not focus on obtaining asymptotic linear wall time curves but we tried to obtain the fastest possible implementation for systems in the range of 10^3−10^6 atoms with controlled approximations.

After presenting the general algorithm, we give the technical implementation details that make the computations efficient for each step of the algorithm. Throughout the paper, the implementation will be illustrated by a benchmark composed of a set of boxes with an increasing number of water molecules.

2. ALGORITHM

The general algorithm consists of solving the self-consistent Roothaan equations by

(I) defining neighboring lists of atoms and MOs,
(II) proposing an initial guess of local MOs,
(III) othornormalizing the MOs,
(IV) computing the electronic density and the Fock/Kohn−Sham matrix, and
(V) partially diagonalizing the Fock/Kohn−Sham matrix in the MO basis.

Steps IV and V are iterated to obtain the self-consistent solution. Note that this algorithm is general: only step IV is specific to the underlying method (Hartree−Fock, DFT, and their semiempirical frameworks). In the following we define **H** as the Fock/Kohn−Sham operator expressed in the atomic basis set and **S** the overlap matrix of the atomic basis functions.

3. COMPUTATIONAL DETAILS

Due to the strongly sequential character of the algorithm, each step of the algorithm was parallelized independently using OpenMP[32] in an experimental version of the deMonNano package.[22] In recent years the number of compute nodes in x86 clusters has been roughly constant with an increasing number of CPU cores per node. In addition, the number of cores increases faster than the available memory, so the memory per CPU core decreases. This evolution of supercomputers is not well suited to implementations handling parallelism only with the message passing interface (MPI),[33] where each CPU core is running one MPI process. First, as the number of CPU cores per node increases, the number of MPI processes per node increases correspondingly, and the total required memory per node increases proportionally. Second, all the MPI processes running on one node will share the same network interfaces, and the bottleneck on the network will become more and more important, especially in collective communications. Hybrid parallel implementations usually combine distributed processes with threaded parallelism. The most popular framework is the combination of MPI for distributed parallelism and OpenMP for threaded parallelism. Implementations running one distributed process per node or socket where each process uses multiple threads are expected to be much more long-lasting. In this context, we chose to use a shared-memory approach for the calculation of the energy and leave the distributed aspect for coarser-grained parallelism, which will be investigated in a future work. Indeed, potential energy surfaces of large systems involve multiple local minima due to the large number of atomic degrees of freedom, and the chemical study of large systems implies the resolution of the Schrödinger equation at many different molecular geometries (geometry optimization, molecular dynamics, Monte Carlo, etc.) that may be distributed with a very low network communication overhead.

The systems chosen as a benchmark are cubic boxes of liquid water, containing from 184 to 504 896 molecules (from 552 to 1.5 million atoms). Running the benchmark consists of computing the single-point energy of each independent box at the DFTB level. All the reported timings correspond to the *wall time*. The calculations were performed on two different architectures. The first one is a standard dual socket Intel Xeon E5-2670 (each socket has 8 cores, 20 MiB cache, 2.60 GHz, 8.00 GT/s QPI) with hyperthreading and turbo activated: CPU frequencies were 3.3 GHz for 1−4 threads, 3.2 GHz for 8 threads, and 3.0 GHz for 16−32 threads. The second architecture is an SGI Altix UV 100 with 48 sockets (384 cores) and 3 TiB of memory. As the Altix UV 100 is based on the so-called cc-NUMA architecture, in our case 24 dual-socket blades are interconnected thanks to the SGI NUMAlink technology with a 2D-Torus topology. Hence this single system image (SSI) allows one to access the 3 TiB of memory as one single unified memory space. Each blade of the Altix machine is equipped with 128 GiB RAM and two Intel Xeon E7-8837 sockets (each one with 8 cores, 24 MiB cache, 2,67 GHz, 6.4 GT/s QPI). The hyperthreading and turbo features were disabled. The memory latencies measured on such a system are higher than the 80 ns measured on the dual-socket server: (i) 195 ns when the memory module is directly attached to the socket, (ii) 228 ns when accessing memory attached to the other socket of the blade through the Quick Path Interconnect link, and (iii) 570, 670, 760, 875, and 957 ns when accessing directly memory located in another blade through the NUMAlink interconnect (larger latencies correspond to an increasing number of jumps in the NUMAlink network before the target blade can be reached).

On the dual-socket architecture, the code was compiled with the Intel Fortran Compiler version 12.1.0 with the following options: -openmp -xAVX -opt-prefetch=4 -ftz -ip -mcmodel=-large -pc 64. The threads were pinned to CPU cores using the taskset tool. For jobs using less than 16 cores, the threads were scattered among the sockets: for instance, a 4 thread job was running 2 threads on each socket.

On the Altix-UV, the code was compiled with the Intel Fortran Compiler version 12.0.4 with the following options: -openmp -g -xSSE4.2 -opt-prefetch=4 -ftz -ip -mcmodel=large -pc 64. The threads were pinned to CPU cores using the omplace (SGI MPT) tool and runs were always performed such that the blades were fully active (16 threads running of 16 cores for each used blade).

4. IMPLEMENTATION

To make calculations feasible for large systems it is mandatory to reduce the storage of the matrices. Obviously, we use a sparse representation of the matrices by storing only the matrix elements with an absolute value above a given threshold. The choice of the sparse storage is the LIL scheme (Figure 1) discussed in section A of the Appendix.

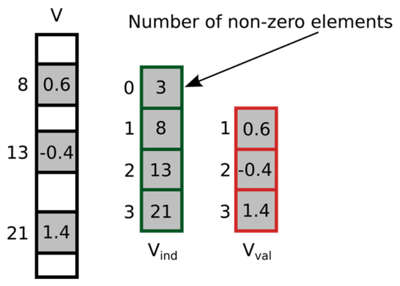


Figure 1. Sparse representation of a vector in the LIL format.

For the largest system presented here, a box made of 504 896 water molecules (1.5 million atoms), the maximum value of the total resident memory was 982 GiB, which corresponds to an average of ∼680 KiB per atom.

4.1. Defining Neighboring Lists of Atoms and MOs (Step I).

Two levels of localization or neighboring are used in this work. Two atoms are always considered as neighbors, unless all the off-diagonal elements of the overlap matrix **S** and of the Fock/Kohn−Sham matrix **H** involving one AO centered on each atom are zero. At the beginning of the program, an atomic neighboring map is built and is used to avoid computing zero elements of **S** and **H** in the AO basis.

It is also convenient to use groups of localized MOs. This is done by generating an initial guess of MOs (see below) with nonzero coefficients on a limited number of AOs centered on atoms in the same region of space. During the orthonormalization and SCF process, MOs belonging to one group can of course delocalize and have coefficients on AOs centered on atoms belonging to another group. We however keep the initial group attribution of MOs during the whole SCF process, as the MOs are not expected to delocalize sufficiently to change the MO group attribution. Two groups are considered as neighbors if there exists a distance between one atom of the first group and an atom of the second one which is smaller than a given value (in practice we use 20 b for DFTB calculations, corresponding to twice the cutoff of the matrix elements). A map of MO group neighbors is built and we will assume that, in the MO basis, blocks of **H** and **S** involving non-neighboring groups will always be zero. This assumption relies on the fact that as both occupied and virtual MOs are localized they are not expected to delocalize by more than 20 b during the SCF process. This assumption can be checked after the SCF has converged and must not be misinterpreted as a screening of long-range $1/R$ interactions.

We now describe the procedure to build MO groups. Atomic orbitals centered on atoms that are geometrically far from each other have a zero overlap, and do not interact through the Fock/Kohn−Sham operator. All the atoms of the system are reordered such that atoms close to each other in the three-dimensional space are also close in the list of atoms. To achieve this goal, we use a constrained version of the *k-means* clustering algorithm.[34]

A set of m centers (*means*) is first distributed evenly in the three-dimensional space. Every atom is linked to its closest center with the constraint that a center must be linked to no more than N_{link} atoms: If an atom cannot be linked to a center because the latter already has N_{link} linked atoms, the next closest center is chosen until an available center is found. Each center is then moved to the centroid of its connected atoms and the procedure is iterated until the average displacement of the centers is below a given threshold or until a maximum number of iterations is reached (typically 100). Finally, the new list of atoms is built as the concatenation of the lists of atoms linked to each center.

Because of the constraint to have no more than N_{link} atoms attached to one center, the atom-to-center attribution depends on the order in which the atom list is explored and multiple solutions exist. This has no effect in the final energy since the definition of the neighbors depends on the atom−atom distances. However, it can have an impact on the computational time as the lists of neighbors can have different lengths. To cure this problem, the list of centers is randomly shuffled at every k-means iteration.

Each k-means attribution step is parallelized over the atoms. The N_{link} constraint introduces a coupling between the threads that is handled using an array of OpenMP locks: a lock is associated with each center, and the lock is taken by the thread before modifying its list of atoms and then released. The displacement of the centers is trivially parallelized over the centers, and this initialization step takes 2−5% of the total wall time.

4.2. Initial Guess of Molecular Orbitals (Step II). For each molecule of the system, a non-self-consistent calculation is performed. The molecular orbitals (MO) matrix **C** in the atomic orbitals (AO) basis set is built from the concatenation of all the lists of occupied MOs of each group and all the lists of virtual MOs of each group.

These MOs are not orthonormal but have the following properties:

(i) MOs are local; they have nonzero coefficients only on the basis functions that belong to the same molecule.

(ii) MOs belonging to the same molecule are orthonormal.

(iii) MOs (i, j) belonging to MO groups that are not neighbors have a zero overlap and do not interact; $(\mathbf{C}^{\dagger}\mathbf{SC})_{ij} = (\mathbf{C}^{\dagger}\mathbf{HC})_{ij} = 0$.

If the MOs do not delocalize by more than 20 b, which was the case on all our benchmarks, the large off-diagonal blocks of zeros corresponding to the last point persist in the orthonormalization and SCF processes, so **C** always stays sparse.

The non-SCC calculations are parallelized over the molecules using a dynamic scheduling to keep a good load-balancing, and this initialization step takes less than 1% of the total wall time.

4.3. Orthonormalization of the Molecular Orbitals (Step III). The MOs are orthonormalized by diagonalizing iteratively the $\mathbf{C}^{\dagger}\mathbf{SC}$ matrix. At this point, **C** is already stored in a sparse format.

(1) **S** is computed (or read from memory) only in elements involving AOs of atomic neighbors.

(2) $\mathbf{C}^{\dagger}\mathbf{SC}$ is computed.

(3) The MOs are normalized using diagonal elements of $\mathbf{C}^{\dagger}\mathbf{SC}$.

(4) Combinations (approximate rotations) are performed between the pairs of MOs (i, j) that involve the largest off-diagonal elements of the MO overlap matrix:

$$C'_{ki} = C_{ki} - \frac{C_{kj}(\mathbf{C}^{\dagger}\mathbf{SC})_{ij}}{2}$$

$$C'_{kj} = C_{kj} - \frac{C_{ki}(\mathbf{C}^{\dagger}\mathbf{SC})_{ij}}{2}$$

(5) Return to step 3 until the largest off-diagonal element is below a given threshold.

Note that, if the computational cost of building of the overlap matrix elements is comparable to reading it from the memory, it is suitable to compute the overlap matrix elements only when they are needed in the computation of $\mathbf{C}^{\dagger}\mathbf{SC}$ to reduce the amount of storage.

The parallelization of the MO combinations (step 4) is not trivial. If one combination of MOs (i, j) is being performed by one thread, any combination of pairs of MOs involving either i or j cannot be performed simultaneously by another thread. This is achieved by the use of OpenMP locks, as detailed in the section B of the Appendix.

The orthonormalization takes 40−50% of the total wall time. Half of the time is spent in the MO rotations and the other half is spent in the computation of $\mathbf{C}^{\dagger}\mathbf{SC}$, which is discussed in detail below.

4.4. Partial Diagonalization of the Fock/Kohn−Sham Matrix (Step V). The wave function is invariant with respect to unitary transformations among doubly occupied orbitals or among virtual orbitals. Therefore, the full diagonalization of **H** is not necessary to obtain the SCF minimum energy and properties. It is sufficient to annihilate only the matrix elements of the Fock/Kohn−Sham matrix expressed in the MO basis involving one occupied and one virtual orbital. To do that, we use an approach similar to the orthonormalization procedure: we apply Jacobi rotations between the occupied and virtual orbitals, as proposed by Stewart.[21] To preserve the orthonormality of the MOs, the orbitals are combined using an accurate rotation but with an approximate angle.

(1) Compute **H** only in elements involving AOs of neighboring atoms.

(2) Compute $\mathbf{C}^{\dagger}\mathbf{HC}$ as explained in next section.

(3) Rotate the pair of MOs (i, j) that involves the largest off-diagonal element:

$$p = \frac{(\mathbf{C}^{\dagger}\mathbf{HC})_{ii} - (\mathbf{C}^{\dagger}\mathbf{HC})_{jj}}{2(\mathbf{C}^{\dagger}\mathbf{HC})_{ij}}$$

$$t = \left(p + \text{sgn}(p)\sqrt{1 + p^2}\right)^{-1}$$

$$c = (1 + t^2)^{-1/2}$$

$$s = tc$$

$$C'_{ki} = cC_{ki} + sC_{kj}$$

$$C'_{kj} = -sC_{kj} + cC_{ki}$$

(4) Return to step 3 until the largest off-diagonal element is below a given threshold or if the number of rotations is equal to 10 times the number of occupied MOs.

As mentioned for the orthonormalization, it can be suitable to compute the elements of the Fock/Kohn−Sham matrix directly when needed at the second step to avoid the storage of a matrix.

Avoiding the performance of occupied−occupied and virtual−virtual rotations keeps both the occupied and virtual MOs localized during the optimization.[35,36] In this way, the **C** matrix stays sparse. The $\mathbf{C}^{\dagger}\mathbf{HC}$ matrix is displayed in Figure 2. As the matrix is symmetric and only the occupied−virtual rotations are considered, only one occupied−virtual block is computed for the orbital rotations, appearing in white on the figure. This block can be divided in sub-blocks using the MO groups definition.

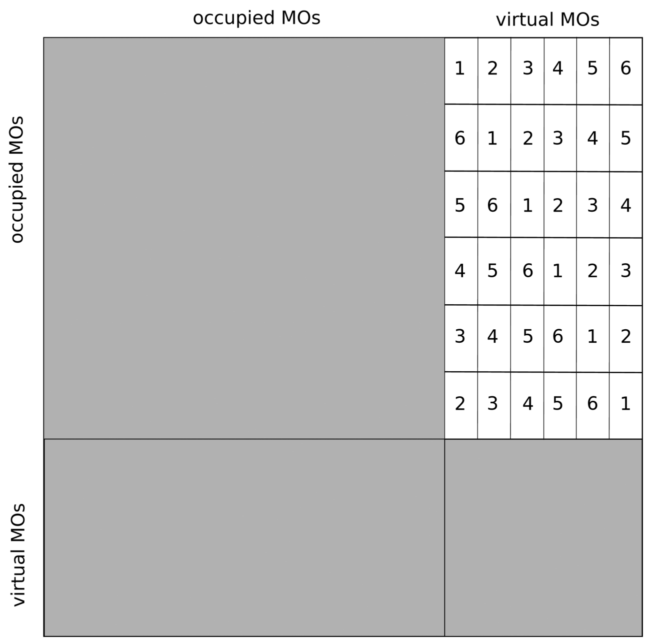


Figure 2. Representation of the $\mathbf{C}^{\dagger}\mathbf{HC}$ matrix. Only one off-diagonal block is annihilated by pseudo-Jacobi rotations. The numbers identify iterations. On the first iteration, blocks labeled 1 are annihilated in parallel, etc.

The parallelization of the rotations is implemented such that one orbital can never be rotated simultaneously by two threads: each thread performs a rotation in a block of the occupied−virtual elements. The figure shows the outer loop iterations. On the first iteration, the threads perform rotations in all the blocks labeled 1. After a synchronization, on the second iteration they perform rotations in the blocks labeled 2, etc. Blocks between non-neighboring MO groups are not considered for the rotation process. Let us mention that in practice the occupied−virtual $\mathbf{C}^{\dagger}\mathbf{HC}$ block is computed and stored in a sparse format.

The time spent in the MO rotations represents 3−6% of the total wall time. The bottleneck is the calculation of $\mathbf{C}^{\dagger}\mathbf{HC}$. Adding the timing of $\mathbf{C}^{\dagger}\mathbf{HC}$ yields 33−40% of the total wall time.

4.5. Computation Kernel for the Matrix Products $\mathbf{C}^{\dagger}\mathbf{XC}$ (Steps III and V). From the two previous sections, it appears that the main hot spot is the computation of double sparse matrix products of the form $\mathbf{C}^{\dagger}\mathbf{XC}$, where $\mathbf{X} = \mathbf{S}$ for the orthonormalization part and $\mathbf{X} = \mathbf{H}$ for the diagonalization part. We explain here how this step is performed by taking advantage of the sparsity of matrices $\mathbf{C}$, $\mathbf{H}$, and $\mathbf{S}$.

In another work, an efficient dense × sparse matrix product subroutine for small matrices was implemented in the QMC=Chem quantum Monte Carlo software.[37] These matrix products are hand-tuned for Intel Sandy-Bridge and Ivy-Bridge microarchitectures and have reached more than 60% of the peak performance of a CPU core. To obtain such results, some programming constraints are necessary and are detailed in the next section.

In what follows, large sparse × sparse matrix products are performed as a collection of small dense × sparse products avoiding the zero-blocks connecting non-neighboring MO groups.

4.5.1. Small Dense × Sparse Matrix Products. Let us recall the dense × sparse matrix products of ref 37. In the multiplication routine, the loops are reordered such that vectorization is possible, giving a dense matrix in output (see Figure 3). As the innermost loop has a small number of

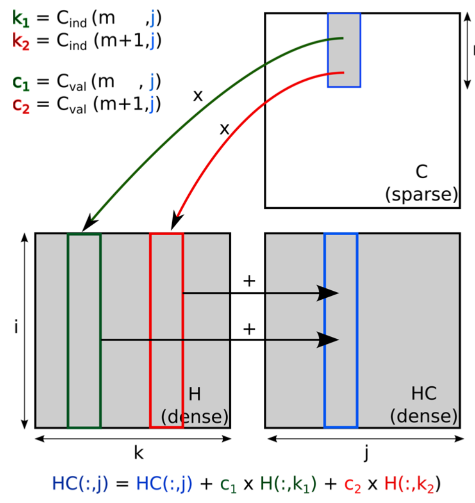


Figure 3. Dense × sparse matrix product.

iterations, the following constraints are mandatory to obtain performance: (i) all arrays have to be 32-byte aligned using compiler directives, and (ii) the leading dimensions of arrays should be multiples of 8 elements such that all the columns are 32-byte aligned.

Without these constraints, the compiler will generate both the scalar and the vector version of the loop, and the scalar version will be used as long as the 32-byte alignment is not reached (peeling loop) or if a register of 4 double precision or 8 single precision elements cannot be filled (tail loop). In such a case, a 16-iteration loop in single precision will perform 8 scalar loop cycles and one vector loop cycle. Using the alignment and padding constraints, the compiler can produce 100% vector instructions with no branching and will execute two vector loop cycles unrolled by the compiler, resulting in a speedup of more than five. Note that these constraints contribute to the choice of the memory layout for sparse matrices explained previously.

4.5.2. Computation of $\mathbf{C}^{\dagger}\mathbf{XC}$. For each thread, the algorithm is the following:

(1) Compute 16 contiguous columns of $\mathbf{XC}$ in a dense representation.

(2) Transpose the result to obtain 16 rows of $\mathbf{C}^{\dagger}\mathbf{X}$.

(3) Multiply on the right by $\mathbf{C}$ to obtain 16 rows of $\mathbf{C}^{\dagger}\mathbf{XC}$.

(4) As $\mathbf{C}^{\dagger}\mathbf{XC}$ is symmetric, transpose the dense 16 rows of $\mathbf{C}^{\dagger}\mathbf{XC}$ into a sparse representation of 16 columns of $\mathbf{C}^{\dagger}\mathbf{XC}$.

(5) Iterate until all the columns of $\mathbf{C}^{\dagger}\mathbf{XC}$ are built.

In step 1, dense × sparse matrix products are performed using only the 16 × 16 blocks of $\mathbf{X}$ containing at least one nonzero element. This exploits the sparse character of $\mathbf{X}$ by avoiding large blocks of zeros using the atomic neighbor list. The 16 × 16 shape of the arrays is constant and known at compile time, and all the columns of the 16 × 16 arrays are 32-byte aligned. This permits the compiler to remove the peeling and tail loops and generate a fully vectorized code.

The 16 dense columns of $\mathbf{XC}$ are not stored as a $(N_{\text{basis}},16)$-array, but are instead represented as an array of 16 × 16 matrices: the dimension of the array is $(16,16\lfloor N_{\text{basis}}/16\rfloor + 1)$. The reason for this layout is 2-fold. First, the result of $\mathbf{XC}$ is written linearly into memory. The spatial locality in the cache is improved and this reduces the traffic between the caches and the main memory. Second, step 2 can be performed very fast since the transposition of **XC** consists of $\lfloor N_{\text{basis}}/16 \rfloor + 1$ in-place transpositions of contiguous 16 × 16 matrices. Using the (N_{basis},16) layout, the transposition would involve memory accesses distant by N_{basis} elements suffering from a high memory latency overhead for large systems, since the hardware prefetchers prefetch data only up to a memory page boundary. Using our layout, the transposition is always memory-bandwidth bound, as each 16 × 16 transposition occurs in the L1 cache and the next 16 × 16 matrix is automatically prefetched.

In step 3, the dense × sparse product is performed using only the columns of **C** belonging to MO groups that are neighbors of the MO groups to which the 16 rows of $\mathbf{C}^\dagger\mathbf{X}$ belong. Again, the loop count of the innermost loop is 16 and this part is fully vectorized as all the columns of the arrays are properly aligned.

4.5.3. Parallel Efficiency. The parallel speedup curve of the $\mathbf{C}^\dagger\mathbf{XC}$ products measured on the dual-socket server is shown in Figure 4. For small sizes, the speedup is nearly optimal (the perfect value of 16 obtained here is fortunate, since measuring very small wall times contains large uncertainties). This can be explained by the fact that most of the matrices can fit in the level 3 cache. Then, as the size of the **C** matrix increases, more and more pressure is made between the level 3 cache and the memory, as every core has to read the whole **C** matrix when multiplying the 16-rows with **C**. The latency of data access to **C** increases accordingly and the processor cores cannot be kept busy. This is confirmed by the fact that hyperthreading improves the parallel efficiency by giving some work to a core for one thread while another thread is stalled. The speed-up converges to a limit of 11 for a 16-core machine. The wall time curve of Figure 5 shows a scaling with the number of atoms of $O(N^{1.47})$. The scaling is not linear, because of the nonuniformity of the memory latencies.

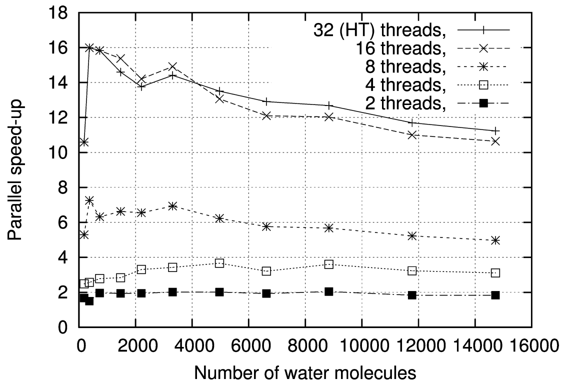


Figure 4. Parallel speedup of $\mathbf{C}^\dagger\mathbf{XC}$ as a function of the number of water molecules obtained on the dual-socket server. The speedups were renormalized to remove the effect of the turbo feature. The 32-thread results were obtained using hyperthreading (2 threads per physical core).

4.6. Specific Implementation for DFTB. We now turn to the specific implementation of the previously described algorithm for DFTB, an approximate DFT scheme.[23−28,38] The electronic problem is only solved for valence electrons, and molecular orbitals are expressed in a minimal atomic basis set. The DFTB Kohn−Sham (KS) matrix is

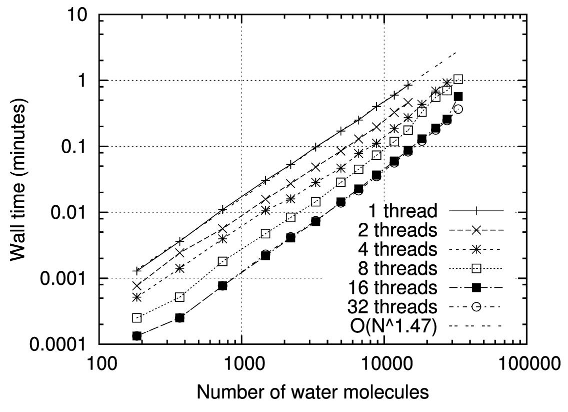


Figure 5. Wall time of $\mathbf{C}^\dagger\mathbf{XC}$ as a function of the number of water molecules obtained on the dual-socket server. The 32-thread results were obtained using hyperthreading (2 threads per core).

$$H_{\mu\nu} = H^0_{\mu\nu} + \frac{1}{2} S_{\mu\nu} \sum_{\xi} (\gamma_{\alpha\xi} + \gamma_{\beta\xi}) \Delta q_{\xi} \tag{1}$$

$H^0_{\mu\nu}$ is the KS matrix element between orbitals μ of atom α and ν of atom β at a reference density. This value is usually interpolated from precalculated DFT curves (the so-called Slater−Koster tables[39]). With the set of parameters used for the benchmarks (mio-set[25,40]), the Hamiltonian and overlap matrix elements are zero for interatomic distances larger than 10 b. Notice that it corresponds to half of the distance used to define MO neighboring blocks. One-site $\gamma_{\alpha\alpha}$ are obtained from Hubbard parameters and interatomic $\gamma_{\alpha\xi}$ depend on the distances between atoms α and ξ and contain the long-range $1/R$ Coulomb interaction, which is not screened. $\Delta q_\xi = q_\xi - q^0_\xi$ is the charge fluctuation of atom ξ calculated with the Mulliken analysis scheme:

$$q_{\xi} = \sum_{i}^{N_{\text{MOs}}} n_i \sum_{\nu}^{N_{\text{AOs}}} \sum_{\mu \in \xi} C_{\mu i} C_{\nu i} S_{\mu\nu} = \sum_{\nu}^{N_{\text{AOs}}} \sum_{\mu \in \xi} P_{\mu\nu} S_{\mu\nu} \tag{2}$$

where **P** is the density matrix in the AO basis set.

Computing the atomic Mulliken population of eq 2 could be an expensive part that has to be efficiently parallelized. The parallelization can be done either on the MOs (one thread deals with a subset of MOs and computes their contributions to the atomic populations of the whole system) or on the atomic population (one thread computes the atomic population of some atoms arising from all the MOs). The first scheme is limited by storing data, whereas the second one is limited by reading data. Storing data in memory is more costly than reading data, so the second scheme was chosen. An acceleration can be achieved by considering the spatial localization of molecular orbitals (only a limited number of MOs contribute to the population of a given atom). For each atom α we dress the list of the indexes of MOs having a nonzero contribution on atom α. Then, we compute a set of rows of the density matrix $P_{\mu\in\alpha,\nu} = \sum_i C_{\mu i} C_{\nu i}$, where ν is such that $S_{\mu,\nu} \neq 0$ (using the atomic neighbors list), and the sum over MOs i only runs on the MOs listed as contributors on atom α. q_ξ is then computed from eq 2. The calculation of the charges q_ξ takes typically 3% of the total wall time, and the observed scaling is $O(N^{1.1})$. Note that we limit the number of threads to a maximum of 16 in this part. Indeed, this section is extremely sensitive to memory latency, as the arithmetic intensity is very low. Using more than 16 threads would necessitate interblade communications on the

Altix UV, which always hurt the performance for this particular section.

From the computational point of view, it is better to compute first a vector $Q_\alpha = \sum_\xi \gamma_{\alpha\xi}\Delta q_\xi$, to rewrite eq 2 as

$$H_{\mu\nu} = H^0_{\mu\nu} + \frac{1}{2}S_{\mu\nu}(Q_\alpha + Q_\beta) \tag{3}$$

The computation of the Q_α values requires $O(N^2)$ operations. This term contains actually the long-range $1/R$ contributions to the Coulomb energy, and a linear scaling on this part could only be reached at the price of approximations. We decided not to screen these long-range contributions in order to keep the same results as those one would obtain from a standard diagonalization scheme, the computational cost remaining acceptable (less than 20% of the total wall time for the largest systems of our benchmarks).

5. RESULTS OBTAINED ON THE WHOLE SCHEME

5.1. Sparsity and Numerical Accuracy.

In the deMon-Nano program, the convergence criterion of the SCF procedure is based on the largest fluctuation of the Mulliken charge:

$$\delta q_{\max} = \max_i |q_i^{(n)} - q_i^{(n-1)}| \tag{4}$$

where $q_i^{(n)}$ denotes the Mulliken charge of atom i at SCF iteration n. As this criterion is local, it ensures that all the different regions of the system have converged to a desired quality, as opposed to a criterion based on the total energy of the system, which is global. We define the *sparse threshold* ε as a quantity used to adjust the sparsity of the MO matrices. ε is set 10 times lower than the target SCF convergence criterion $\delta q_{\max}$, such that if more precision is requested for the SCF, the matrices become less and less sparse. In the limit where the requested SCF convergence threshold is zero, the matrices are dense and the exact result is recovered.

Once the initial guess is done, the initial MOs are improved by running an SCF with very loose thresholds: $\varepsilon \leq 10^{-3}$ and $\delta q_{\max} \leq 10^{-1}$ a.u. The calculation is then started using the thresholds given by the user in input. In what follows, the thresholds were $\varepsilon \leq 10^{-6}$ and $\delta q_{\max} \leq 10^{-5}$ a.u. Typically 8 SCF iterations were needed to reach a convergence of $\delta q_{\max} \leq 10^{-1}$ a.u., and 9−13 additional iterations were necessary for $\delta q_{\max} \leq 10^{-5}$ a.u.

The orthonormalization needs to be very accurate, so the whole orthonormalization procedure is run in double precision. After each rotation of pairs of MOs, the MO coefficients $\mathbf{C}_{ki}$ such that $|\mathbf{C}_{ki}| < 10^{-3}\varepsilon$ are set to zero. This cutoff is chosen to be much smaller than ε to ensure that the quality of the orthonormalization will be sufficient. However, this leads to an increase of the number of nonzero elements in $\mathbf{C}$ (see Figure 6). The orthonormalization is considered converged when the absolute value of the largest off-diagonal element of $\mathbf{C}^\dagger\mathbf{SC}$ is lower than $\varepsilon/100$.

The partial diagonalization of $\mathbf{H}$ does not need to be as precise as the orthonormalization, so single precision is used to accelerate the calculation of $\mathbf{C}^\dagger\mathbf{HC}$. A threshold of ε is applied to the matrix elements of $\mathbf{H}$ and to the elements of $\mathbf{C}$ after a rotation of MOs. This larger cutoff increases the sparsity of $\mathbf{C}$ but degrades the orthonormality of the MOs. To cure this latter problem, an additional orthonormalization step among only the occupied MOs is performed after the SCF has converged.

The total energies obtained with the divide and conquer LAPACK routine DSYGVD are compared to our implementation in Table 1 with different SCF convergence criteria. One can first remark that as the convergence criterion of the SCF procedure gets lower, the error due to our implementation decreases as ε decreases accordingly. Moreover, the difference in total energies between the reference LAPACK result and our implementation is orders of magnitude below the error due to the lack of convergence of the SCF procedure. The absolute error per water molecule is below 2×10^{-10} using $\delta q_{\max} = 10^{-5}$ and around 3×10^{-12} for $\delta q_{\max} = 10^{-6}$.

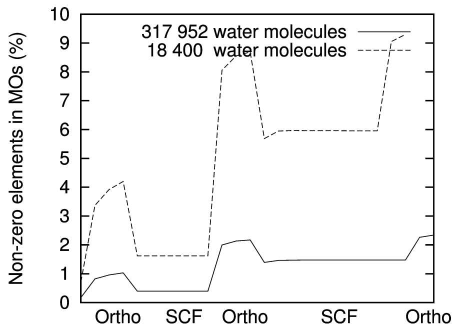


Figure 6. Percentage of nonzero elements during the execution of the program.

5.2. Parallel Efficiency and Scaling.

In this section, the time values correspond to the timing of the *whole* program, as obtained by the time UNIX tool.

Figure 7 shows the parallel speedup of the program on the dual-socket server. When using all the 16 available physical cores of the machine, the parallel efficiency converges quickly to a value between 11 and 12. One can remark that hyperthreading improves the parallel efficiency. Indeed, as the memory latencies are the bottleneck, the CPU core can be kept busy for one thread while another thread waits for data. On the Altix UV (Figure 8), the speedup is presented as a function of the number of 16-core blades. Indeed, the important NUMA effects show up when using more than one blade, since the interblade memory latency is significantly higher than the intrablade latency by a factor between 2 and 5. This explains why the parallel efficiency is disappointing on the Altix UV: a speedup close to 2 is obtained with 8 blades for small systems. However, when the systems is so large that the total memory cannot fit on a single blade, the 16-core reference suffers from a degradation of performance due to the NUMA effects, so the comparison becomes more fair between the single-blade and the 8-blade runs, and the parallel speedup is much more acceptable, around 5.

From Figures 9 and 10, the scaling of the program as a function of the number of atoms is $O(N^{1.7})$ on the dual-socket server and $O(N^{1.8})$ on the Altix UV. The global scaling is not linear. This is first due to the nonuniformity of the memory access: the code is faster with smaller memory footprints, since the latencies are smaller. Second, in the expression of the Fock matrix elements (eqs 2 and 3), for each atom one has to compute the Coulomb contributions with all the other atoms. We chose not to truncate the $1/R$ function in order to keep a good error control on the final energies, and this results in a quadratic scaling with the number of atoms appearing for large systems (10% of the total time for 33 120 molecules on the dual-socket server, and 18% for 504 896 molecules on the Altix UV).

The CPU time per atom is given in Figure 11. A similar benchmark was presented with a linear-scaling approach in the

Table 1. Total Energies (in a.u.) Obtained with the Full Diagonalization Using the Intel MKL Implementation of the LAPACK Library[41] (E_{LAPACK}) or the Implementation Proposed in This Work (E)

N[a]	E_{LAPACK}	E	$(E - E_{\mathrm{LAPACK}})/E_{\mathrm{LAPACK}}$
		$\delta q_{\max} = 10^{-5}$	
184	−749.639 387 02	−749.639 386 96	8.0×10^{-11}
368	−1 499.331 634 92	−1 499.331 634 79	8.7×10^{-11}
736	−2 998.742 371 66	−2 998.742 371 36	1.0×10^{-10}
1472	−5 997.782 679 88	−5 997.782 679 20	1.1×10^{-10}
2208	−8 996.781 924 70	−8 996.781 923 64	1.2×10^{-10}
3312	−13 495.305 539 28	−13 495.305 537 64	1.2×10^{-10}
		$\delta q_{\max} = 10^{-6}$	
184	−749.639 426 05	−749.639 426 05	$<6.7 \times 10^{-12}$
368	−1 499.331 689 60	−1 499.331 689 60	$<3.3 \times 10^{-12}$
736	−2 998.742 512 47	−2 998.742 512 47	$<1.7 \times 10^{-12}$
1472	−5 997.782 969 64	−5 997.782 969 62	3.3×10^{-12}
2208	−8 996.782 313 27	−8 996.782 313 25	2.2×10^{-12}
3312	−13 495.306 178 32	−13 495.306 178 28	3.0×10^{-12}

[a]N is the number of water molecules.

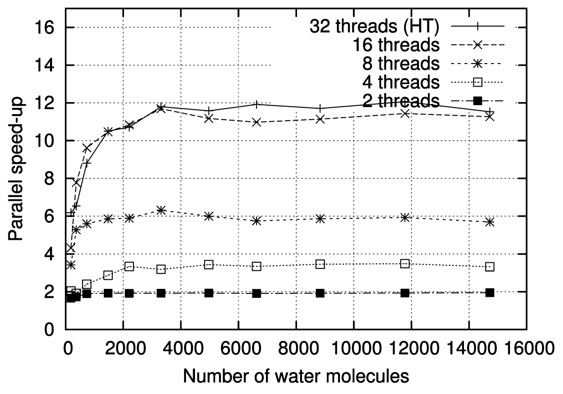


Figure 7. Parallel speedup as a function of the number of water molecules on the dual-socket server. The data were renormalized to remove the effect of the turbo feature. The 32-thread results were obtained using hyperthreading (2 threads per core).

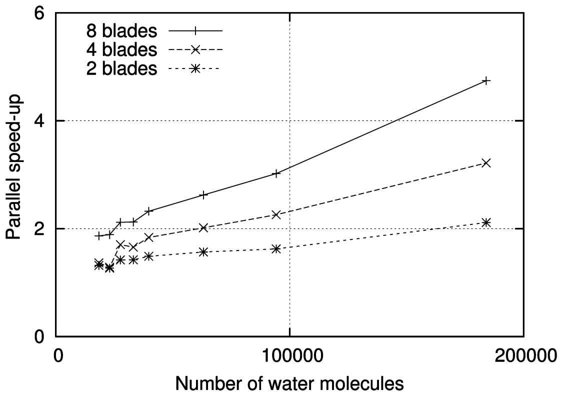


Figure 8. Parallel speedup as a function of the number of water molecules obtained on the Altix UV machine. The reference is one 16-core blade.

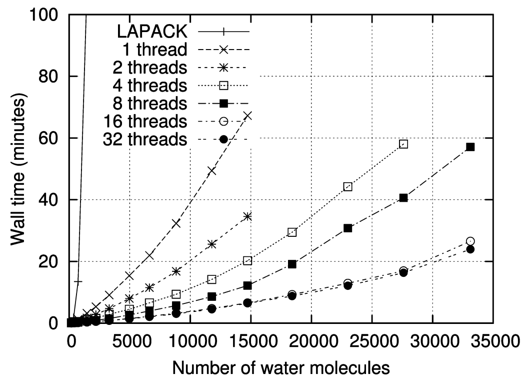


Figure 9. Total wall time (s) as a function of the number of water molecules obtained on the dual-socket server. The 32-thread results were obtained using hyperthreading (2 threads per core).

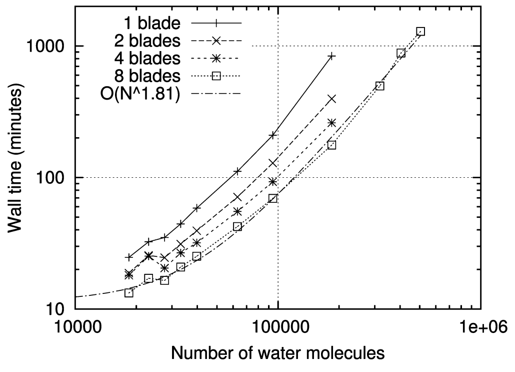


Figure 10. Total wall time (s) as a function of the number of water molecules obtained on the Altix UV machine.

density-matrix framework.[19] The authors reported roughly 10–19 CPU seconds per atom for systems with (50×10^3)–(1.1×10^6) atoms. It is difficult to compare quantitatively our timings with theirs, because the method is different, molecular geometries are different, and the hardware on which the code ran is different. However, one can tell that our implementation gives CPU timings on the same order of magnitude. Another important point is that these authors used an MPI-based implementation, where they report roughly 20% of the time spent in MPI communication routines. Indeed, as linear-scaling techniques are memory-latency bound, obtaining an efficient MPI implementation is a challenge, since MPI communications have a higher latency than memory accesses, even with low-latency network hardware (Infiniband). Note that our large simulations on the Altix UV are very efficient because hardware prefetching occurs even when the target memory is located on a distant blade. Therefore, the hardware makes automatically

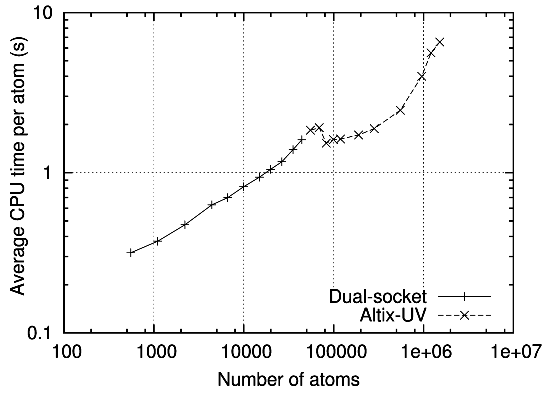


Figure 11. Average CPU time (s) per atom with increasing numbers of water molecules.

asynchronous interblade communication to further reduce the memory latency bottleneck, which is not trivial to implement in a portable MPI framework. This confirms also that our choice to defer the distributed parallelism to coarser graining is a good choice.

5.3. Behavior with Other Systems. We have tested our approach on boxes of coronene molecules and on boxes containing both coronene and water with 6 times more water molecules than coronene. The coronene unit was chosen as it is a fragment on which the molecular orbitals are much more delocalized than on water fragments. The results are displayed in Figure 12.

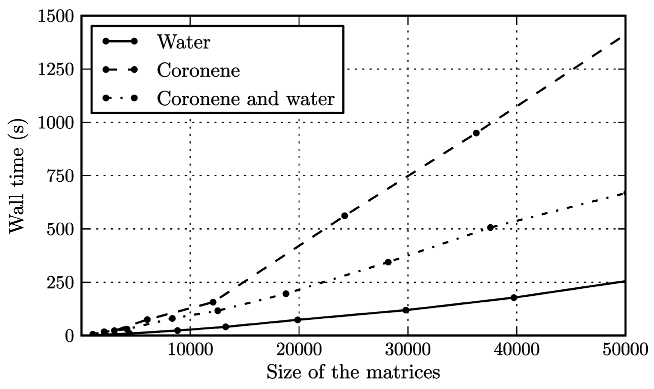


Figure 12. Total wall time as a function of the size of the matrices (total number of MOs) for boxes of water, coronene, and mix of coronene and water. The results were obtained using 32 threads on the 16-core server.

A linear scaling behavior is observed with coronene boxes for large sizes, but with a prefactor 6 times larger than for the box of water molecules. This prefactor is related to the number of nonzero coefficients on the MOs of coronene, which is 2 times larger than is the case for water. Additionally, the number of self-consistent iterations for coronene was also 2 times larger than the number of cycles needed for water. Finally, the number of k-mean centers was 4 times larger for water than for coronene (there was one k-means center per coronene fragment), which is in favor of the water box. As expected, when water and coronene are mixed, an average between the behavior of water and coronene is observed.

6. SUMMARY AND FUTURE WORK

We have presented an efficient OpenMP implementation enabling MO-based SCC-DFTB for very large molecular systems. As linear-scaling techniques reduce the arithmetic intensity, these algorithms are inevitably memory-latency bound. Our implementation was designed to minimize the effect of this bottleneck. We have shown that we were able to run a calculation on more than 1.5 million atoms (504 896 water molecules) with a very good efficiency on 128 cores of an SGI Altix UV machine, a machine with very strong NUMA effects and memory latencies that can be up to 10 times larger than the memory latencies of a standard desktop computer. The program is even more efficient for smaller systems (1000–100 000 molecules), the range which is the most suited to practical applications of DFTB. This aspect will permit one to run much longer molecular dynamics trajectories on medium-sized systems. This OpenMP-only implementation makes it possible to run simulations with up to 100 000 atoms using standard servers, without the need of expensive low-latency network hardware. The errors introduced by our approximations were shown to be below the error of the SCF convergence threshold, which makes us trust our results for simulations with more than 1 million atoms, a domain where it is impossible to verify the validity of the result using an exact diagonalization. The next step of this project will be to implement distributed parallelism to enable large-scale Monte Carlo simulations of biological molecules in water.

The present paper benchmarked the approach applied to DFTB, which is a very favorable case. Indeed, the matrix elements can be computed very fast and the atomic basis sets are very small compared to those used in ab initio calculations. In this context, the MO-coefficient matrices can easily be made sparse. Our approach could be helpful for DFT calculations if the atomic basis set does not contain too diffuse orbitals in order to keep the MO-coefficient matrix sparse. Moreover, it will be efficient only if the calculation of the matrix elements is not the bottleneck, as for instance with the auxiliary density functional theory scheme.[42] Finally, we notice that the scheme presented was benchmarked for molecular clusters, a system for which the building of localized guess MOs is straightforward. Treating fully covalent or metallic systems now requires a more general scheme to derive the set of localized guess MOs.

■ APPENDIX

A. Sparse Storage

Many different sparse storage schemes exist. The choice of the format depends on the work that has to be done with the data. For instance, the compressed sparse row and the compressed sparse column formats,[43] as well as their blocked variants, are usually chosen when using a sparse solver as a black box. Compressed diagonal storage format is mostly used in the finite-element community, since it is well-suited to diagonal band matrices,[44] and the skyline storage format is usually chosen when no pivoting is required for a LU factorization.[45]

The algorithm detailed here implies many rotations between vectors appearing in an unpredictable order. Knowing this, it is convenient to sacrifice some memory and use a scheme where the sparse vectors are equally spaced using padding, such that their size can be easily expanded or reduced after a rotation. Therefore, we use the list of lists (LIL) format implemented as two-dimensional arrays where the first dimensions of the arrays has a fixed size s_m, a little larger than an estimation of the largest encountered size in the calculations, which should be constant for large systems.

Using Fortran notation, a sparse array A(N,N) is represented by an integer array of indices A_ind(0:sm,N) and a real or double-precision array of values A_val(s_m,N). The access to a column is direct using the second index. A column is stored as the list of non-zero values in A_val and the corresponding row indices in A_ind (see Figure 1). The elements at A_ind(0,:) contain the number of non-zero elements in the corresponding columns.

Using compiler directives, the floating-point arrays are aligned on a 512-byte boundary, and s_m is constrained to be a multiple of 8 elements. In this way, all the columns of the array are properly aligned on the 256-bit boundary allowing aligned vector load and store instructions using both the SSE or the AVX instruction sets. Hence, each column starts whether at the beginning of a cache line or at the middle of a cache line (in single precision).

If s_m is set to a multiple of 512 double-precision elements, the columns are spaced by a multiple of 4 KiB. In this case, there will be conflict in the cache when loading two columns, since they will have the same address in the cache. To avoid this so-called *4k-aliasing*, we constrain s_m to follow the rule

$$s_m = \alpha \times 512 + 24 \tag{5}$$

B. Parallelization of the Orthonormalization Step

We now explain how the rotations between orbitals are performed in parallel. First, we prepare an array of OpenMP locks, one lock for each MO. Then, we dress the list of orbital pairs (i, j) to combine those which correspond to an off-diagonal element $|\mathbf{C}^\dagger\mathbf{SC}|_{ij} \geq 10^{-3}\eta$, where η is the largest off-diagonal element of $|\mathbf{C}^\dagger\mathbf{SC}|$. The total length of the list is N_{pair}. Each thread executes simultaneously the following steps:

(1) Pick the first combination (i, j) and go to step 3.
(2) Pick the next combination (i, j).
(3) If the combination (i, j) is already done, go to step 2.
(4) Try to take lock L_i with OMP_TEST_LOCK. If not possible, go to step 2.
(5) Try to take lock L_j with OMP_TEST_LOCK. If not possible, free lock L_i and go to step 2.
(6) Combine i and j and mark (i, j) as done.
(7) Free locks L_i and L_j.
(8) Go back to step 2 until the end of the list is reached.
(9) Go back to step 1 until all combinations are done.

The list of combinations is stored in an array, which can be large. Using the algorithm as presented explores many times the long list of combinations. After the first pass most of the combinations are done and looping over the whole array will spend a lot of time checking that a combination has already been done. An optimized approach consists of looping over only the first half of the array ($k \in [1,N_{\text{pair}}/2]$). After a combination has been done, the elements k and $k + N_{\text{pair}}/2$ of the list are swapped. In this way, when the $N_{\text{pair}}/2$ first elements have been explored, the first $N_{\text{pair}}/2$ elements of the list are combinations to be done. Then, N_{pair} is set to $N_{\text{pair}}/2$, and the procedure is repeated until $N_{\text{pair}} < N_{\text{basis}}$. Using this strategy allows one to loop over only combinations to do and it avoids to check if a combination has already been done (loop over steps 2 and 3). At this point, there remains a few combinations to perform scattered in the array. Those last combinations are handled with the algorithm presented above (steps 1−9), but using blocking functions with OMP_SET_LOCK instead of the non-blocking OMP_TEST_LOCK. As most of the combinations have already been done, the probability of trying to lock simultaneously the same MO will be low and the locking steps will be occasional. Therefore, using blocking locks is relevant here, since it will allow ome to perform only one traversal of the array to complete the remaining combinations.

■ AUTHOR INFORMATION

Corresponding Author

*E-mail: mathias.rapacioli@irsamc.ups-tlse.fr.

Notes

The authors declare no competing financial interest.

■ ACKNOWLEDGMENTS

The authors would like to thank ANR for support under Grant No ANR GASPARIM, ANR PARCS and ANR QMCCHEM. This work was also supported by "Programme Investissements d'Avenir" under the program ANR-11-IDEX-0002-02 reference ANR-10-LABX-0037-NEXT, CALMIP and Equipex Equip@ Meso for computational facilities.

■ REFERENCES

(1) Hohenberg, P.; Kohn, W. *Phys. Rev.* **1964**, *136*, B:864−871.
(2) Kohn, W.; Sham, L. J. *Phys. Rev.* **1965**, *140*, A:1133−1138.
(3) Palser, A. H.; Manolopoulos, D. E. *Phys. Rev. B* **1998**, *58*, 12704−12711.
(4) Niklasson, A. M.; Tymczak, C.; Challacombe, M. *J. Chem. Phys.* **2003**, *118*, 8611−8620.
(5) Niklasson, A. M.; Weber, V.; Challacombe, M. *J. Chem. Phys.* **2005**, *123*, 044107:1−8.
(6) McWeeny, R. *Rev. Mod. Phys.* **1960**, *32*, 335−369.
(7) Challacombe, M. *J. Chem. Phys.* **1999**, *110*, 2332−2342.
(8) Jordan, D. K.; Mazziotti, D. A. *J. Chem. Phys.* **2005**, *122*, 084114:1−5.
(9) Rubensson, E. H.; Rudberg, E.; Sałek, P. *J. Chem. Phys.* **2008**, *128*, 074106:1−14.
(10) Rudberg, E.; Rubensson, E. H. *J. Phys.: Condens. Matter* **2011**, *23*, 075502:1−10.
(11) Rudberg, E.; Rubensson, E. H.; Sałek, P. *J. Chem. Theory Comput.* **2010**, *7*, 340−350.
(12) Li, X.-P.; Nunes, R.; Vanderbilt, D. *Phys. Rev. B* **1993**, *47*, 10891−10894.
(13) Bowler, D.; Gillan, M. *Comput. Phys. Commun.* **1999**, *120*, 95−108.
(14) Li, X.; Millam, J. M.; Scuseria, G. E.; Frisch, M. J.; Schlegel, H. B. *J. Chem. Phys.* **2003**, *119*, 7651−7658.
(15) Millam, J. M.; Scuseria, G. E. *J. Chem. Phys.* **1997**, *106*, 5569−5577.
(16) Adhikari, S.; Baer, R. *J. Chem. Phys.* **2001**, *115*, 11−14.
(17) Daniels, A. D.; Millam, J. M.; Scuseria, G. E. *J. Chem. Phys.* **1997**, *107*, 425−431.
(18) Hung, L.; Carter, E. A. *Chem. Phys. Lett.* **2009**, *475*, 163−170.
(19) VandeVondele, J.; Borštnik, U.; Hutter, J. *J. Chem. Theory Comput.* **2012**, *8*, 3565−3573.
(20) Dixon, S. L.; Merz, K. M., Jr. *J. Chem. Phys.* **1997**, *107*, 879−893.
(21) Stewart, J.; Csaszar, P.; Pulay, P. *J. Comput. Chem.* **1982**, *3*, 227−228.
(22) Heine, T.; Rapacioli, M.; Patchkovskii, S.; Frenzel, J.; Koster, A.; Calaminici, P.; Duarte, H. A.; Escalante, S.; Flores-Moreno, R.; Goursot, A.; Reveles, J.; Salahub, D.; Vela, A. http://physics.jacobs-university.de/theine/research/deMon/, accessed 23 April 2014.
(23) Porezag, D.; Frauenheim, T.; Köhler, T.; Seifert, G.; Kaschner, R. *Phys. Rev. B* **1995**, *51*, 12947−12958.
(24) Seifert, G.; Porezag, D.; Frauenheim, T. *Int. J. Quantum Chem.* **1996**, *58*, 185−192.

(25) Elstner, M.; Porezag, D.; Jungnickel, G.; Elsner, J.; Haugk, M.; Frauenheim, T.; Suhai, S.; Seifert, G. *Phys. Rev. B* **1998**, *58*, 7260–7268.
(26) Frauenheim, T.; Seifert, G.; Elsterner, M.; Hajnal, Z.; Jungnickel, G.; Porezag, D.; Suhai, S.; Scholz, R. *Phys. Status Solidi B* **2000**, *217*, 41–62.
(27) Frauenheim, T.; Seifert, G.; Elstner, M.; Niehaus, T.; Köhler, C.; Amkreutz, M.; Sternberg, M.; Hajnal, Z.; Di Carlo, A.; Suhai, S. *J. Phys.: Condens. Matter* **2002**, *14*, 3015–3047.
(28) Oliveira, A. F.; Seifert, G.; Heine, T.; Duarte, H. A. *J. Braz. Chem. Soc.* **2009**, *20*, 1193–1205.
(29) Aradi, B.; Hourahine, B.; Frauenheim, T. *J. Phys. Chem. A* **2007**, *111*, 5678–5684.
(30) Liu, H.; Elstner, M.; Kaxiras, E.; Frauenheim, T.; Hermans, J.; Yang, W. *Proteins: Struct., Funct., Bioinf.* **2001**, *44*, 484–489.
(31) Giese, T. J.; Chen, H.; Dissanayake, T.; Giambasu, G. M.; Heldenbrand, H.; Huang, M.; Kuechler, E. R.; Lee, T.-S.; Panteva, M. T.; Radak, B. K. *J. Chem. Theory Comput.* **2013**, *9*, 1417–1427.
(32) Dagum, L.; Menon, R. *IEEE Comput. Sci. Eng.* **1998**, *5*, 46–55.
(33) Gropp, W.; Lusk, E.; Doss, N.; Skjellum, A. *Parallel Comput.* **1996**, *22*, 789–828.
(34) MacQueen, J. Some methods for classification and analysis of multivariate observations. *Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Statistics*; University of California Press: Berkley, CA, 1967; pp 281–297.
(35) Angeli, C.; Evangelisti, S.; Cimiraglia, R.; Maynau, D. *J. Chem. Phys.* **2002**, *117*, 10525–10533.
(36) Maynau, D.; Evangelisti, S.; Guihéry, N.; Calzado, C. J.; Malrieu, J.-P. *J. Chem. Phys.* **2002**, *116*, 10060–10068.
(37) Scemama, A.; Caffarel, M.; Oseret, E.; Jalby, W. *J. Comput. Chem.* **2013**, *34*, 938–951.
(38) Elstner, M. *Theor. Chem. Acc.* **2006**, *116*, 316–325.
(39) Slater, J. C.; Koster, G. F. *Phys. Rev.* **1954**, *94*, 1498–1524.
(40) Krüger, T.; Elstner, M.; Schiffels, P.; Frauenheim, T. *J. Chem. Phys.* **2005**, *122*, 114110:1–5.
(41) Anderson, E.; Bai, Z.; Dongarra, J.; Greenbaum, A.; McKenney, A.; Du Croz, J.; Hammerling, S.; Demmel, J.; Bischof, C.; Sorensen, D. LAPACK: A portable linear algebra library for high-performance computers. *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*; 1990; pp 2–11.
(42) Alvarez-Ibarra, A.; Köster, A. M. *J. Chem. Phys.* **2013**, *139*, 024102:1–10.
(43) Kincaid, D. R.; Respess, J. R.; Young, D. M.; Grimes, R. R. *ACM Trans. Math. Soft.* **1982**, *8*, 302–322.
(44) Melhem, R.; Gannon, D. *IJHPCA* **1987**, *1*, 70–98.
(45) Duff, I. S.; Erisman, A. M.; Reid, J. K. *Direct Methods for Sparse Matrices (Monographs on Numerical Analysis)*; Clarendon Press: Oxford, UK, 1986; pp 149–150.