A fast Sparse SCF implementation: Application to DFTB

A. Scemama[1], M. Rapacioli[1]

[1]Laboratoire de Chimie et Physique Quantiques / IRSAMC, Toulouse, France

25 June 2013

Outline

Dense $\times$ dense matrix products can reach $\sim 96\%$ of the peak performance of a CPU:

- Floating-point operations (flops) scale as $\mathcal{O}(N^3)$
- Memory accesses scale as $\mathcal{O}(N^2)$
- Memory accesses have a very high latency (>300 CPU cycles) (1 RAM access is 50–90 nanoseconds: 130–230 CPU cycles at 2.6GHz) but
- Regular memory access patterns can be pre-fetched by the CPU in the caches, hiding the memory latencies
- Compute-bound, very efficient

Dense $\times$ dense matrix products can reach $\sim 96\%$ of the peak performance of a CPU:

- Floating-point operations (flops) scale as $\mathcal{O}(N^3)$
- Memory accesses scale as $\mathcal{O}(N^2)$
- Memory accesses have a very high latency (>300 CPU cycles) (1 RAM access is 50–90 nanoseconds: 130–230 CPU cycles at 2.6GHz) but
- Regular memory access patterns can be pre-fetched by the CPU in the caches, hiding the memory latencies
- Compute-bound, very efficient

Dense $\times$ dense matrix products can reach $\sim 96\%$ of the peak performance of a CPU:

- Floating-point operations (flops) scale as $\mathcal{O}(N^3)$
- Memory accesses scale as $\mathcal{O}(N^2)$
- Memory accesses have a very high latency (>300 CPU cycles) (1 RAM access is 50–90 nanoseconds: 130–230 CPU cycles at 2.6GHz) but
- Regular memory access patterns can be pre-fetched by the CPU in the caches, hiding the memory latencies
- Compute-bound, very efficient

Dense $\times$ dense matrix products can reach $\sim 96\%$ of the peak performance of a CPU:

- Floating-point operations (flops) scale as $\mathcal{O}(N^3)$
- Memory accesses scale as $\mathcal{O}(N^2)$
- Memory accesses have a very high latency (>300 CPU cycles) (1 RAM access is 50–90 nanoseconds: 130–230 CPU cycles at 2.6GHz) but
- Regular memory access patterns can be pre-fetched by the CPU in the caches, hiding the memory latencies
- Compute-bound, very efficient

Dense $\times$ dense matrix products can reach $\sim 96\%$ of the peak performance of a CPU:

- Floating-point operations (flops) scale as $\mathcal{O}(N^3)$
- Memory accesses scale as $\mathcal{O}(N^2)$
- Memory accesses have a very high latency (>300 CPU cycles) (1 RAM access is 50–90 nanoseconds: 130–230 CPU cycles at 2.6GHz) but
- Regular memory access patterns can be pre-fetched by the CPU in the caches, hiding the memory latencies
- Compute-bound, very efficient

Sparse $\times$ sparse matrix products are very difficult to perform efficiently:

- Memory accesses scale as $\mathcal{O}(N)$
- Floating-point operations (flops) scales as $\mathcal{O}(N)$
- Necessarily memory-bound

Element-wise implementations enhance the inefficiency:

- Compiler can't produce efficient code (memory indirections, branches)
- No vectorization

Sparse $\times$ sparse matrix products are very difficult to perform efficiently:

- Memory accesses scale as $\mathcal{O}(N)$
- Floating-point operations (flops) scales as $\mathcal{O}(N)$
- Necessarily memory-bound

Element-wise implementations enhance the inefficiency:

- Compiler can't produce efficient code (memory indirections, branches)
- No vectorization

Sparse $\times$ sparse matrix products are very difficult to perform efficiently:

- Memory accesses scale as $\mathcal{O}(N)$
- Floating-point operations (flops) scales as $\mathcal{O}(N)$
- Necessarily memory-bound

Element-wise implementations enhance the inefficiency:

- Compiler can't produce efficient code (memory indirections, branches)
- No vectorization

Sparse $\times$ sparse matrix products are very difficult to perform efficiently:

- Memory accesses scale as $\mathcal{O}(N)$
- Floating-point operations (flops) scales as $\mathcal{O}(N)$
- Necessarily memory-bound

Element-wise implementations enhance the inefficiency:

- Compiler can't produce efficient code (memory indirections, branches)
- No vectorization

Sparse $\times$ sparse matrix products are very difficult to perform efficiently:

- Memory accesses scale as $\mathcal{O}(N)$
- Floating-point operations (flops) scales as $\mathcal{O}(N)$
- Necessarily memory-bound

Element-wise implementations enhance the inefficiency:

- Compiler can't produce efficient code (memory indirections, branches)
- No vectorization

In another context (Quantum Monte Carlo), we developed an efficient dense $\times$ sparse matrix subroutine for small matrices (<4000), reaching more than 60% of the peak performance:

- All arrays are 256-bit aligned (compiler directives)
- Leading dimensions are multiples of 8 : all columns 256-bit aligned
- Unroll and jam to reduce nb of stores
- All inner-most loops are fully vectorized : always a multiple of 8 loop cycles
- Static analysis of the binary (MAQAO): All inner-most loops can reach the peak of 16 flops/cycle
- "Only" 60%: Both memory and flops $\mathcal{O}(N^2)$
- Memory bound, but memory latencies reduced by pre-fetching in the dense matrix.

In another context (Quantum Monte Carlo), we developed an efficient dense $\times$ sparse matrix subroutine for small matrices (<4000), reaching more than 60% of the peak performance:

- All arrays are 256-bit aligned (compiler directives)
- Leading dimensions are multiples of 8 : all columns 256-bit aligned
- Unroll and jam to reduce nb of stores
- All inner-most loops are fully vectorized : always a multiple of 8 loop cycles
- Static analysis of the binary (MAQAO): All inner-most loops can reach the peak of 16 flops/cycle
- "Only" 60%: Both memory and flops $\mathcal{O}(N^2)$
- Memory bound, but memory latencies reduced by pre-fetching in the dense matrix.

In another context (Quantum Monte Carlo), we developed an efficient dense $\times$ sparse matrix subroutine for small matrices (<4000), reaching more than 60% of the peak performance:

- All arrays are 256-bit aligned (compiler directives)
- Leading dimensions are multiples of 8 : all columns 256-bit aligned
- Unroll and jam to reduce nb of stores
- All inner-most loops are fully vectorized : always a multiple of 8 loop cycles
- Static analysis of the binary (MAQAO): All inner-most loops can reach the peak of 16 flops/cycle
- "Only" 60%: Both memory and flops $\mathcal{O}(N^2)$
- Memory bound, but memory latencies reduced by pre-fetching in the dense matrix.

In another context (Quantum Monte Carlo), we developed an efficient dense $\times$ sparse matrix subroutine for small matrices (<4000), reaching more than 60% of the peak performance:

- All arrays are 256-bit aligned (compiler directives)
- Leading dimensions are multiples of 8 : all columns 256-bit aligned
- Unroll and jam to reduce nb of stores
- All inner-most loops are fully vectorized : always a multiple of 8 loop cycles
- Static analysis of the binary (MAQAO): All inner-most loops can reach the peak of 16 flops/cycle
- "Only" 60%: Both memory and flops $\mathcal{O}(N^2)$
- Memory bound, but memory latencies reduced by pre-fetching in the dense matrix.

In another context (Quantum Monte Carlo), we developed an efficient dense $\times$ sparse matrix subroutine for small matrices (<4000), reaching more than 60% of the peak performance:

- All arrays are 256-bit aligned (compiler directives)
- Leading dimensions are multiples of 8 : all columns 256-bit aligned
- Unroll and jam to reduce nb of stores
- All inner-most loops are fully vectorized : always a multiple of 8 loop cycles
- Static analysis of the binary (MAQAO): All inner-most loops can reach the peak of 16 flops/cycle
- "Only" 60%: Both memory and flops $\mathcal{O}(N^2)$
- Memory bound, but memory latencies reduced by pre-fetching in the dense matrix.

In another context (Quantum Monte Carlo), we developed an efficient dense $\times$ sparse matrix subroutine for small matrices (<4000), reaching more than 60% of the peak performance:

- All arrays are 256-bit aligned (compiler directives)
- Leading dimensions are multiples of 8 : all columns 256-bit aligned
- Unroll and jam to reduce nb of stores
- All inner-most loops are fully vectorized : always a multiple of 8 loop cycles
- Static analysis of the binary (MAQAO): All inner-most loops can reach the peak of 16 flops/cycle
- "Only" 60%: Both memory and flops $\mathcal{O}(N^2)$
- Memory bound, but memory latencies reduced by pre-fetching in the dense matrix.

In another context (Quantum Monte Carlo), we developed an efficient dense $\times$ sparse matrix subroutine for small matrices (<4000), reaching more than 60% of the peak performance:

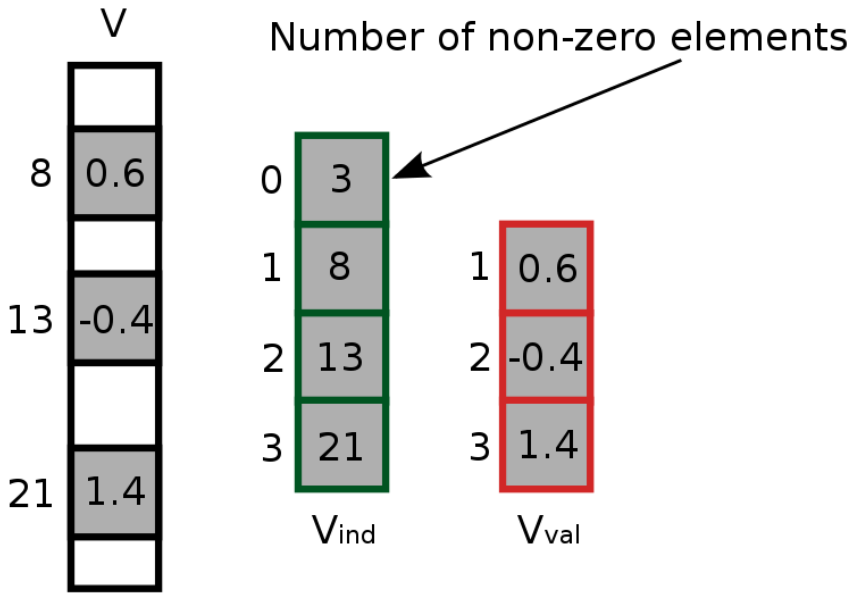
- All arrays are 256-bit aligned (compiler directives)
- Leading dimensions are multiples of 8 : all columns 256-bit aligned
- Unroll and jam to reduce nb of stores
- All inner-most loops are fully vectorized : always a multiple of 8 loop cycles
- Static analysis of the binary (MAQAO): All inner-most loops can reach the peak of 16 flops/cycle
- “Only” 60%: Both memory and flops $\mathcal{O}(N^2)$
- Memory bound, but memory latencies reduced by pre-fetching in the dense matrix.

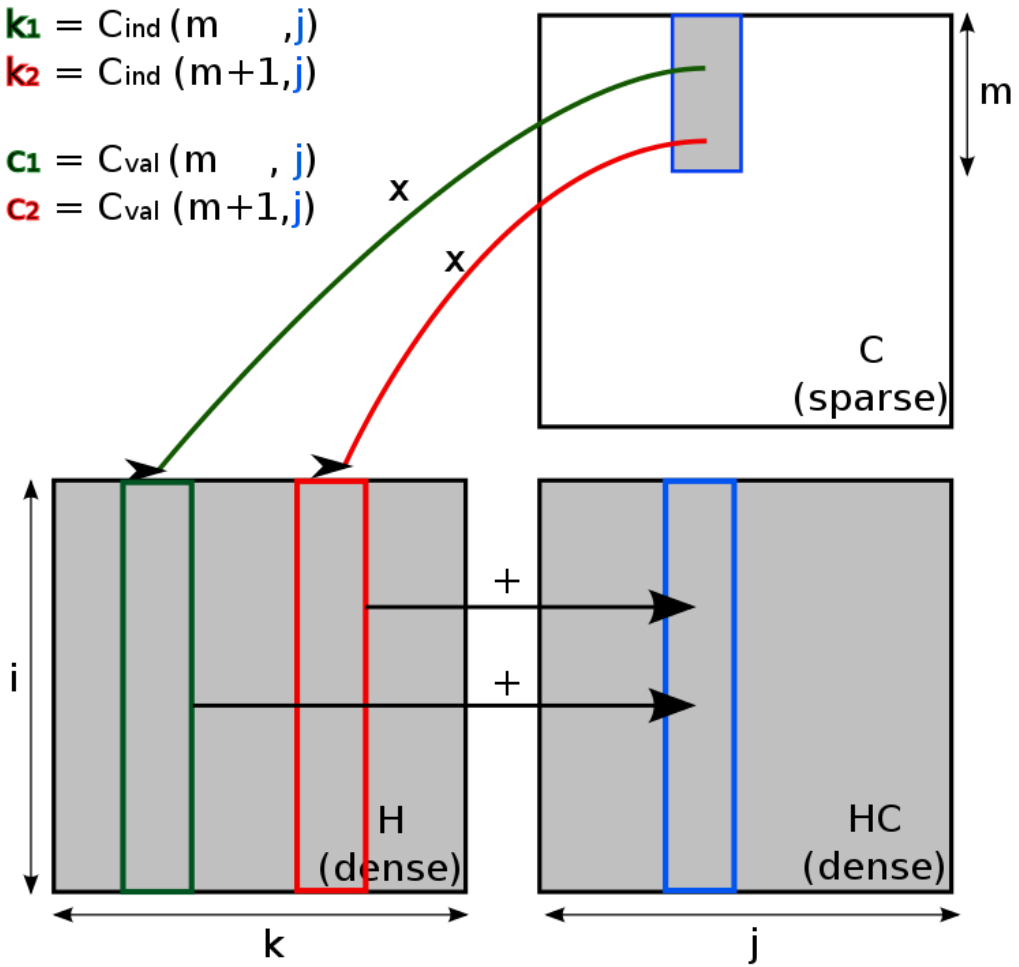
Sparse representation of the matrices

Each sparse matrix is represented by two arrays:

- An array of non-zero indices per column
- An array of corresponding values

Example: One column of matrix V




k1 = Cind (m ,j)
k2 = Cind (m+1,j)
c1 = Cval (m , j)
c2 = Cval (m+1,j)
x
x
m
C
(sparse)
+
+
i
H
(dense)
HC
(dense)
k
j
HC(:,j) = HC(:,j) + c1 x H(:,k1) + c2 x H(:,k2)

Efficient sparse $\times$ sparse matrix product in deMon-Nano:

- Represent one sparse matrix as a collection of small dense sub-matrices
- Each dense sub-matrix is 256-bit aligned
- Each column of the sub-matrix is 256-bit aligned (padding)
- The number of loop-cycles is set to a multiple of 8

Efficient sparse $\times$ sparse matrix product in deMon-Nano:

- Represent one sparse matrix as a collection of small dense sub-matrices
- Each dense sub-matrix is 256-bit aligned
- Each column of the sub-matrix is 256-bit aligned (padding)
- The number of loop-cycles is set to a multiple of 8

Efficient sparse $\times$ sparse matrix product in deMon-Nano:

- Represent one sparse matrix as a collection of small dense sub-matrices
- Each dense sub-matrix is 256-bit aligned
- Each column of the sub-matrix is 256-bit aligned (padding)
- The number of loop-cycles is set to a multiple of 8

Efficient sparse $\times$ sparse matrix product in deMon-Nano:

- Represent one sparse matrix as a collection of small dense sub-matrices
- Each dense sub-matrix is 256-bit aligned
- Each column of the sub-matrix is 256-bit aligned (padding)
- The number of loop-cycles is set to a multiple of 8

Outline

The system is partitioned using a constrained variant of the *k-means* clustering algorithm:

1. A set of m centers (*means*) is first distributed evenly in the 3D-space
2. Each molecule is attached to its closest center, such that each center is connected to 4 molecules
3. The position of the centers is moved to the centroid of the connected molecules
4. Go back to step 2 until the partition doesn't change

The system is partitioned using a constrained variant of the *k-means* clustering algorithm:

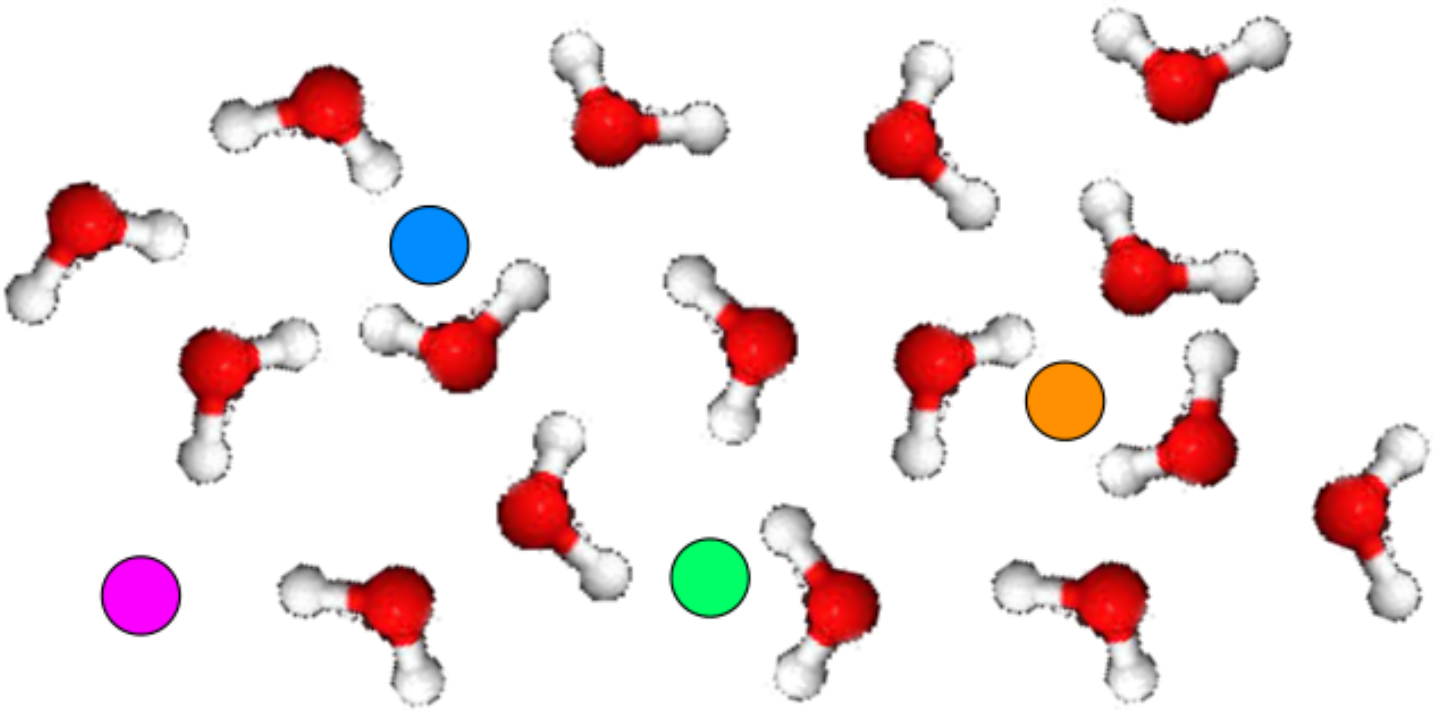
1. A set of m centers (*means*) is first distributed evenly in the 3D-space
2. Each molecule is attached to its closest center, such that each center is connected to 4 molecules
3. The position of the centers is moved to the centroid of the connected molecules
4. Go back to step 2 until the partition doesn't change

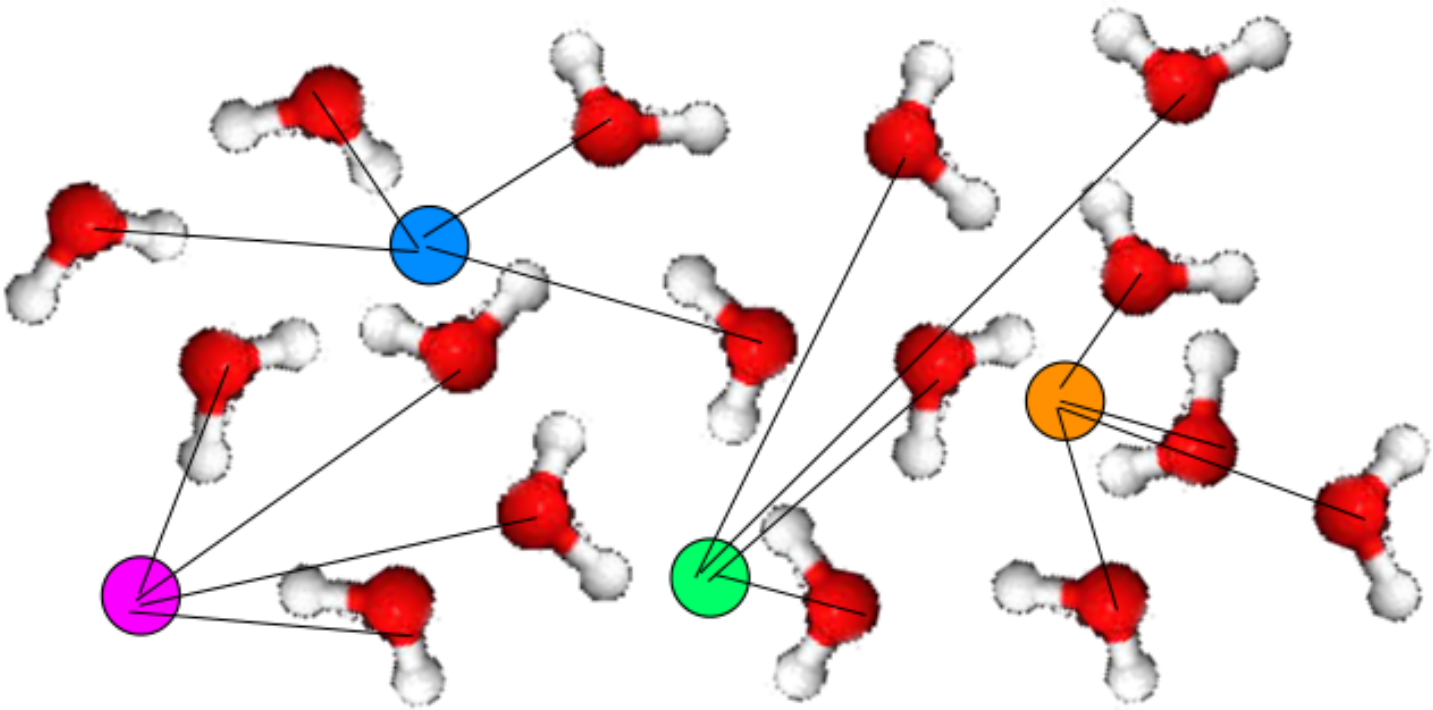
The system is partitioned using a constrained variant of the *k-means* clustering algorithm:

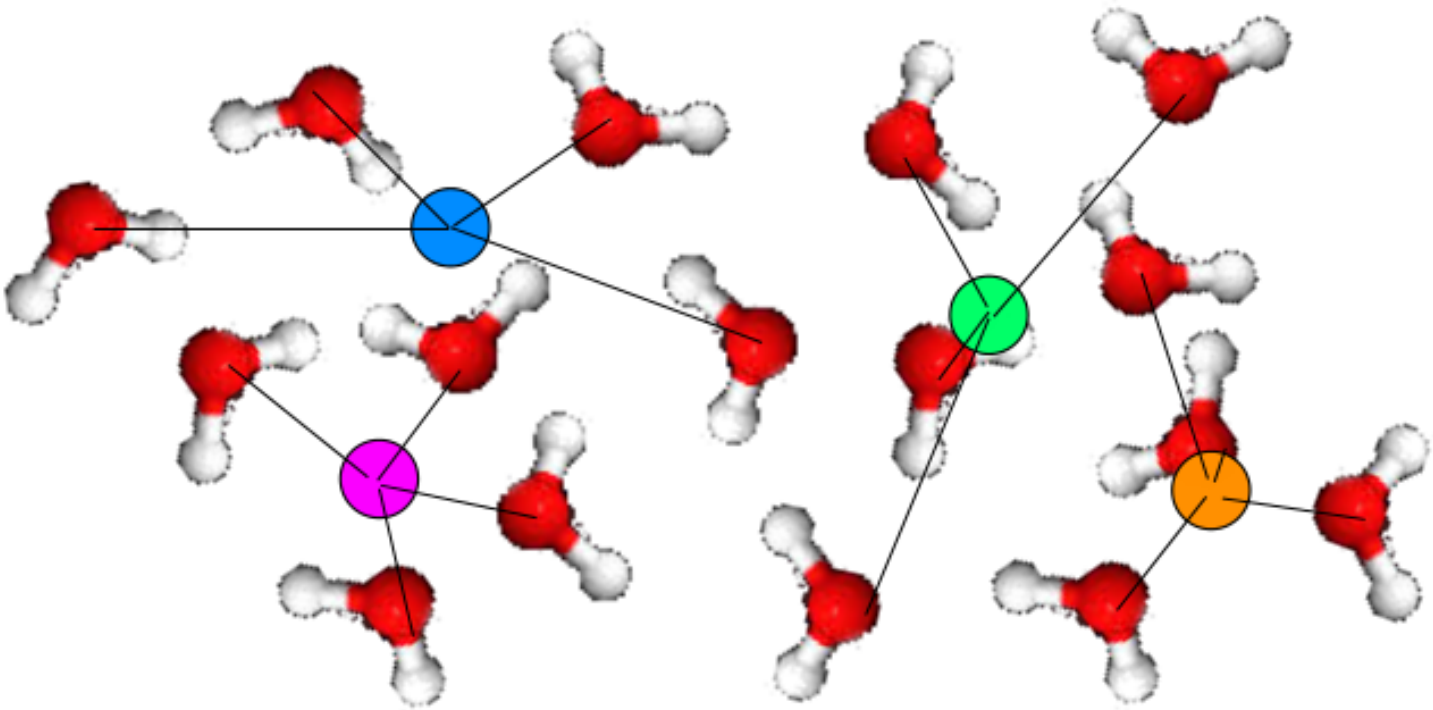
1. A set of m centers (*means*) is first distributed evenly in the 3D-space
2. Each molecule is attached to its closest center, such that each center is connected to 4 molecules
3. The position of the centers is moved to the centroid of the connected molecules
4. Go back to step 2 until the partition doesn't change

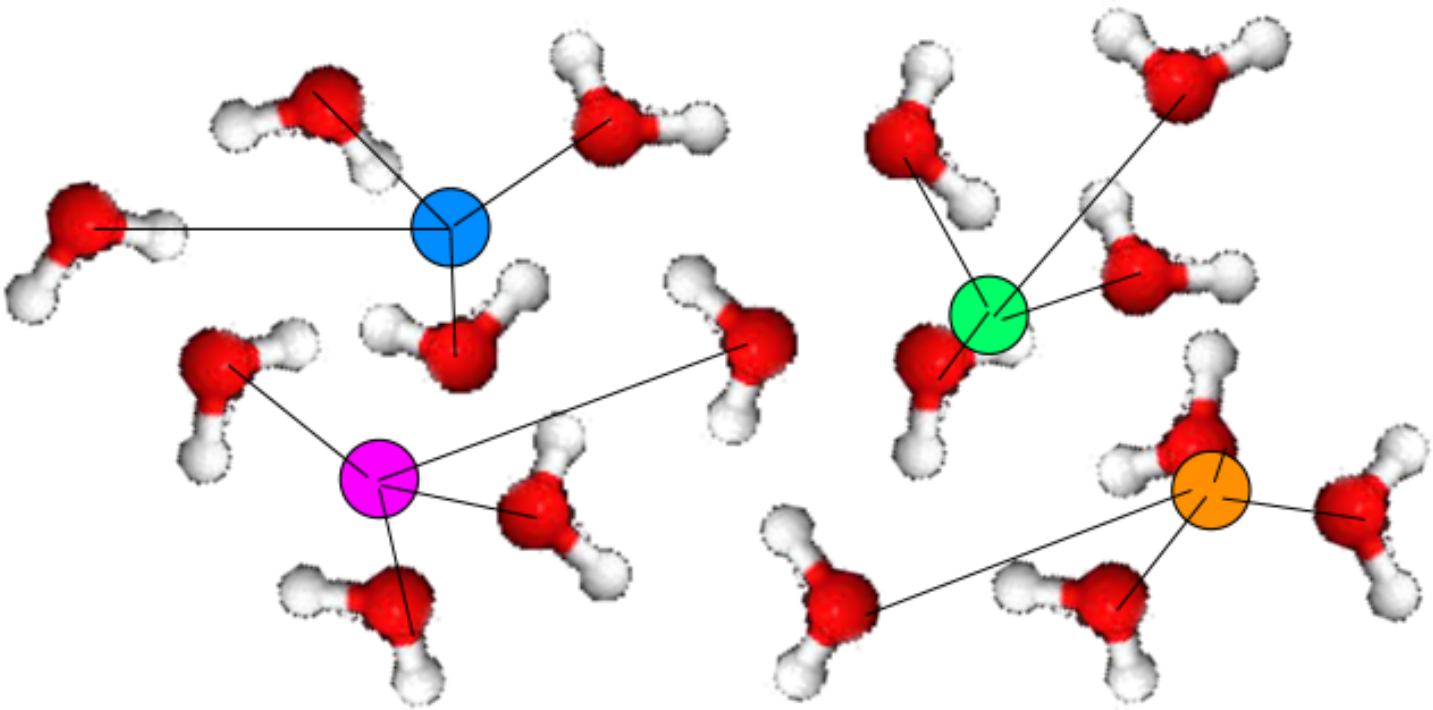
The system is partitioned using a constrained variant of the *k-means* clustering algorithm:

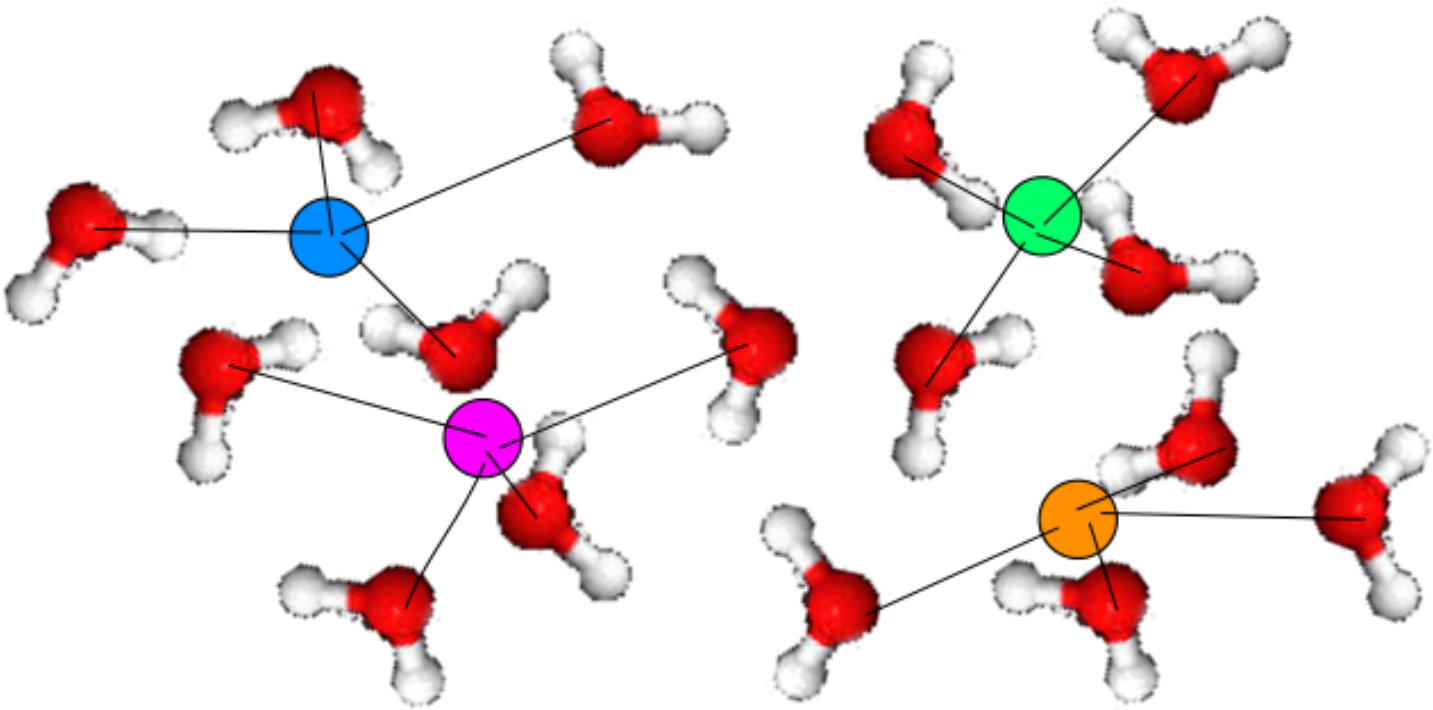
1. A set of m centers (*means*) is first distributed evenly in the 3D-space
2. Each molecule is attached to its closest center, such that each center is connected to 4 molecules
3. The position of the centers is moved to the centroid of the connected molecules
4. Go back to step 2 until the partition doesn't change











- The atoms are then ordered by *k-means* centers
- *k-means* neighbours are centers with at least 2 atoms less than 20 a.u

Outline

- For each molecule, perform an independent non-SCC DFTB calculation (Solve $\mathcal{H}\mathbf{C} = \mathbf{ESC}$)
- Pack together occupied MOs of each *k-means* center
- Pack together virtual MOs of each *k-means* center

Remarks:

- MOs belonging to the same molecule are orthonormal
- MOs belonging to *k-means* centers which are not neighbours have a zero overlap
- MOs have non-zero coefficients only on basis functions which belong to the same molecule
- The orbitals are not orthonormal, but both the overlap ($\mathbf{S}$) and the MO coefficient ($\mathbf{C}$) matrix are sparse.

- For each molecule, perform an independent non-SCC DFTB calculation (Solve $\mathcal{H}\mathbf{C} = \mathbf{ESC}$)
- Pack together occupied MOs of each *k-means* center
- Pack together virtual MOs of each *k-means* center

Remarks:

- MOs belonging to the same molecule are orthonormal
- MOs belonging to *k-means* centers which are not neighbours have a zero overlap
- MOs have non-zero coefficients only on basis functions which belong to the same molecule
- The orbitals are not orthonormal, but both the overlap ($\mathbf{S}$) and the MO coefficient ($\mathbf{C}$) matrix are sparse.

- For each molecule, perform an independent non-SCC DFTB calculation (Solve $\mathcal{H}\mathbf{C} = \mathbf{ESC}$)
- Pack together occupied MOs of each *k-means* center
- Pack together virtual MOs of each *k-means* center

Remarks:

- MOs belonging to the same molecule are orthonormal
- MOs belonging to *k-means* centers which are not neighbours have a zero overlap
- MOs have non-zero coefficients only on basis functions which belong to the same molecule
- The orbitals are not orthonormal, but both the overlap ($\mathbf{S}$) and the MO coefficient ($\mathbf{C}$) matrix are sparse.

- For each molecule, perform an independent non-SCC DFTB calculation (Solve $\mathcal{H}\mathbf{C} = \mathbf{ESC}$)
- Pack together occupied MOs of each *k-means* center
- Pack together virtual MOs of each *k-means* center

Remarks:

- MOs belonging to the same molecule are orthonormal
- MOs belonging to *k-means* centers which are not neighbours have a zero overlap
- MOs have non-zero coefficients only on basis functions which belong to the same molecule
- The orbitals are not orthonormal, but both the overlap ($\mathbf{S}$) and the MO coefficient ($\mathbf{C}$) matrix are sparse.

- For each molecule, perform an independent non-SCC DFTB calculation (Solve $\mathcal{H}\mathbf{C} = \mathbf{ESC}$)
- Pack together occupied MOs of each *k-means* center
- Pack together virtual MOs of each *k-means* center

Remarks:

- MOs belonging to the same molecule are orthonormal
- MOs belonging to *k-means* centers which are not neighbours have a zero overlap
- MOs have non-zero coefficients only on basis functions which belong to the same molecule
- The orbitals are not orthonormal, but both the overlap ($\mathbf{S}$) and the MO coefficient ($\mathbf{C}$) matrix are sparse.

- For each molecule, perform an independent non-SCC DFTB calculation (Solve $\mathcal{H}\mathbf{C} = \mathbf{ESC}$)
- Pack together occupied MOs of each *k-means* center
- Pack together virtual MOs of each *k-means* center

Remarks:

- MOs belonging to the same molecule are orthonormal
- MOs belonging to *k-means* centers which are not neighbours have a zero overlap
- MOs have non-zero coefficients only on basis functions which belong to the same molecule
- The orbitals are not orthonormal, but both the overlap ($\mathbf{S}$) and the MO coefficient ($\mathbf{C}$) matrix are sparse.

- For each molecule, perform an independent non-SCC DFTB calculation (Solve $\mathcal{H}\mathbf{C} = \mathbf{ESC}$)
- Pack together occupied MOs of each *k-means* center
- Pack together virtual MOs of each *k-means* center

Remarks:

- MOs belonging to the same molecule are orthonormal
- MOs belonging to *k-means* centers which are not neighbours have a zero overlap
- MOs have non-zero coefficients only on basis functions which belong to the same molecule
- The orbitals are not orthonormal, but both the overlap ($\mathbf{S}$) and the MO coefficient ($\mathbf{C}$) matrix are sparse.

Outline

Diagonalization of $\mathbf{C}^\dagger\mathbf{S}\mathbf{C}$

1. The $\mathbf{C}$ matrix is already stored sparse
2. Compute $\mathbf{S}$ (sparse)
3. Compute $\mathbf{C}^\dagger\mathbf{S}\mathbf{C}$ (sparse)
4. Normalize using diagonal elements
5. Perform Jacobi-like rotations to remove the largest off-diagonal elements of $\mathbf{C}^\dagger\mathbf{S}\mathbf{C}$
6. Go back to step 3 until the largest off-diagonal element is below a threshold

Output : $\mathbf{C}^\dagger\mathbf{S}\mathbf{C} = \mathbf{I}$

Diagonalization of $\mathbf{C}^\dagger\mathbf{S}\mathbf{C}$

1. The $\mathbf{C}$ matrix is already stored sparse
2. Compute $\mathbf{S}$ (sparse)
3. Compute $\mathbf{C}^\dagger\mathbf{S}\mathbf{C}$ (sparse)
4. Normalize using diagonal elements
5. Perform Jacobi-like rotations to remove the largest off-diagonal elements of $\mathbf{C}^\dagger\mathbf{S}\mathbf{C}$
6. Go back to step 3 until the largest off-diagonal element is below a threshold

Output : $\mathbf{C}^\dagger\mathbf{S}\mathbf{C} = \mathbf{I}$

Diagonalization of $\mathbf{C}^\dagger\mathbf{S}\mathbf{C}$

1. The $\mathbf{C}$ matrix is already stored sparse
2. Compute $\mathbf{S}$ (sparse)
3. Compute $\mathbf{C}^\dagger\mathbf{S}\mathbf{C}$ (sparse)
4. Normalize using diagonal elements
5. Perform Jacobi-like rotations to remove the largest off-diagonal elements of $\mathbf{C}^\dagger\mathbf{S}\mathbf{C}$
6. Go back to step 3 until the largest off-diagonal element is below a threshold

Output : $\mathbf{C}^\dagger\mathbf{S}\mathbf{C} = \mathbf{I}$

Diagonalization of $\mathbf{C}^\dagger\mathbf{S}\mathbf{C}$

1. The $\mathbf{C}$ matrix is already stored sparse
2. Compute $\mathbf{S}$ (sparse)
3. Compute $\mathbf{C}^\dagger\mathbf{S}\mathbf{C}$ (sparse)
4. Normalize using diagonal elements
5. Perform Jacobi-like rotations to remove the largest off-diagonal elements of $\mathbf{C}^\dagger\mathbf{S}\mathbf{C}$
6. Go back to step 3 until the largest off-diagonal element is below a threshold

Output : $\mathbf{C}^\dagger\mathbf{S}\mathbf{C} = \mathbf{I}$

Diagonalization of $\mathbf{C}^\dagger\mathbf{S}\mathbf{C}$

1. The $\mathbf{C}$ matrix is already stored sparse
2. Compute $\mathbf{S}$ (sparse)
3. Compute $\mathbf{C}^\dagger\mathbf{S}\mathbf{C}$ (sparse)
4. Normalize using diagonal elements
5. Perform Jacobi-like rotations to remove the largest off-diagonal elements of $\mathbf{C}^\dagger\mathbf{S}\mathbf{C}$
6. Go back to step 3 until the largest off-diagonal element is below a threshold

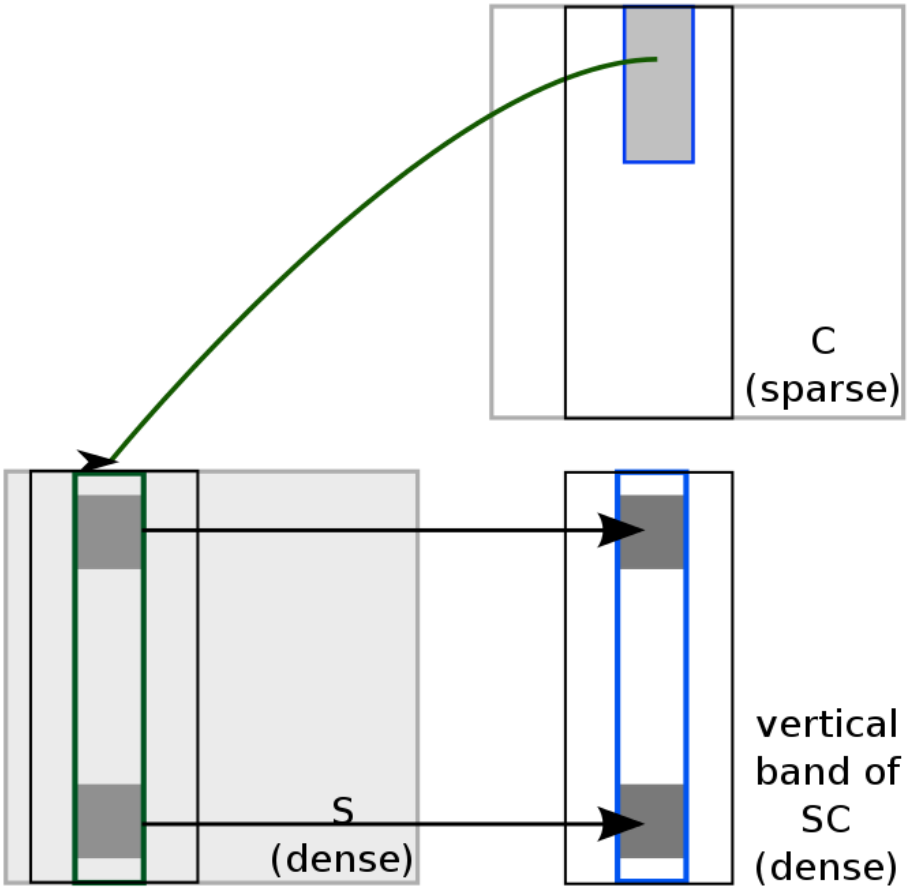
Output : $\mathbf{C}^\dagger\mathbf{S}\mathbf{C} = \mathbf{I}$

Diagonalization of $\mathbf{C}^\dagger\mathbf{S}\mathbf{C}$

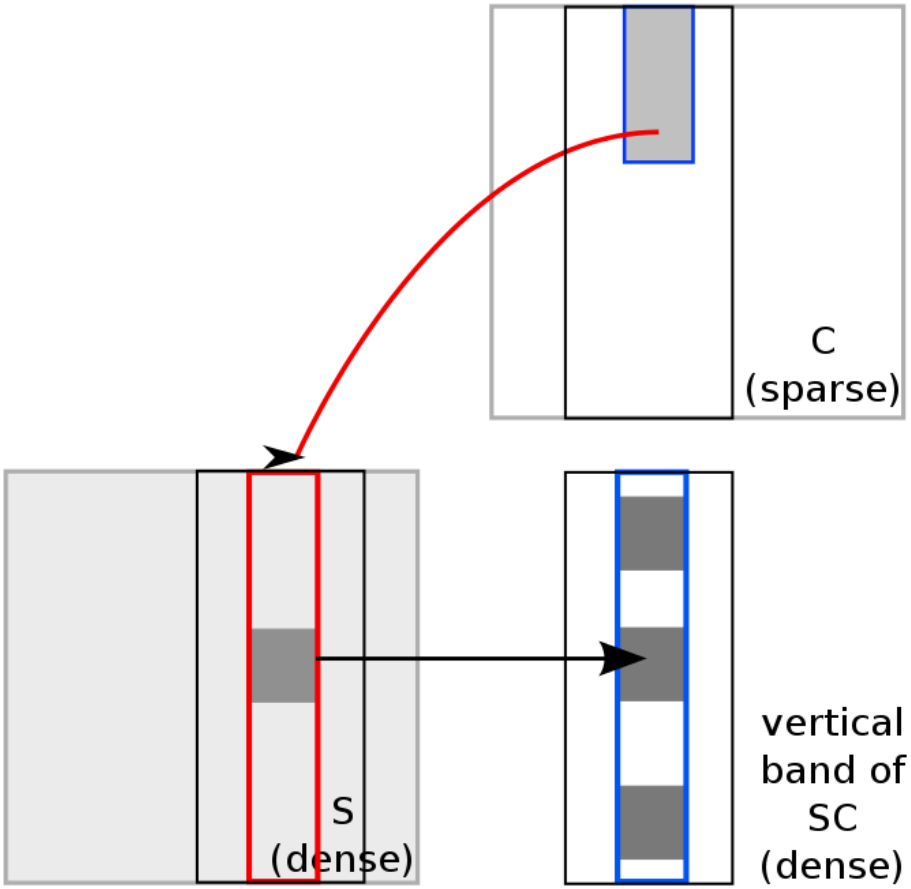
1. The $\mathbf{C}$ matrix is already stored sparse
2. Compute $\mathbf{S}$ (sparse)
3. Compute $\mathbf{C}^\dagger\mathbf{S}\mathbf{C}$ (sparse)
4. Normalize using diagonal elements
5. Perform Jacobi-like rotations to remove the largest off-diagonal elements of $\mathbf{C}^\dagger\mathbf{S}\mathbf{C}$
6. Go back to step 3 until the largest off-diagonal element is below a threshold

Output : $\mathbf{C}^\dagger\mathbf{S}\mathbf{C} = \mathbf{I}$

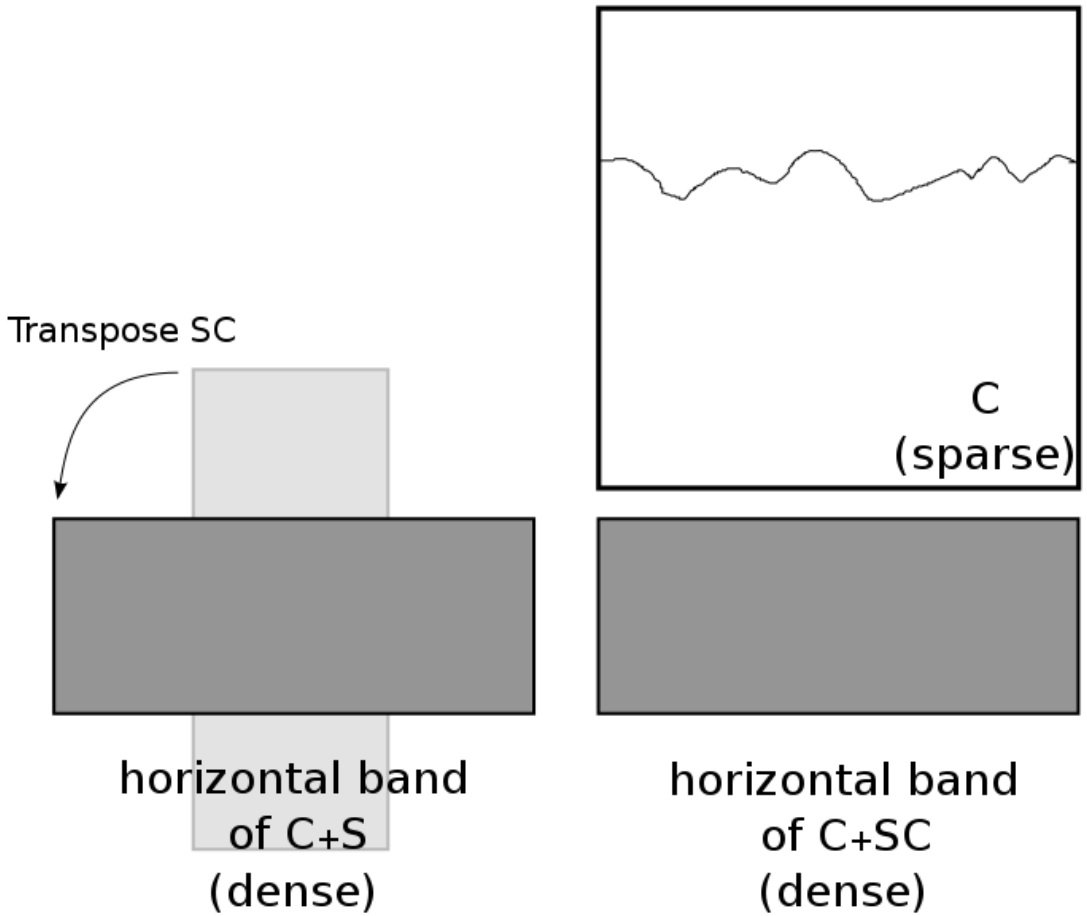
Parallel implementation of $\mathbf{C}^\dagger\mathbf{SC}$



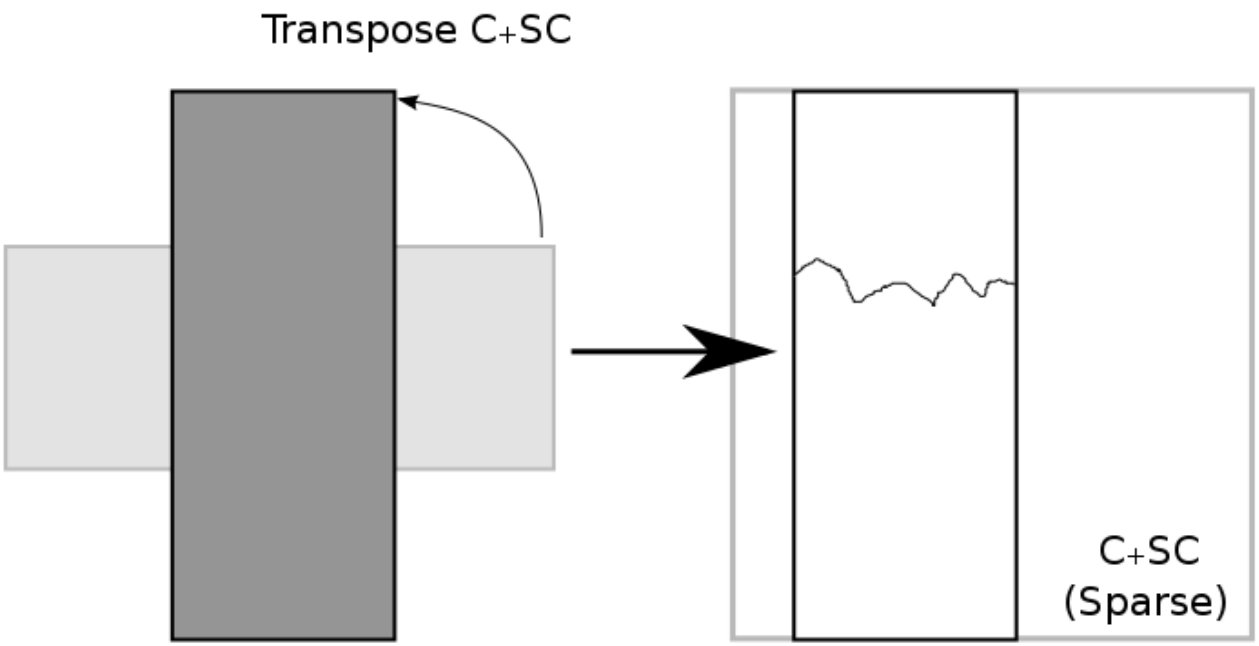
Parallel implementation of $\mathbf{C}^\dagger\mathbf{SC}$



Parallel implementation of $\mathbf{C}^\dagger\mathbf{SC}$



Parallel implementation of $\mathbf{C}^\dagger\mathbf{SC}$



Parallel implementation of Jacobi-like rotations

First, dress the list of rotations to do: $|\left(\mathbf{C}^\dagger\mathbf{S}\mathbf{C}\right)_{i>j}| > \epsilon$.

Prepare a 1D-array of OpenMP locks, (one lock for each MO).

All CPUs do at the same time:

1. Pick the next rotation (i,j)
2. If rotation (i,j) is already done, go to step 1
3. Try to take lock i
4. If not possible, go to step 1
5. Try to take lock j
6. If not possible, free lock i and go to step 1
7. Rotate i and j
8. Mark (i,j) as done
9. Free locks i and j
10. Go back to step 1 until all rotations are done

Parallel implementation of Jacobi-like rotations

First, dress the list of rotations to do: $|\left(\mathbf{C}^{\dagger}\mathbf{S}\mathbf{C}\right)_{i>j}| > \epsilon$.

Prepare a 1D-array of OpenMP locks, (one lock for each MO).

All CPUs do at the same time:

1. Pick the next rotation (i,j)
2. If rotation (i,j) is already done, go to step 1
3. Try to take lock i
4. If not possible, go to step 1
5. Try to take lock j
6. If not possible, free lock i and go to step 1
7. Rotate i and j
8. Mark (i,j) as done
9. Free locks i and j
10. Go back to step 1 until all rotations are done

Parallel implementation of Jacobi-like rotations

First, dress the list of rotations to do: $|\left(\mathbf{C}^\dagger\mathbf{S}\mathbf{C}\right)_{i>j}| > \epsilon$.

Prepare a 1D-array of OpenMP locks, (one lock for each MO).

All CPUs do at the same time:

1. Pick the next rotation (i,j)
2. If rotation (i,j) is already done, go to step 1
3. Try to take lock i
4. If not possible, go to step 1
5. Try to take lock j
6. If not possible, free lock i and go to step 1
7. Rotate i and j
8. Mark (i,j) as done
9. Free locks i and j
10. Go back to step 1 until all rotations are done

Parallel implementation of Jacobi-like rotations

First, dress the list of rotations to do: $|\left(\mathbf{C}^\dagger\mathbf{S}\mathbf{C}\right)_{i>j}| > \epsilon$.

Prepare a 1D-array of OpenMP locks, (one lock for each MO).

All CPUs do at the same time:

1. Pick the next rotation (i,j)
2. If rotation (i,j) is already done, go to step 1
3. Try to take lock i
4. If not possible, go to step 1
5. Try to take lock j
6. If not possible, free lock i and go to step 1
7. Rotate i and j
8. Mark (i,j) as done
9. Free locks i and j
10. Go back to step 1 until all rotations are done

Parallel implementation of Jacobi-like rotations

First, dress the list of rotations to do: $|\left(\mathbf{C}^\dagger\mathbf{S}\mathbf{C}\right)_{i>j}| > \epsilon$.

Prepare a 1D-array of OpenMP locks, (one lock for each MO).

All CPUs do at the same time:

1. Pick the next rotation (i,j)
2. If rotation (i,j) is already done, go to step 1
3. Try to take lock i
4. If not possible, go to step 1
5. Try to take lock j
6. If not possible, free lock i and go to step 1
7. Rotate i and j
8. Mark (i,j) as done
9. Free locks i and j
10. Go back to step 1 until all rotations are done

Parallel implementation of Jacobi-like rotations

First, dress the list of rotations to do: $|\left(\mathbf{C}^{\dagger}\mathbf{S}\mathbf{C}\right)_{i>j}| > \epsilon$.

Prepare a 1D-array of OpenMP locks, (one lock for each MO).

All CPUs do at the same time:

1. Pick the next rotation (i, j)
2. If rotation (i, j) is already done, go to step 1
3. Try to take lock i
4. If not possible, go to step 1
5. Try to take lock j
6. If not possible, free lock i and go to step 1
7. Rotate i and j
8. Mark (i, j) as done
9. Free locks i and j
10. Go back to step 1 until all rotations are done

Parallel implementation of Jacobi-like rotations

First, dress the list of rotations to do: $|\left(\mathbf{C}^{\dagger}\mathbf{S}\mathbf{C}\right)_{i>j}| > \epsilon$.

Prepare a 1D-array of OpenMP locks, (one lock for each MO).

All CPUs do at the same time:

1. Pick the next rotation (i,j)
2. If rotation (i,j) is already done, go to step 1
3. Try to take lock i
4. If not possible, go to step 1
5. Try to take lock j
6. If not possible, free lock i and go to step 1
7. Rotate i and j
8. Mark (i,j) as done
9. Free locks i and j
10. Go back to step 1 until all rotations are done

Parallel implementation of Jacobi-like rotations

First, dress the list of rotations to do: $|\left(\mathbf{C}^{\dagger}\mathbf{S}\mathbf{C}\right)_{i>j}| > \epsilon$.

Prepare a 1D-array of OpenMP locks, (one lock for each MO).

All CPUs do at the same time:

1. Pick the next rotation (i, j)
2. If rotation (i, j) is already done, go to step 1
3. Try to take lock i
4. If not possible, go to step 1
5. Try to take lock j
6. If not possible, free lock i and go to step 1
7. Rotate i and j
8. Mark (i, j) as done
9. Free locks i and j
10. Go back to step 1 until all rotations are done

Parallel implementation of Jacobi-like rotations

First, dress the list of rotations to do: $|\left(\mathbf{C}^{\dagger}\mathbf{S}\mathbf{C}\right)_{i>j}| > \epsilon$.

Prepare a 1D-array of OpenMP locks, (one lock for each MO).

All CPUs do at the same time:

1. Pick the next rotation (i, j)
2. If rotation (i, j) is already done, go to step 1
3. Try to take lock i
4. If not possible, go to step 1
5. Try to take lock j
6. If not possible, free lock i and go to step 1
7. Rotate i and j
8. Mark (i, j) as done
9. Free locks i and j
10. Go back to step 1 until all rotations are done

Parallel implementation of Jacobi-like rotations

First, dress the list of rotations to do: $|\left(\mathbf{C}^{\dagger}\mathbf{S}\mathbf{C}\right)_{i>j}| > \epsilon$.

Prepare a 1D-array of OpenMP locks, (one lock for each MO).

All CPUs do at the same time:

1. Pick the next rotation (i, j)
2. If rotation (i, j) is already done, go to step 1
3. Try to take lock i
4. If not possible, go to step 1
5. Try to take lock j
6. If not possible, free lock i and go to step 1
7. Rotate i and j
8. Mark (i, j) as done
9. Free locks i and j
10. Go back to step 1 until all rotations are done

Outline

- The energy is independent of unitary transformations among occupied-occupied blocks and virtual-virtual blocks
- Brillouin's theorem: It is sufficient to zero the occupied-virtual blocks
- Avoiding occupied-occupied and virtual-virtual rotations keeps the orbitals local (sparse)
- The partial diagonalization of $\mathbf{C}^\dagger\mathbf{H}\mathbf{C}$ is performed like the diagonalization $\mathbf{C}^\dagger\mathbf{S}\mathbf{C}$, computing only the occupied-virtual blocks.
- Double precision is not required here $\longrightarrow$ Single precision.
- The self-consistence step is inserted before each $\mathbf{C}^\dagger\mathbf{H}\mathbf{C}$ product

- The energy is independent of unitary transformations among occupied-occupied blocks and virtual-virtual blocks
- Brillouin's theorem: It is sufficient to zero the occupied-virtual blocks
- Avoiding occupied-occupied and virtual-virtual rotations keeps the orbitals local (sparse)
- The partial diagonalization of $\mathbf{C}^\dagger\mathbf{H}\mathbf{C}$ is performed like the diagonalization $\mathbf{C}^\dagger\mathbf{S}\mathbf{C}$, computing only the occupied-virtual blocks.
- Double precision is not required here $\longrightarrow$ Single precision.
- The self-consistence step is inserted before each $\mathbf{C}^\dagger\mathbf{H}\mathbf{C}$ product

- The energy is independent of unitary transformations among occupied-occupied blocks and virtual-virtual blocks
- Brillouin's theorem: It is sufficient to zero the occupied-virtual blocks
- Avoiding occupied-occupied and virtual-virtual rotations keeps the orbitals local (sparse)
- The partial diagonalization of $\mathbf{C}^\dagger\mathbf{H}\mathbf{C}$ is performed like the diagonalization $\mathbf{C}^\dagger\mathbf{S}\mathbf{C}$, computing only the occupied-virtual blocks.
- Double precision is not required here $\longrightarrow$ Single precision.
- The self-consistence step is inserted before each $\mathbf{C}^\dagger\mathbf{H}\mathbf{C}$ product

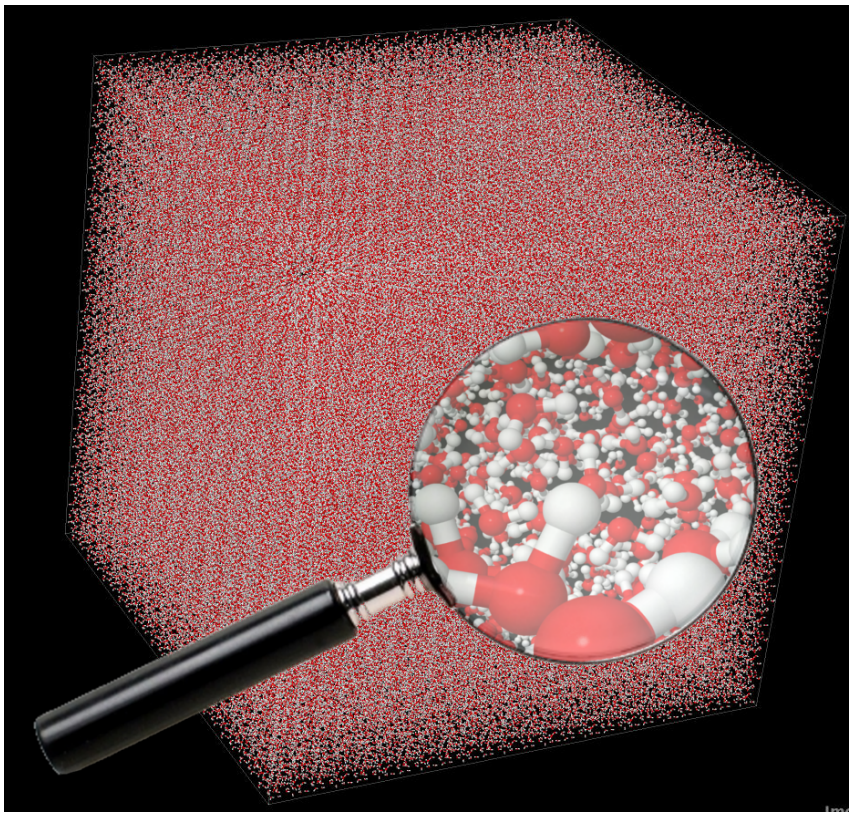
- The energy is independent of unitary transformations among occupied-occupied blocks and virtual-virtual blocks
- Brillouin's theorem: It is sufficient to zero the occupied-virtual blocks
- Avoiding occupied-occupied and virtual-virtual rotations keeps the orbitals local (sparse)
- The partial diagonalization of $\mathbf{C}^\dagger\mathbf{HC}$ is performed like the diagonalization $\mathbf{C}^\dagger\mathbf{SC}$, computing only the occupied-virtual blocks.
- Double precision is not required here $\longrightarrow$ Single precision.
- The self-consistence step is inserted before each $\mathbf{C}^\dagger\mathbf{HC}$ product

- The energy is independent of unitary transformations among occupied-occupied blocks and virtual-virtual blocks
- Brillouin's theorem: It is sufficient to zero the occupied-virtual blocks
- Avoiding occupied-occupied and virtual-virtual rotations keeps the orbitals local (sparse)
- The partial diagonalization of $\mathbf{C}^\dagger\mathbf{HC}$ is performed like the diagonalization $\mathbf{C}^\dagger\mathbf{SC}$, computing only the occupied-virtual blocks.
- Double precision is not required here $\longrightarrow$ Single precision.
- The self-consistence step is inserted before each $\mathbf{C}^\dagger\mathbf{HC}$ product

- The energy is independent of unitary transformations among occupied-occupied blocks and virtual-virtual blocks
- Brillouin's theorem: It is sufficient to zero the occupied-virtual blocks
- Avoiding occupied-occupied and virtual-virtual rotations keeps the orbitals local (sparse)
- The partial diagonalization of $\mathbf{C}^\dagger\mathbf{HC}$ is performed like the diagonalization $\mathbf{C}^\dagger\mathbf{SC}$, computing only the occupied-virtual blocks.
- Double precision is not required here $\longrightarrow$ Single precision.
- The self-consistence step is inserted before each $\mathbf{C}^\dagger\mathbf{HC}$ product

Outline

Benchmark: Boxes of water molecules, up to 120 000 atoms



- One compute node (OpenMP)
- Intel Xeon E5-2670, 2.7GHz (up to 3.3 GHz turbo)
- 2 sockets, 8 cores/socket
- 64Gb RAM

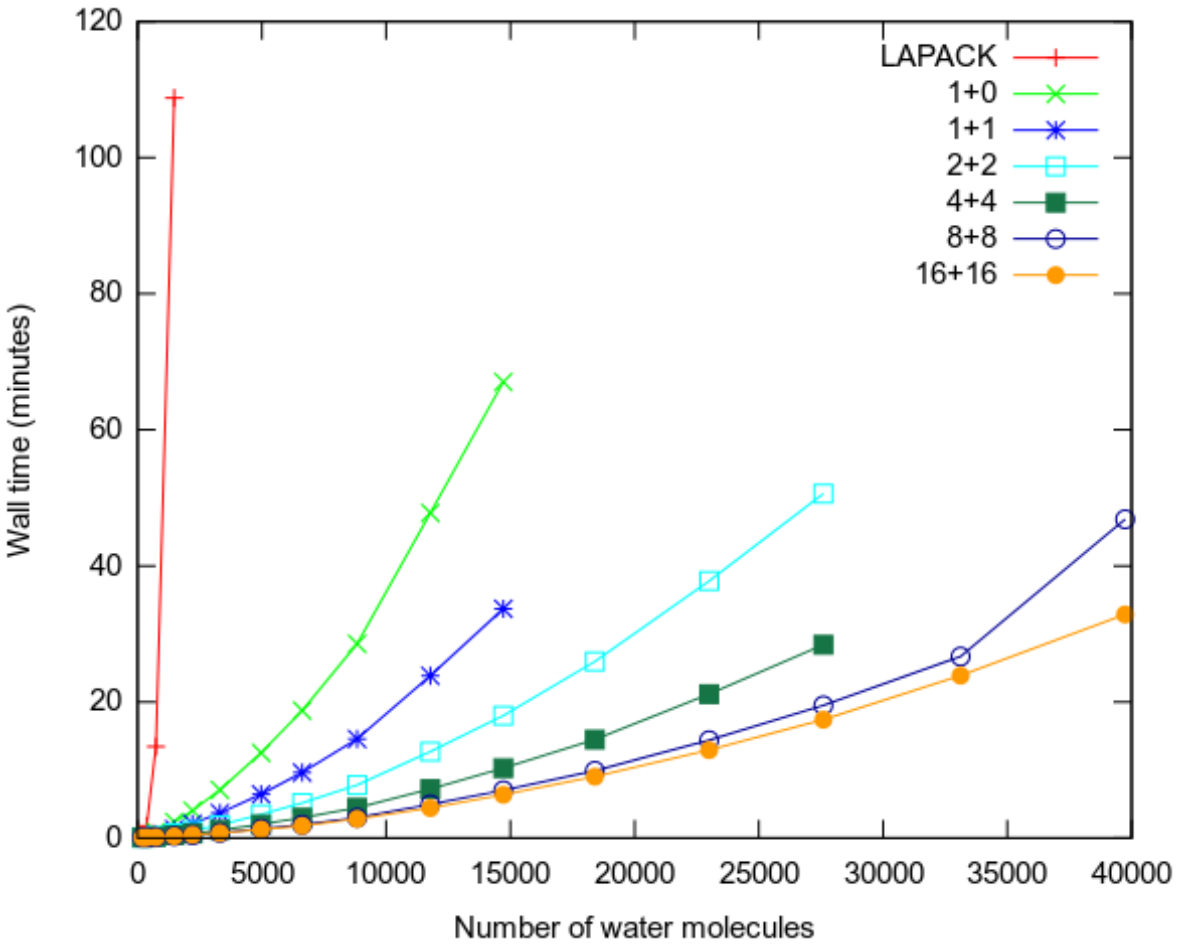
Numerical precision

The relative error $\frac{E-E_{\text{LAPACK}}}{E_{\text{LAPACK}}}$ is below 10^{-9}:

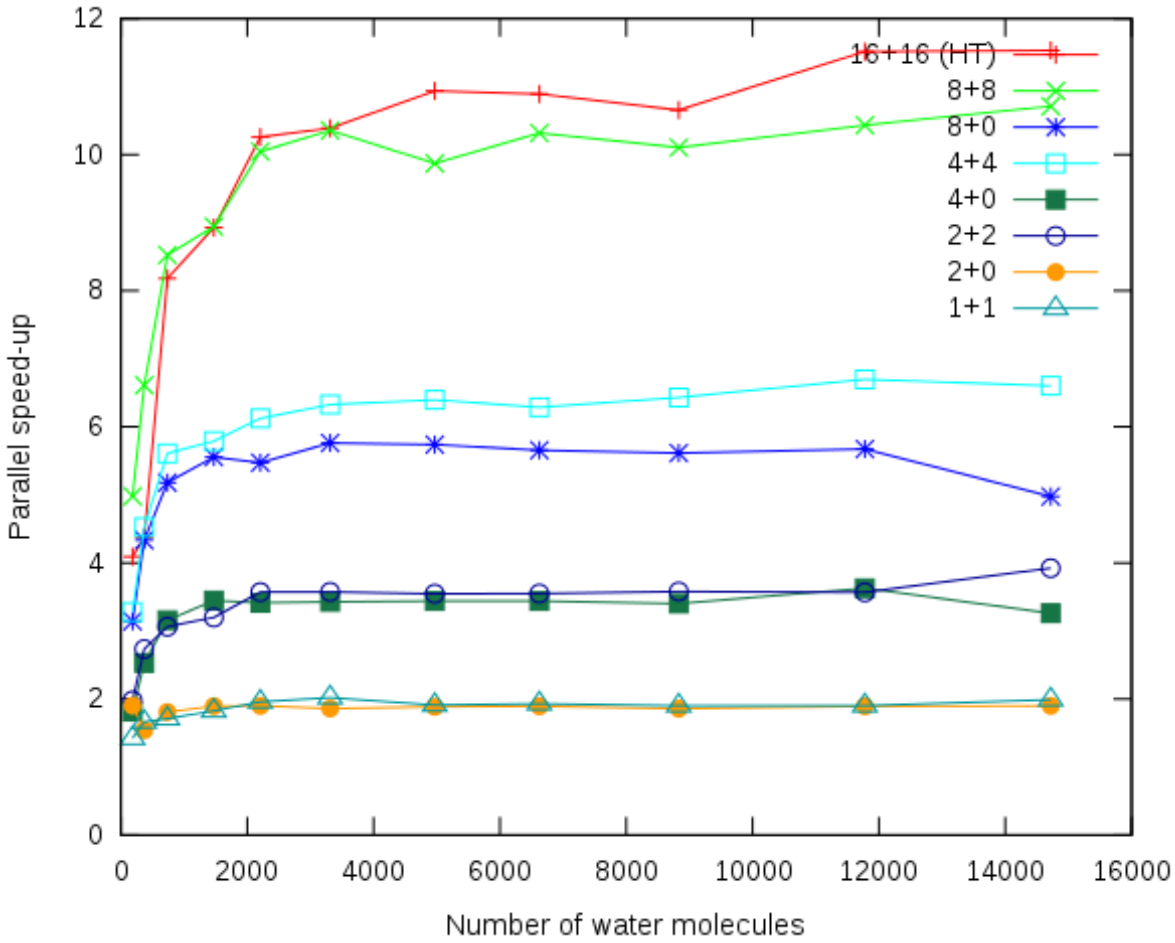
#	LAPACK	This work	Relative Error
184	-749.6393870**2**	-749.6393871**1**	1.2 10^{-10}
368	-1499.3316349**2**	-1499.3316348**3**	0.6 10^{-10}
736	-2998.7423716**6**	-2998.7423721**1**	1.5 10^{-10}
1472	-5997.7826**7988**	-5997.7826**8084**	1.6 10^{-10}

The sparse cut-off criterion is adjusted with the target SCC convergence criterion ϵ:
our numerical errors are always lower than $\epsilon/10$

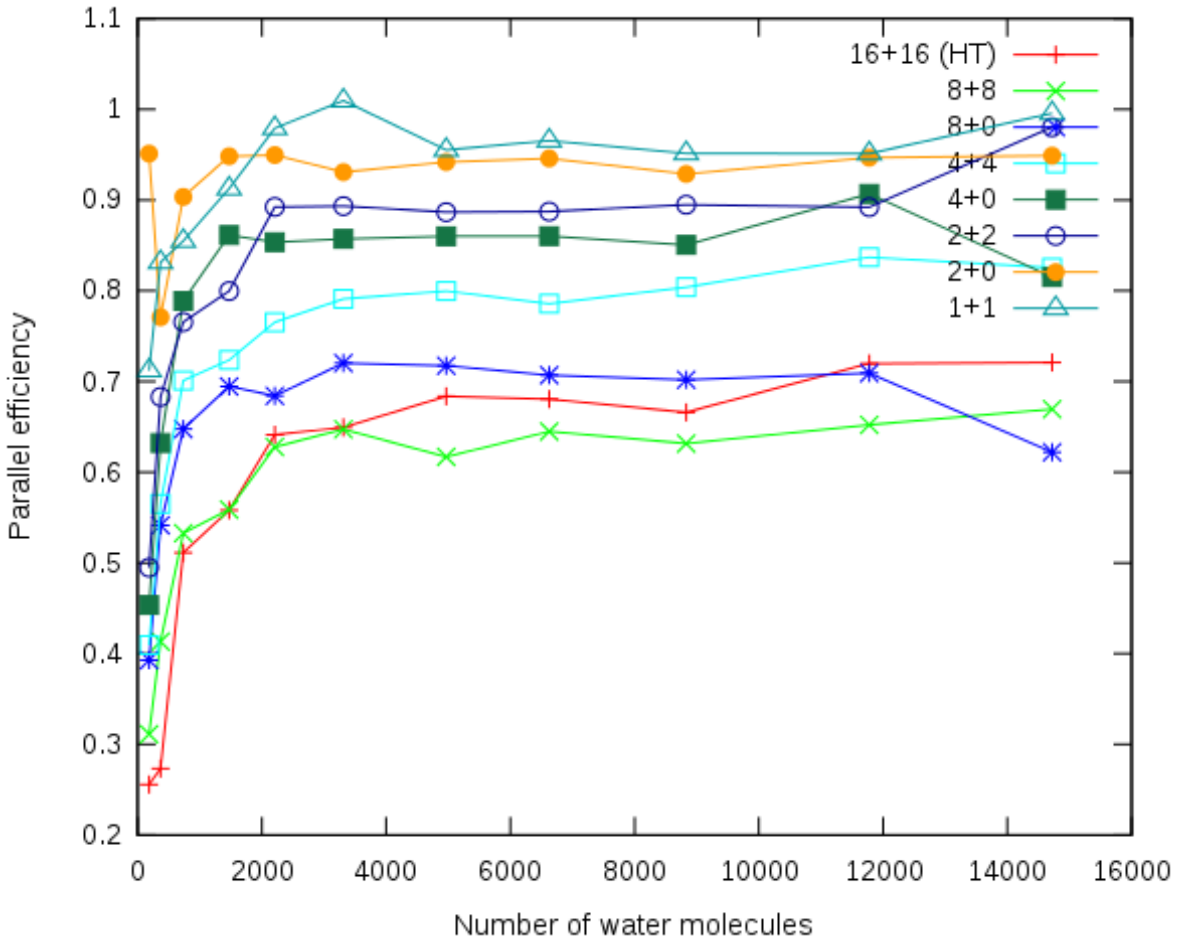
Total wall time



Parallel speed-up



Parallel efficiency



Computational efficiency

- Orthonormalization: 18.5 GFlops/s DP (peak: 345.6 GFlops/s), L3: 83 GiB/sec (RAM peak: 102 GiB/sec)
- Partial diagonalization: 16.5 GFlops/s SP (peak: 691.2 GFlops/s), L3: 64 GiB/sec
- Memory latency -bound (50–90 ns)
- Memory consumption: $\sim$ 1 MiB /water molecule

More results

CALMIP SGI Altix UV: (48 sockets E7-8837 @ 2.67GHz, 8 cores/socket, 3Tb RAM)
(NUMA, RAM latencies: 90– > 1000 ns)
Results with 128 cores

- 184 000 molecules : 3.0 hours
- 244 904 molecules : 5.5 hours
- 317 952 molecules (1 million atoms!) : 9.0 hours