Millions of atoms in DFTB

Anthony Scemama[1] <scemama@irsamc.ups-tlse.fr>
Mathias Rapacioli[1]
Nicolas Renon[2]

[1] Labratoire de Chimie et Physique Quantiques
IRSAMC (Toulouse)
[2] CALMIP (Toulouse)

Large molecular systems:

- No chemistry can be done with a single point: need to calculate the energy for multiple geometries (dynamics, Monte Carlo, ...)
- Memory grows as $\mathcal{O}(N^2)$
- Computational complexity grows as $\mathcal{O}(N^3)$
- Energy differences represent a very small fraction of the total energy -> approximations should be carefully controlled

The calculation of the energy needs to be *dramatically* accelerated.

Linear-scaling techniques are a very common and efficient solution:

- Linear-Scaling in storage : going from $\mathcal{O}(N^2)$ to $\mathcal{O}(N)$
- Linear-Scaling in operations : going from $\mathcal{O}(N^3)$ to $\mathcal{O}(N)$

DFTB is well adapted to linear scaling techniques:

- No integrals to compute
- Minimal basis set (no diffuse functions)
- Larger systems are more sparse

Linear scaling DFTB is not new, and is already available:

- DFTB+ : Divide and Conquer (Yang *et al*, 1991)
- CP2K : Matrix sign function
- ADF : Density-matrix based method

Reducing flops is not necessarily good

```
do j=1,n
 do i=1,j
  dist1(i,j) = X(i,1)*X(j,1) + X(i,2)*X(j,2) + X(i,3)*X(j,3)
 end do
end do

do j=1,n
 do i=j+1,n
  dist1(i,j) = dist1(j,i)
 end do
end do
```

t(n=133) = 13.0 μs, 3.0 GFlops/s
t(n=4125) = 95.4 ms, 0.44 GFlops/s (Large is 6.8x less efficient)

```
do j=1,n
 do i=1,n ! <-- 2x more flops!
  dist3(i,j) = X(i,1)*X(j,1) + X(i,2)*X(j,2) + X(i,3)*X(j,3)
 end do
end do
```

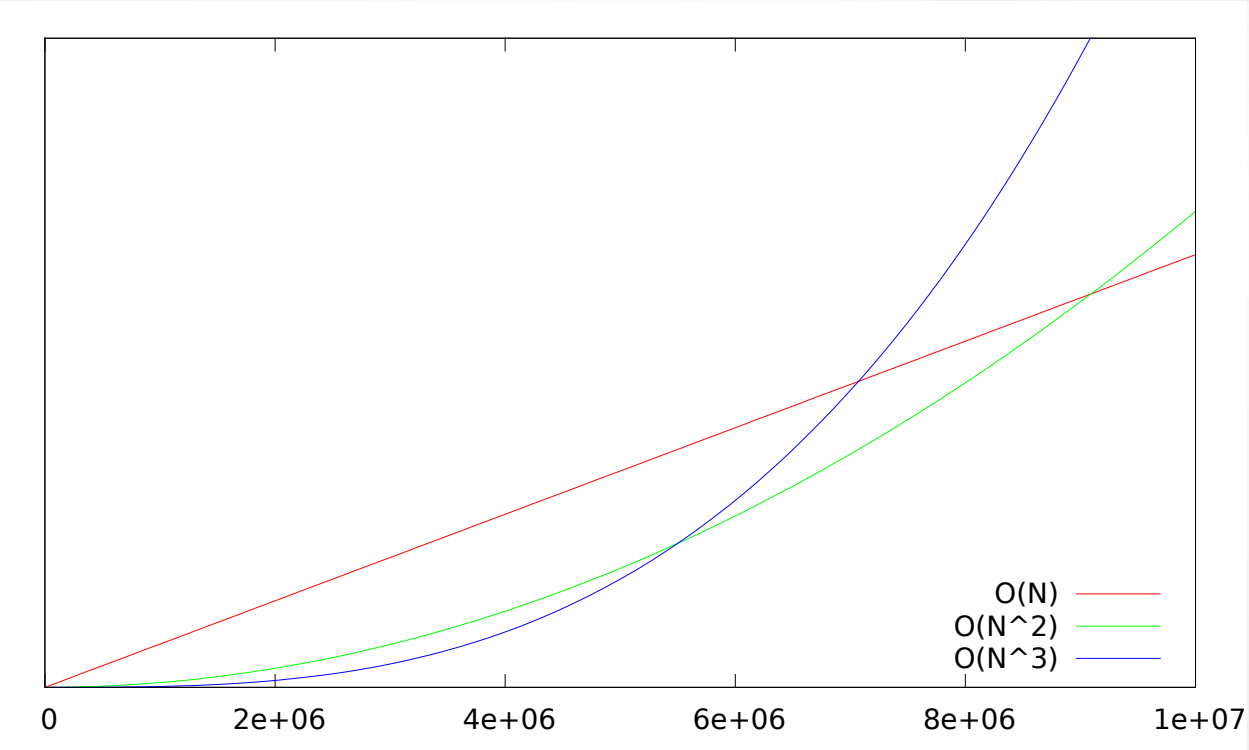
t(n=133) = 10.3 μs : 1.26x speed up, 8.2 GFlops/s
t(n=4125) = 15.7 ms : 6.07x speed up, 5.4 GFlops/s (Large is 1.5x less efficient)

With data aligned on a 256-bit boundary using compiler directives:

t(n=133) = 7.2 μs : 1.80x speed-up, 12.1 GFlops/s
t(n=4125) = 15.5 ms : 6.15x speed-up, 7.5 GFlops/s. (Large is 1.6x less efficient)

WARNING

Linear-scaling is a limit when *N* goes to infinity. What matters is the **wall time** in the useful range, and the control of the approximations.



Difficulties arising with Linear scaling:

- Computers are better at making flops than moving data in memory
- Reduction of the **arithmetic intensity** (nb of operations per loaded or stored byte) -> the bottleneck becomes **data access**
- Data access is never uniform (different levels of cache, hardware prefetching, etc) : NUMA (Non Uniform Memory Access)
- Scaling curves are linear *only* if the data access is **uniform** (uniformly good or uniformly bad)

Goal of this work

- Accelerate DFTB. Whatever the scaling, it has to be fast!
- OpenMP implementation: It will be so fast that MPI will be the bottleneck
- Large simulations have to fit in memory
- Results should be trusted when the $\mathcal{O}(N^3)$ calculation can't be done
- Small simulations should not suffer from the optimizations for large systems

Two reasons for "million of atoms" in the title:

- An impressive title for this presentation
- Test our implementation for much larger systems than needed

Long term project:

- Investigate DNA hairpins in water
- Add a layer of distributed parallelism for massively parallel Monte Carlo simulations

Outline

1. Presentation of the algorithm
2. Hardware considerations
3. Technical implementation details
4. Benchmarks on boxes of liquid water

SCF Algorithm in DFTB

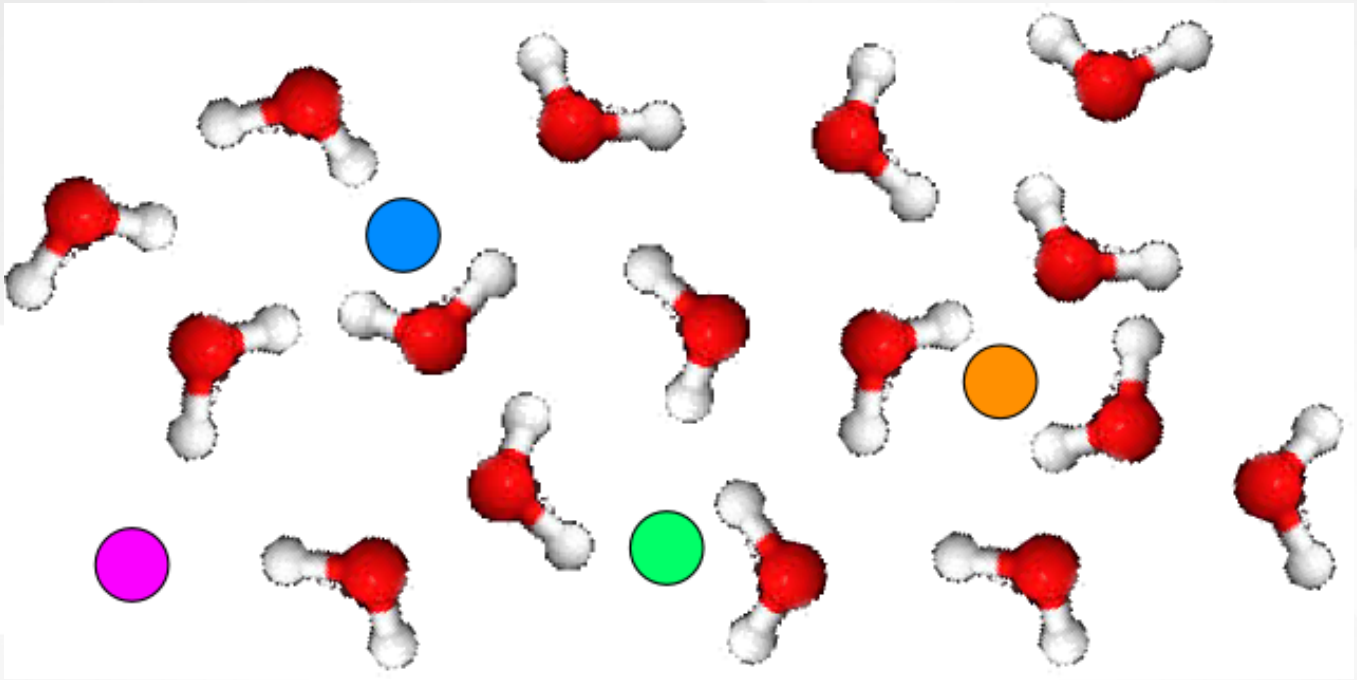
The choice of the algorithm was not driven by the reduction of flops, but on the possibility of the hardware to be efficient at doing the calculation, even for medium-sized systems. We chose a MO-based formalism.

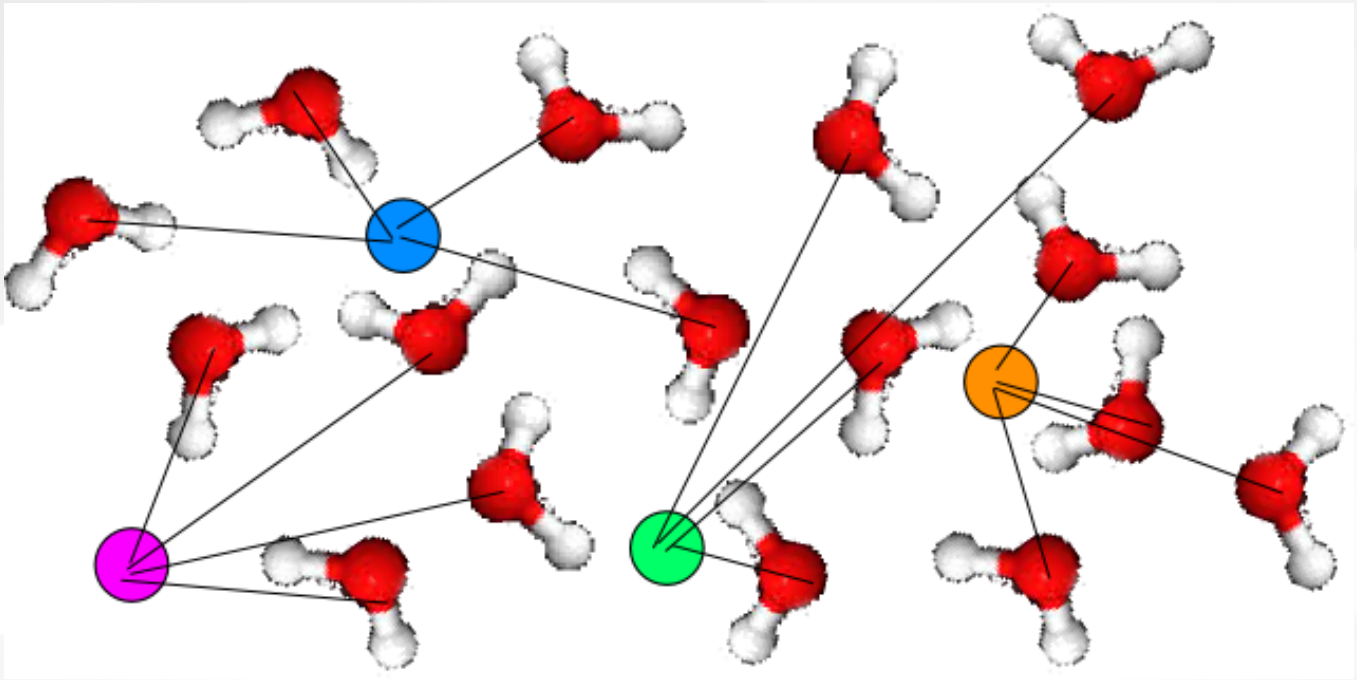
1. Generate an initial guess of *local* orbitals
2. Re-order the orbitals in packets of spatially close orbitals
3. Orthonormalize the guess
4. SCF steps
    - Instead of diagonalizing $\mathbf{H}$, only cancel the occupied-virtual block (Brillouin's theorem, Stewart *et al*)
    - At every SCF iteration, the occupied-virtual block is *approximately* canceled
    - At convergence, the occupied-virtual block is zero to a given numerical precision
5. Re-orthonormalize MOs before computing the final energy

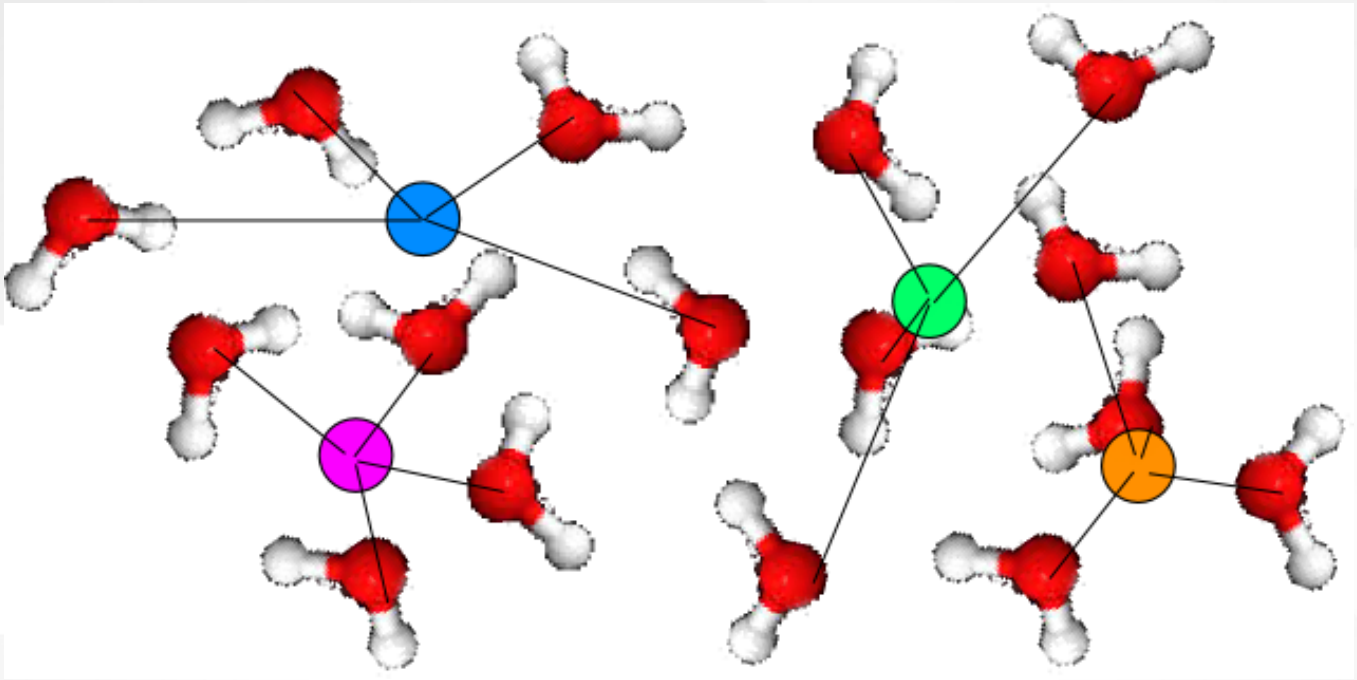
Initial guess

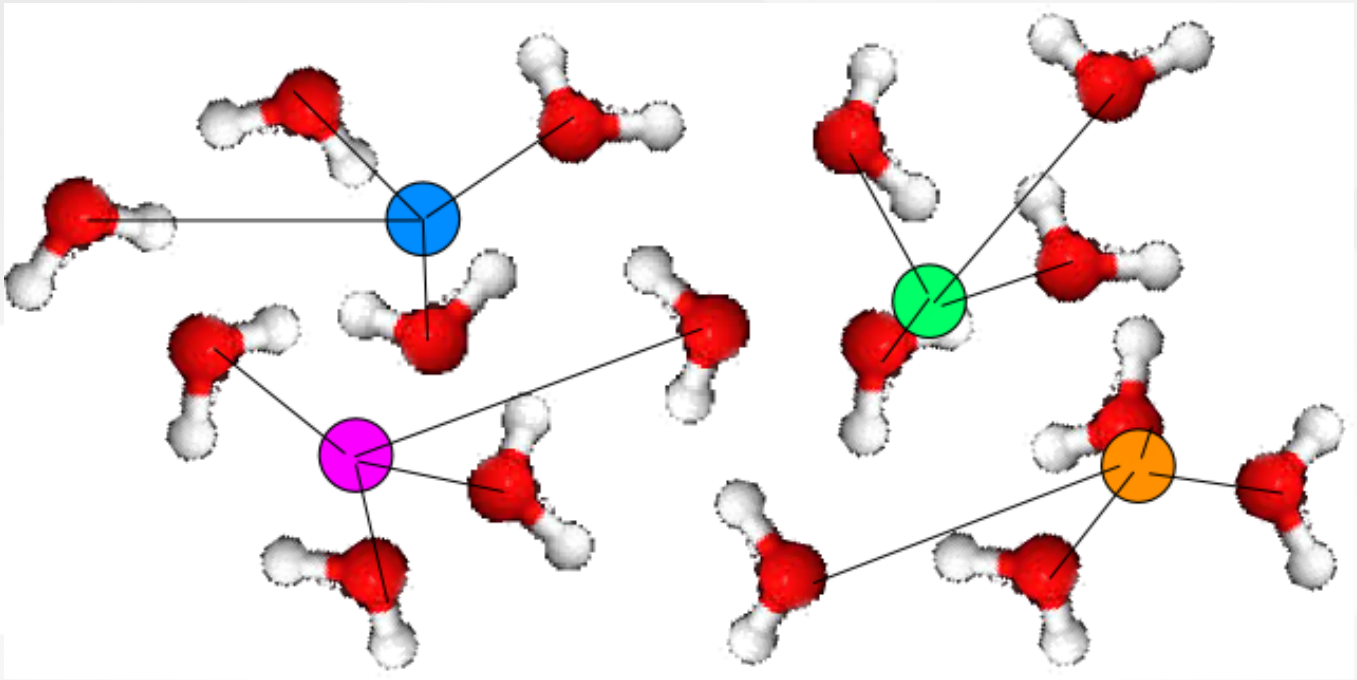
The system is partitioned using a constrained variant of the k-means clustering algorithm:

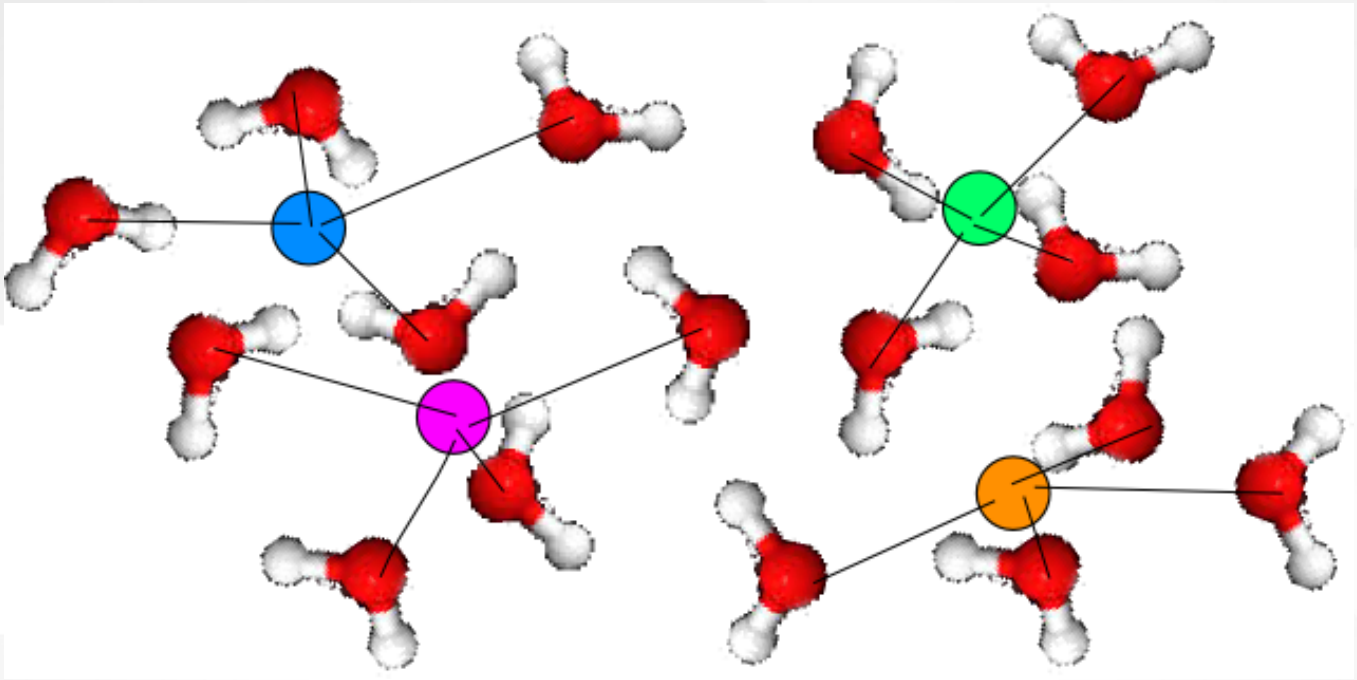
1. A set of m centers (means) is first distributed evenly in the 3D-space
2. Each fragment is attached to its closest center, such that each center is connected to 4 molecules
3. The position of the centers is moved to the centroid of the connected fragments
4. Go back to step 2 until the partition doesn’t 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
- A non-SCC DFTB calculation is performed in parallel on each fragment
- All occupied MOs are packed together
- All virtual MOs are packed together

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.

This step takes less than 1% of the total wall time.

Orthonormalization of the MOs

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

1. The $\mathbf{C}$ matrix is already stored sparse
2. Compute $\mathbf{S}$ (sparse) for each connected k-means groups (neighbours)
3. Compute $\mathbf{C}^\dagger \mathbf{S}\mathbf{C}$ (sparse)
4. Normalize using diagonal elements
5. Perform 1st order 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

- $\mathbf{S}$ is calculated on the fly with $\mathbf{C}^\dagger \mathbf{S}\mathbf{C}$
- Orthonormalization takes 40-50% of the total wall time.
- Bottleneck: $\mathbf{C}^\dagger \mathbf{S}\mathbf{C}$

Parallel implementation of orbital rotations

Difficulty: each MO can be rotated only by one thread at at a time.

- First, dress the list of rotations to do.
- 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*. If not possible, go to step 1
  4. Try to take lock *j*
  5. If not possible, free lock *i* and go to step 1
  6. Rotate *i* and *j* and mark (*i*, *j*) as done
  7. Free locks *i* and *j*
  8. Go back to step 1 until all rotations are done

SCF steps

Partial diagonalization of $\mathbf{C}^\dagger \mathbf{H}\mathbf{C}$

1. The $\mathbf{C}$ matrix is already stored sparse
2. Compute $\mathbf{H}$(sparse)
3. Compute $\mathbf{C}^\dagger \mathbf{H}\mathbf{C}$ (sparse)
4. Perform exact Jacobi rotations to preserve orthonormality, but with approximate angles: $\mathbf{C}^\dagger \mathbf{H}\mathbf{C}$ is not updated after a rotation
5. Go back to step 3 until the largest off-diagonal element is below a threshold

- $\mathbf{H}$ is calculated on the fly with $\mathbf{C}^\dagger \mathbf{H}\mathbf{C}$

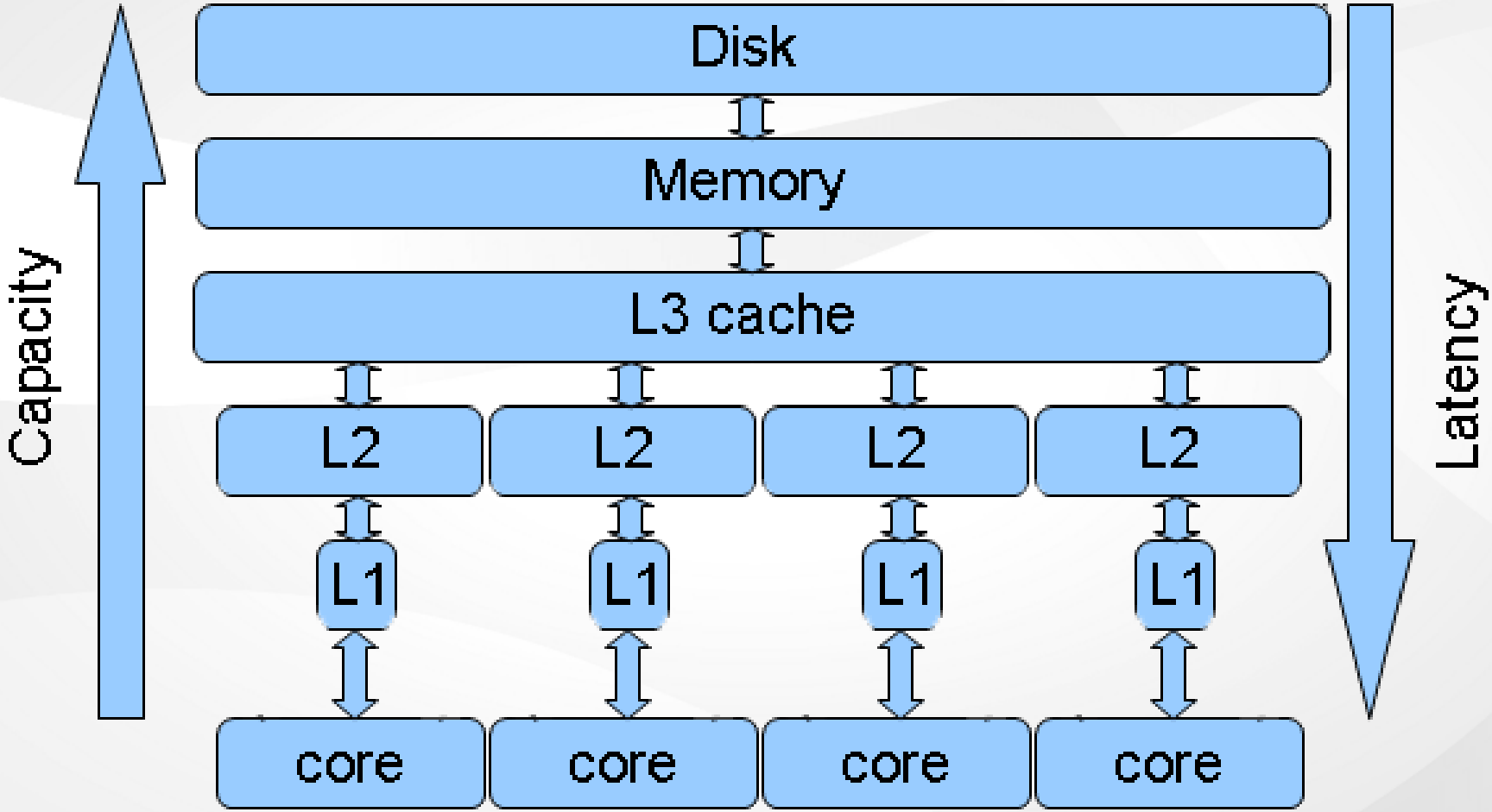
This step can be safely performed in single precision

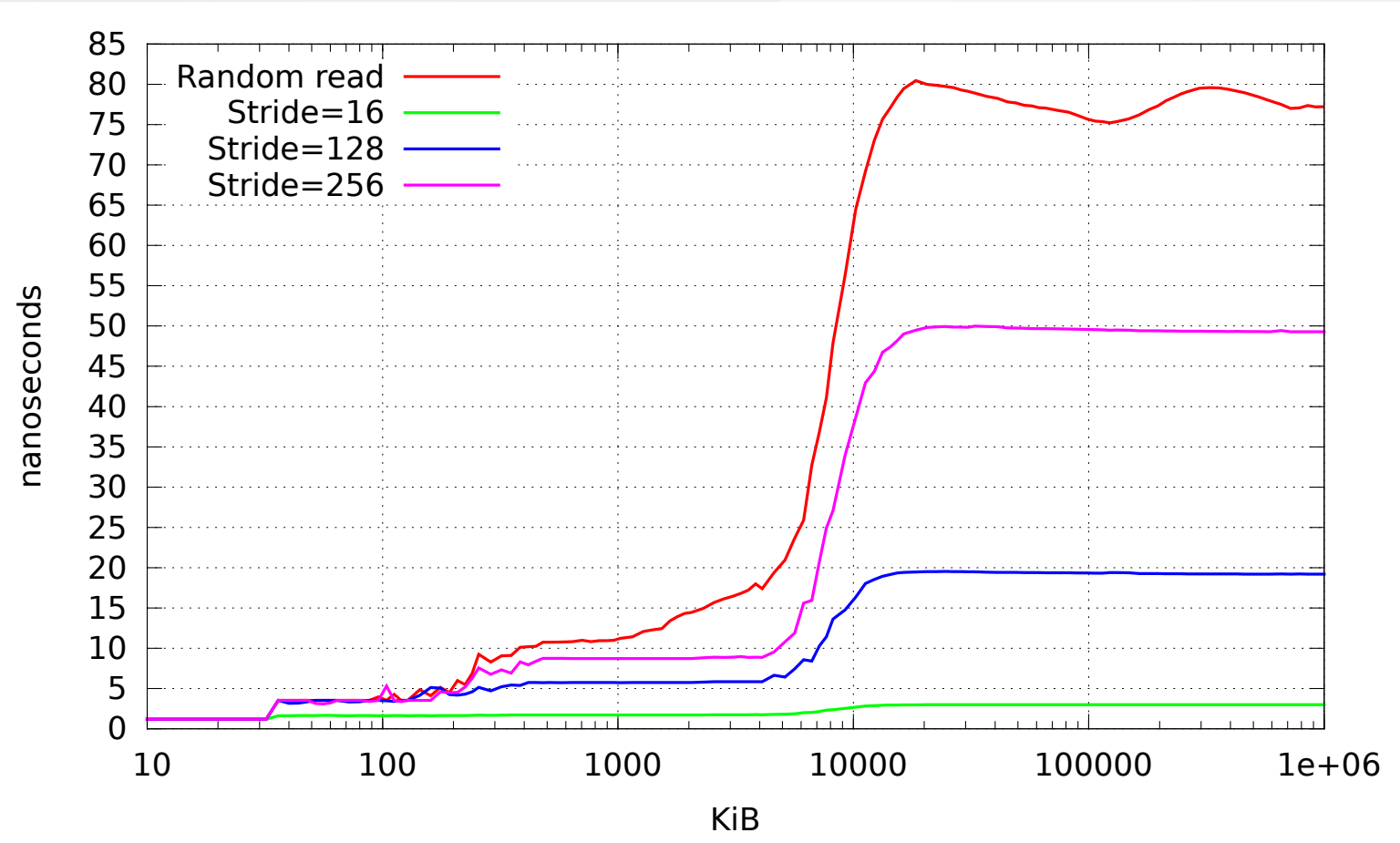
- data is 2x smaller, so data bandwidth is virtually 2x larger
- caches contain 2x more elements
- vectorization is 2x more efficient.

- Exact Jacobi rotations are performed to preserve orthonormality, but with approximate angles: $\mathbf{C}^\dagger\mathbf{HC}$ is not updated after a rotation.
- Rotations are done only in the occupied-virtual MO block
- Occupied MOs don't rotate between each other
- Virtual MOs don't rotate between each other
- The locality of occupied and virtual MOs is preserved

- MO rotations represent 3-6% of the total wall time
- Adding the $\mathbf{C}^\dagger\mathbf{HC}$ step yields 33-40%

General hardware considerations





Measures obtained with LMbench *

1 cycle = 0.29 ns, 1 peak flop SP = 0.018 ns

Integer (ns)	bit	ADD	MUL	DIV	MOD
32 bit	0.3	0.04	0.9	6.7	7.7
64 bit	0.3	0.04	0.9	13.2	12.9

Floating Point (ns)	ADD	MUL	DIV
32 bit	0.9	1.5	4.4
64 bit	0.9	1.5	6.8

Data read (ns)	Random	Prefetched
L1 cache	1.18	1.18
L2 cache	3.5	1.6
L3 cache	13	1.7
Memory on socket	75-80	3.

---

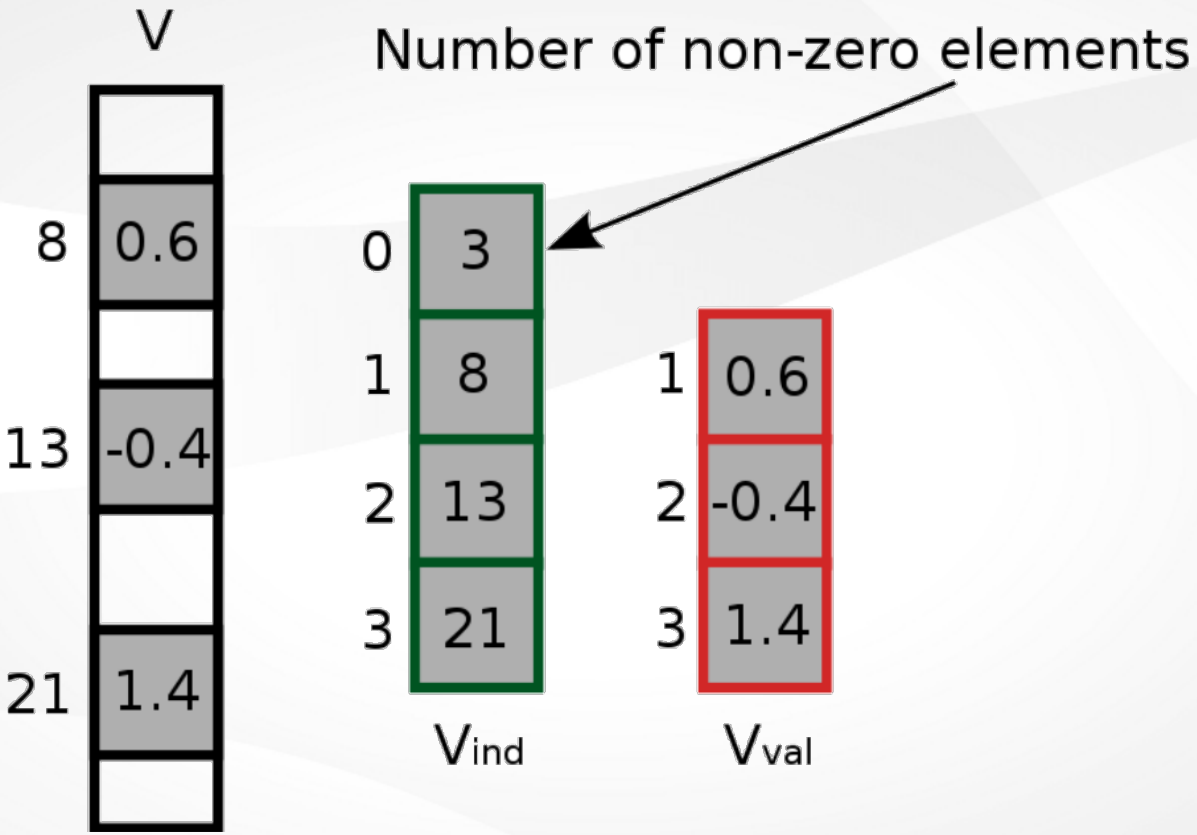
* http://www.bitmover.com/lmbench/

Strategy to optimize shared-memory access

Low aritmetic intensity : Bring the data to the CPU cores as fast as possible

- Reduce storage as much as possible, and re-compute
- Avoid thread migration and allocate/initialize memory locally in parallel regions (first-touch policy)
- Every thread uses as much as possible memory which is close
- Use stride-1 access as much as possible to benefit from prefetching
- Reuse data which is in the caches
- Avoid to synchronize threads
- Static scheduling keeps the access to close memory and avoids thread communication

Sparse storage



Takes a little more memory than CSR or CSC, but:

- Columns can be easily expanded or contracted

- Arrays are 256-bit aligned :
  - optimized data access
  - enables vectorization of small data fragments
- Leading dimension of V_{val} is fixed : $LD = \alpha \times 512 + 24$
- $\alpha \times 512$
  - each column is 256 bit-aligned and starts at the beginning of a cache line
  - two columns start on distinct memory pages : prefetch only the columns
- 24 : Avoid 4k aliasing (round-robin over the cache lines)
- V_{ind} has LD+1 elements (start at zero) : reduced 4k aliasing

Total memory usage:

- 982 GiB for 504 896 water molecules (1.5 million atoms)
- 680 KiB per atom

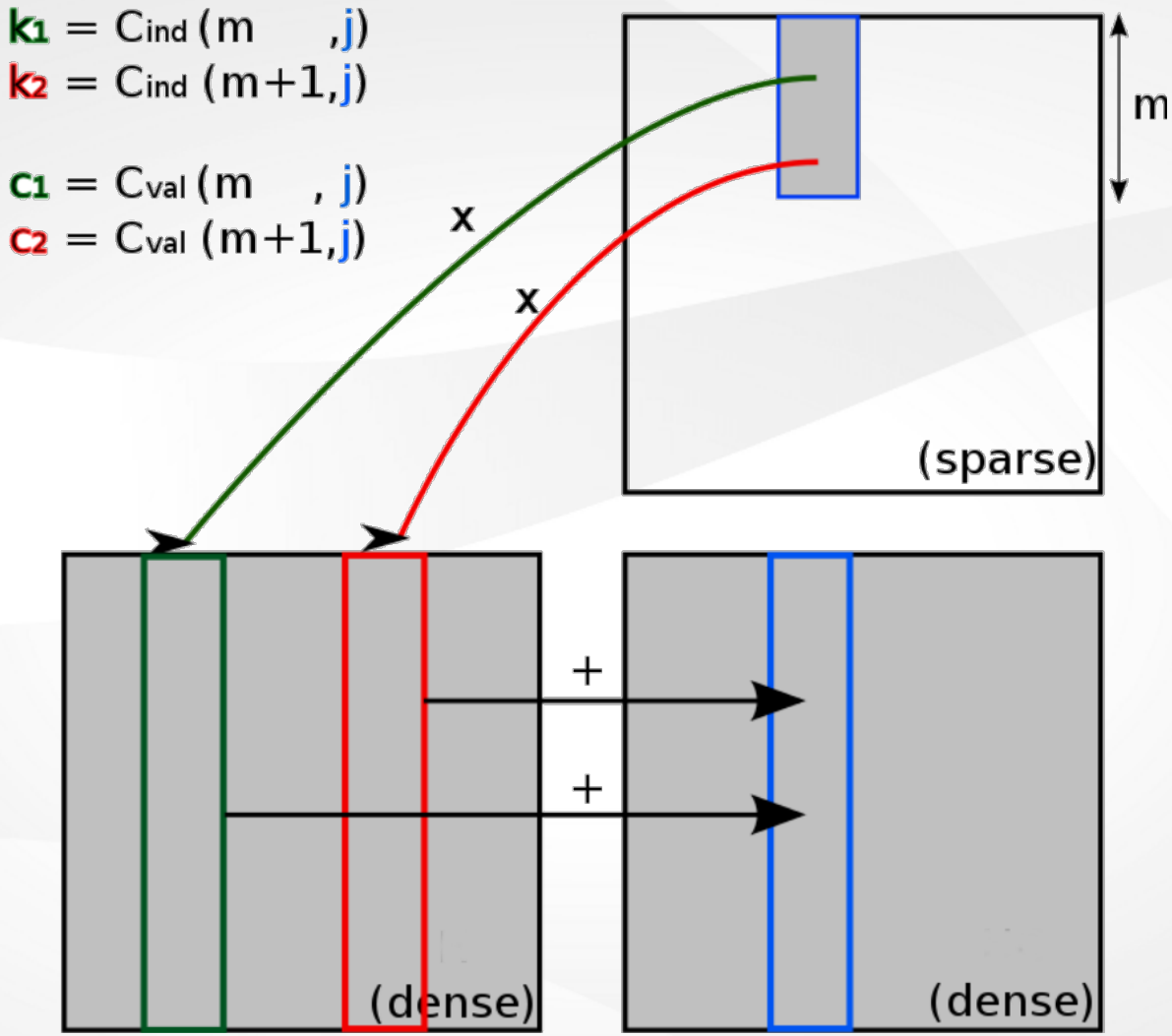
Dense x Sparse Matrix Product from QMC=Chem

$\mathcal{O}(N^2)$ with a very small prefactor.

Inner-most loops:

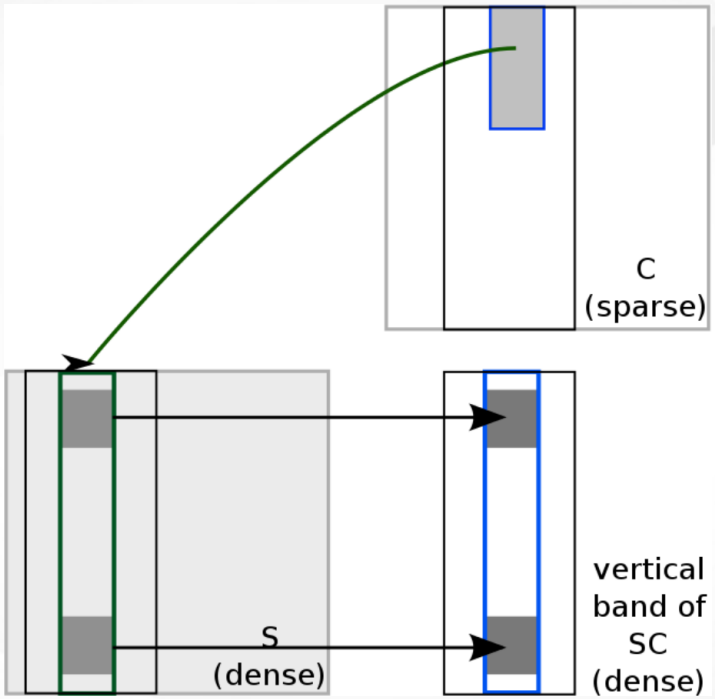
- Perfect ADD/MUL balance
- Does not saturate load/store units
- Only vector operations with no peel/tail loops
- Uses 15 AVX registers. No register spilling
- If all data fits in L1, 100% peak is reached (16 flops/cycle)
- In practice: memory bound, so **50-60% peak is measured**.

Efficient sparse x sparse matrix product in deMon-Nano: Represent one sparse matrix as a collection of small dense sub-matrices.


k1 = Cind (m ,j)
k2 = Cind (m+1,j)
c1 = Cval (m , j)
c2 = Cval (m+1,j)
x
x
m
(sparse)
+
+
(dense)
(dense)

Parallel implementation of CSC

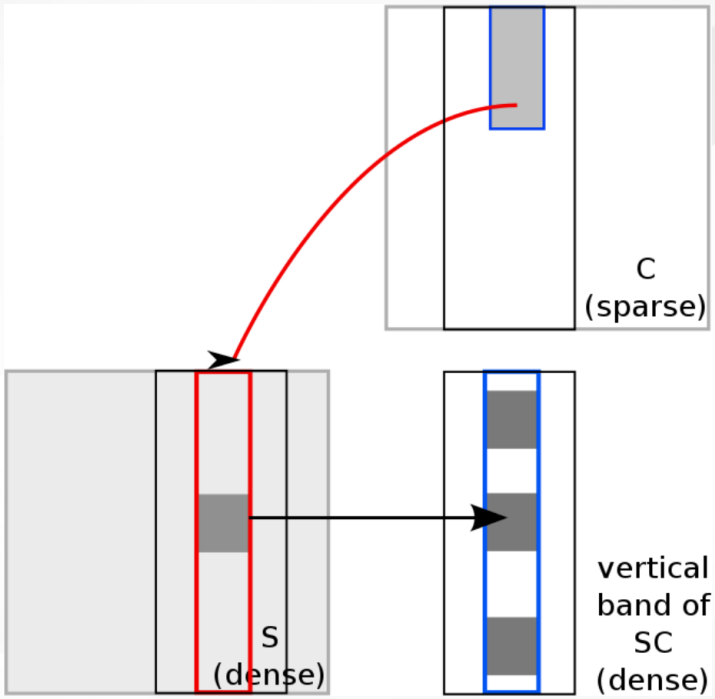
Each thread computes 16 columns of $C + SC$



Data layout is SC(16, 16, N/16) : All 16x16 matrices are 256-bit aligned and fit in L1 cache. 100% vectorized

Parallel implementation of CSC

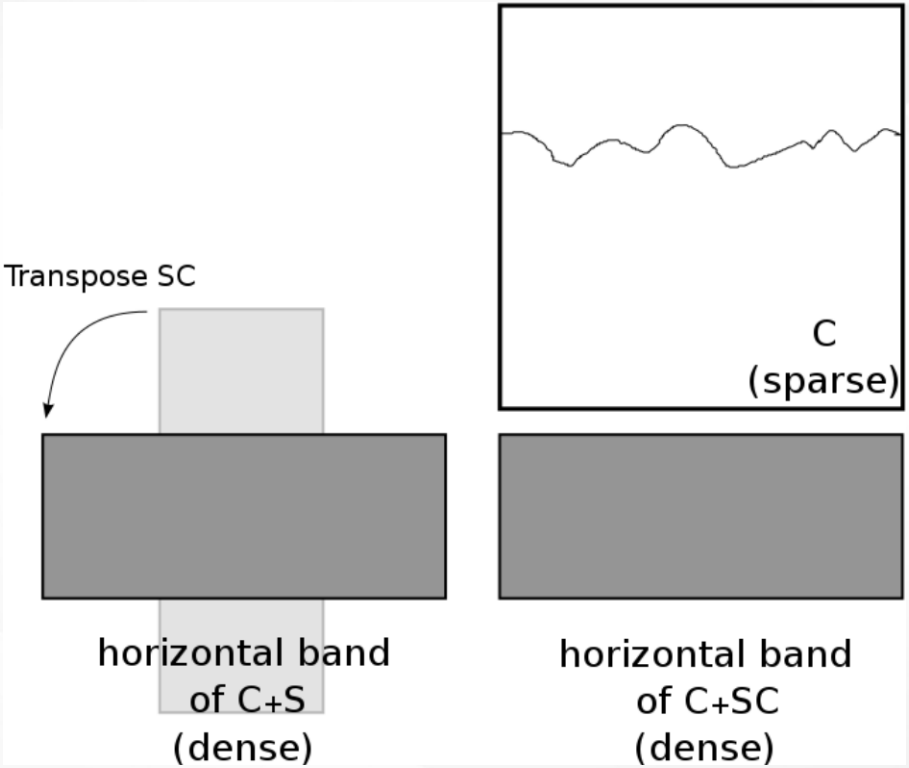
Each thread computes 16 columns of $C + SC$



Data layout is SC(16, 16, N/16) : All 16x16 matrices are 256-bit aligned and fit in L1 cache. 100% vectorized

Parallel implementation of CSC

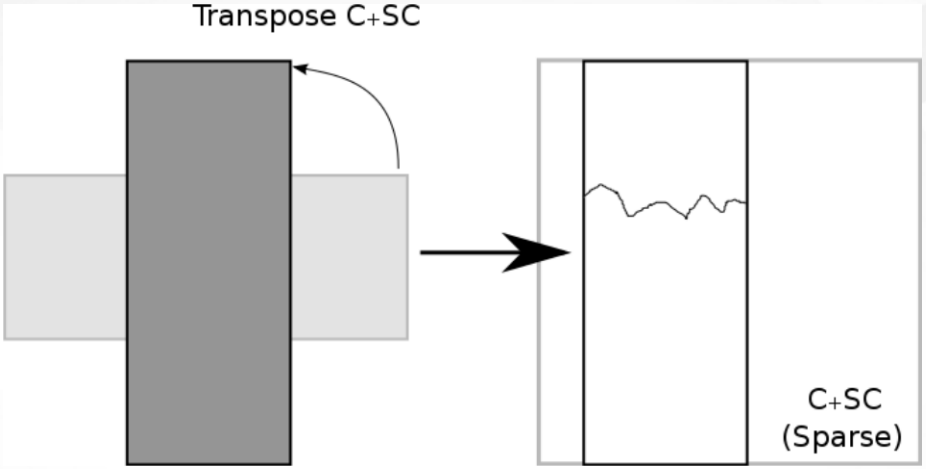
Each thread computes 16 columns of $C + SC$



Data layout is SC(16, 16, N/16) : Fast transposition in cache

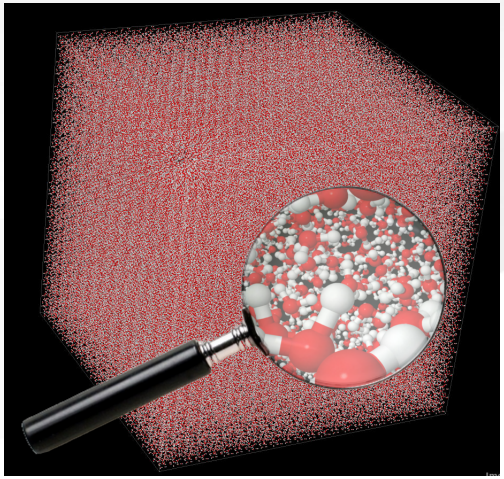
Parallel implementation of CSC

Each thread computes 16 columns of $C^\dagger SC$



Data layout is SC(16, 16, N/16) : Fast transposition in cache

Benchmarks



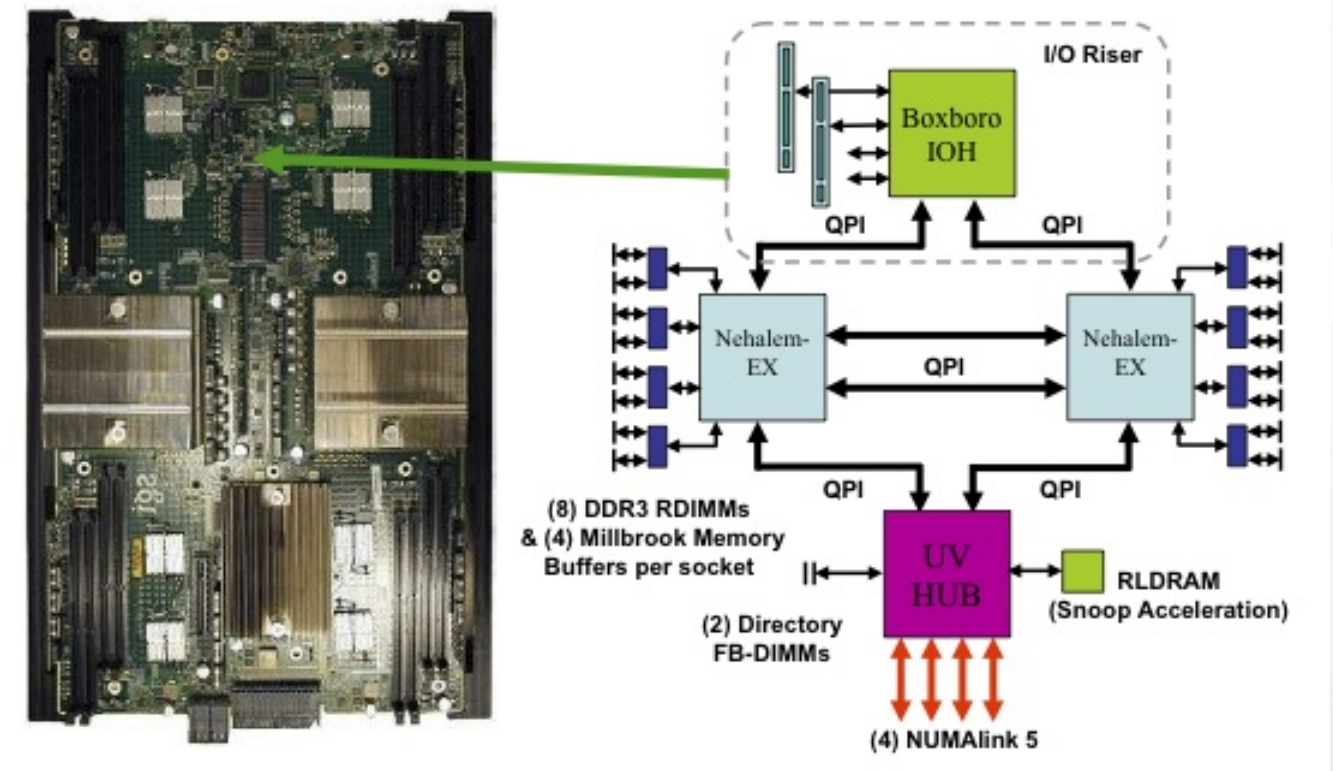
Boxes of water from 384 to 504 896 water molecules.

Two machines:

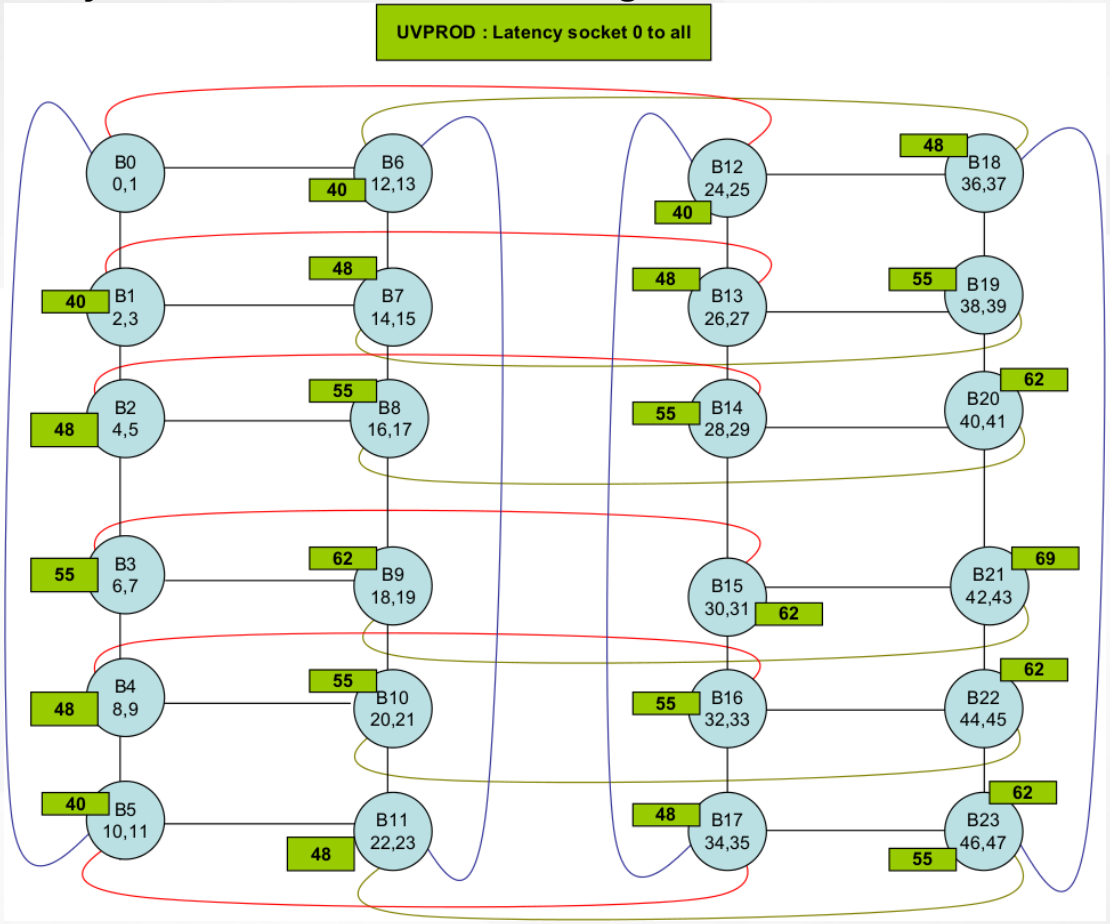
- Dual-socket Intel Xeon E5-2670, 8c @ 2.6GHz, Hyperthreading on, Turbo on, 64GiB RAM
- SGI Altix UV, large SMP machine : 384 cores and 3TiB RAM. *Enormous* NUMA effects.

SGI Altiv UV

24 blades: 2x8 cores, 128 GB RAM, connected with NUMAlink
Single OS, shared memory, 384 cores, 3 TB RAM

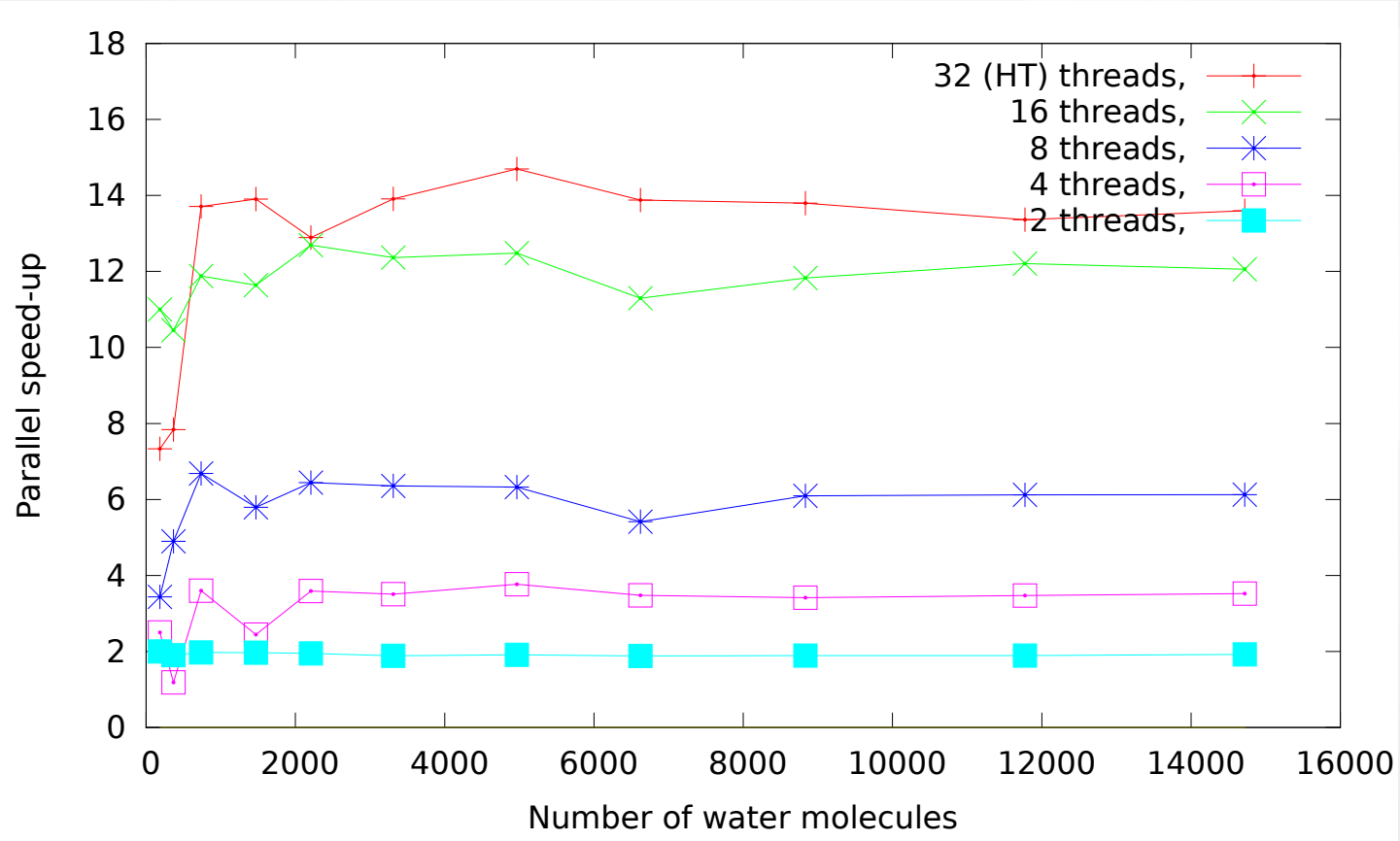


NUMAlink : (latency: 195 - 957 ns, 2-10x larger than a standard server)

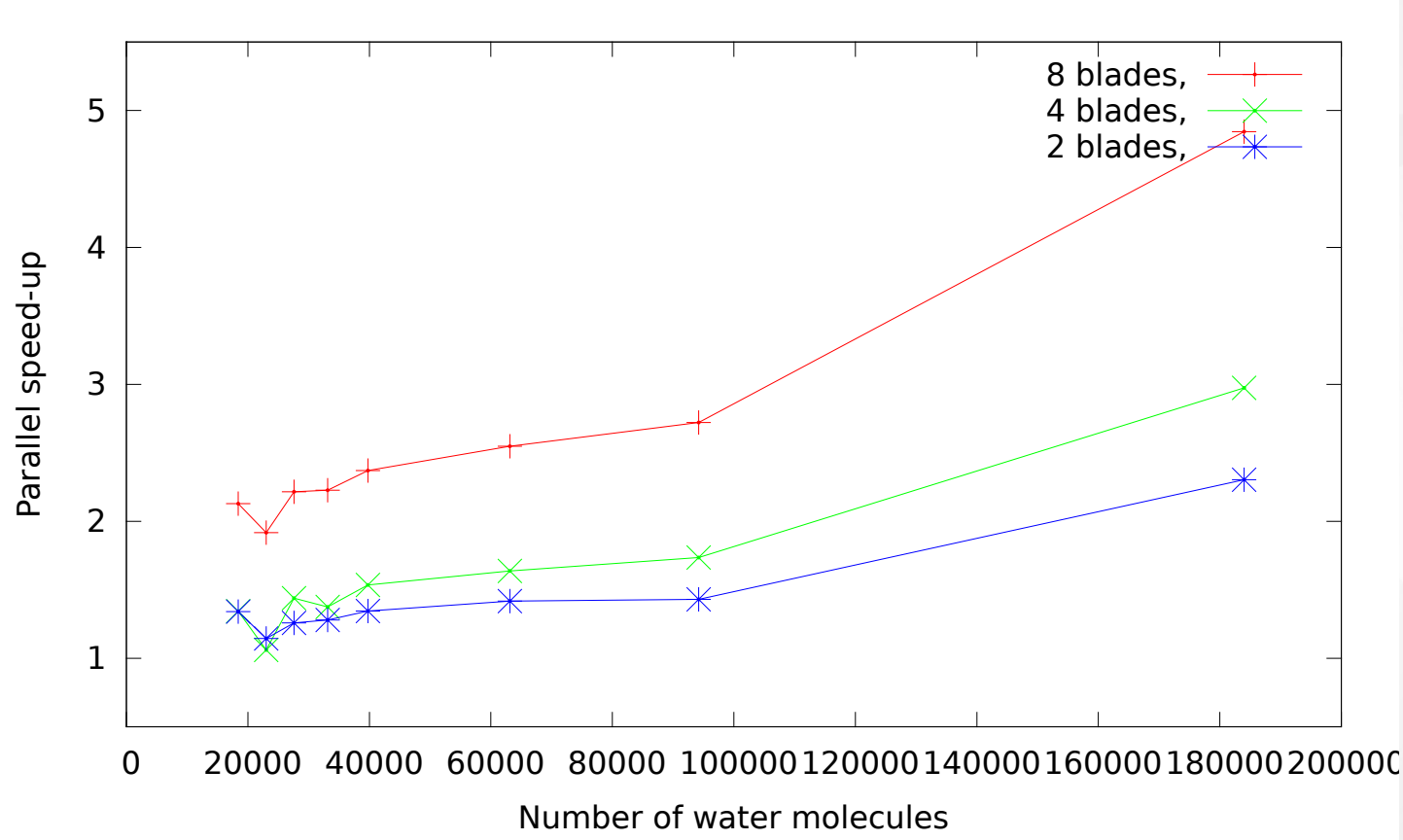


Parallel speed-up of (CXC)

Dual-socket server:

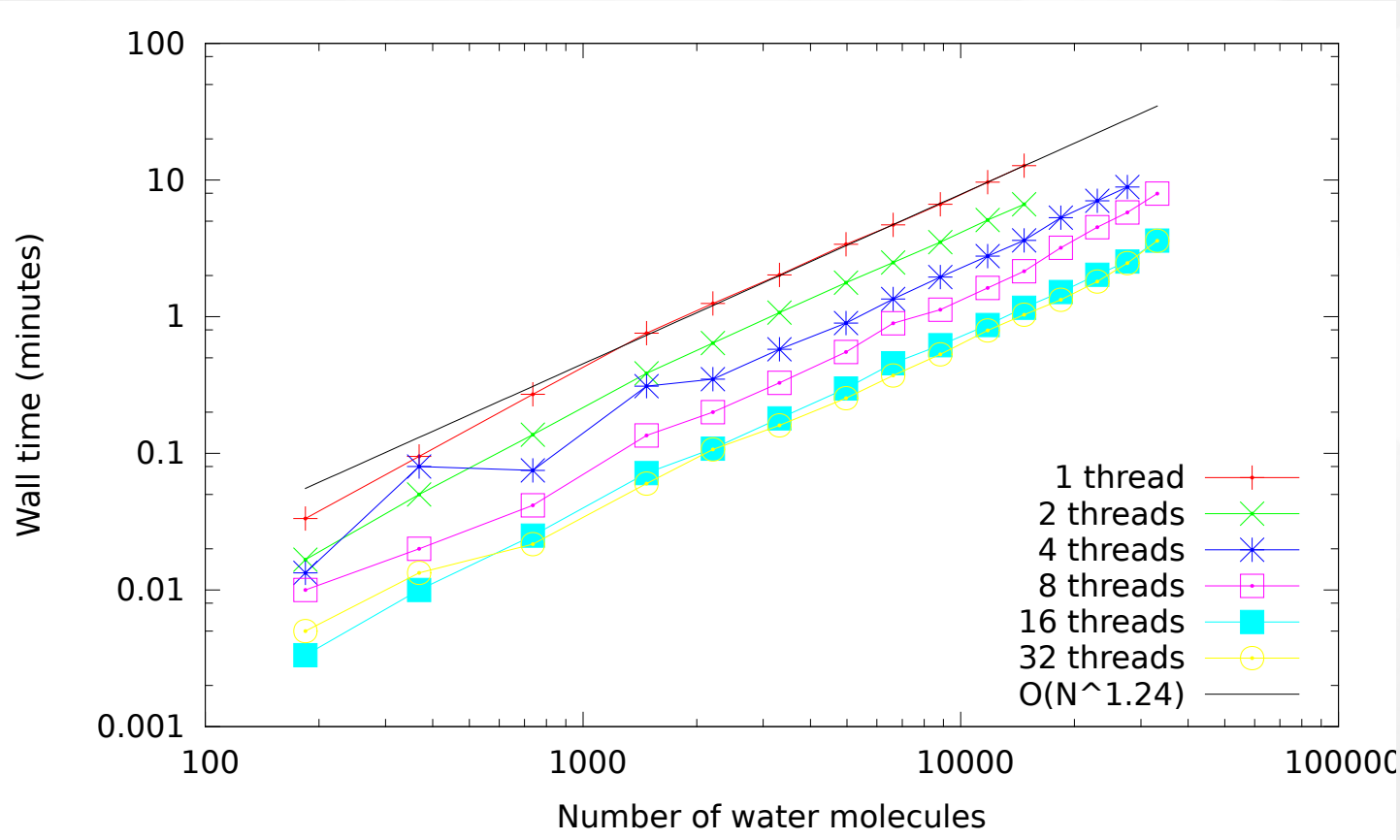


Altix-UV :

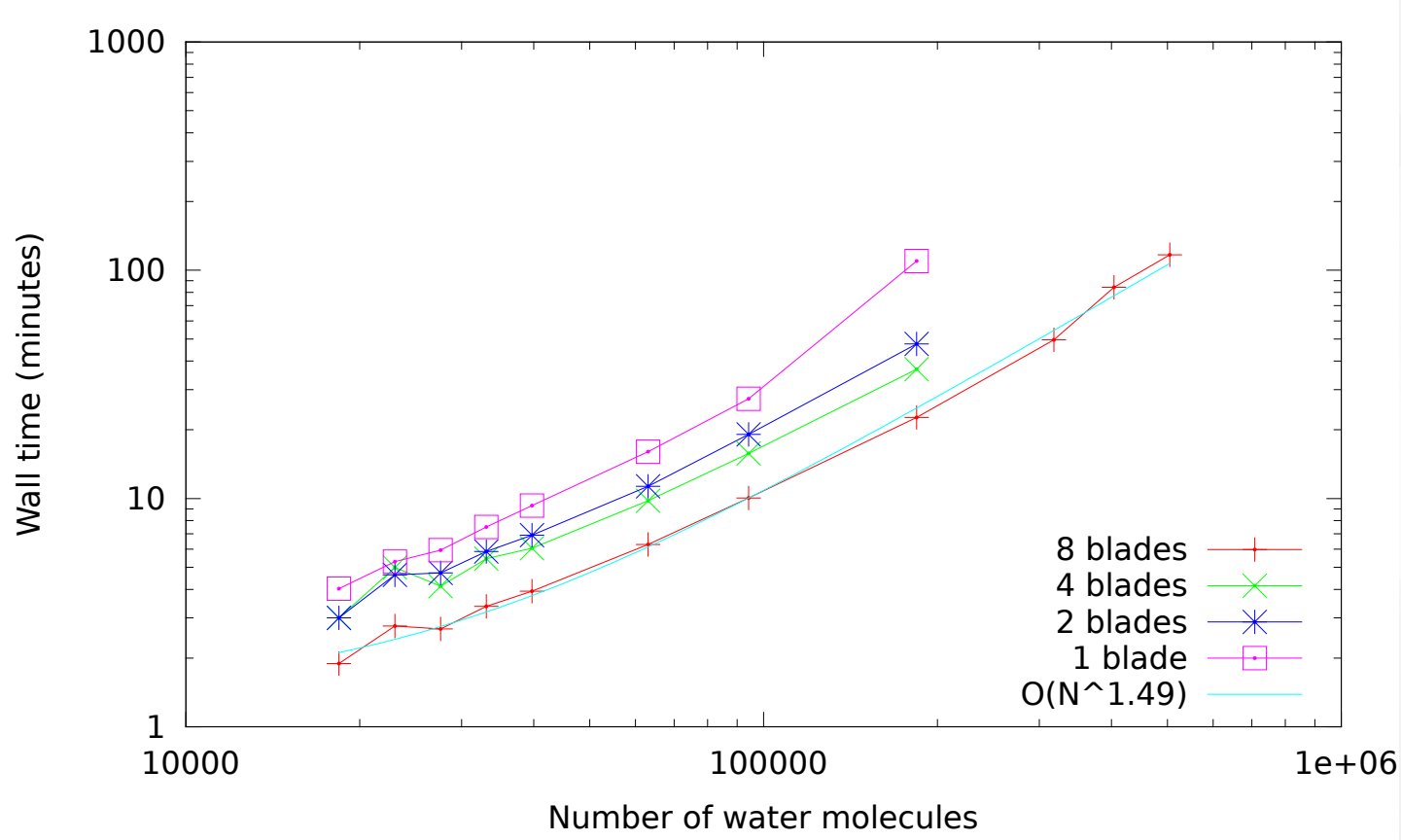

8 blades,
4 blades,
2 blades,
Parallel speed-up
1
2
3
4
5
0
20000
40000
60000
80000
100000
120000
140000
160000
180000
200000
Number of water molecules

Wall time of CXC

Dual-socket server:

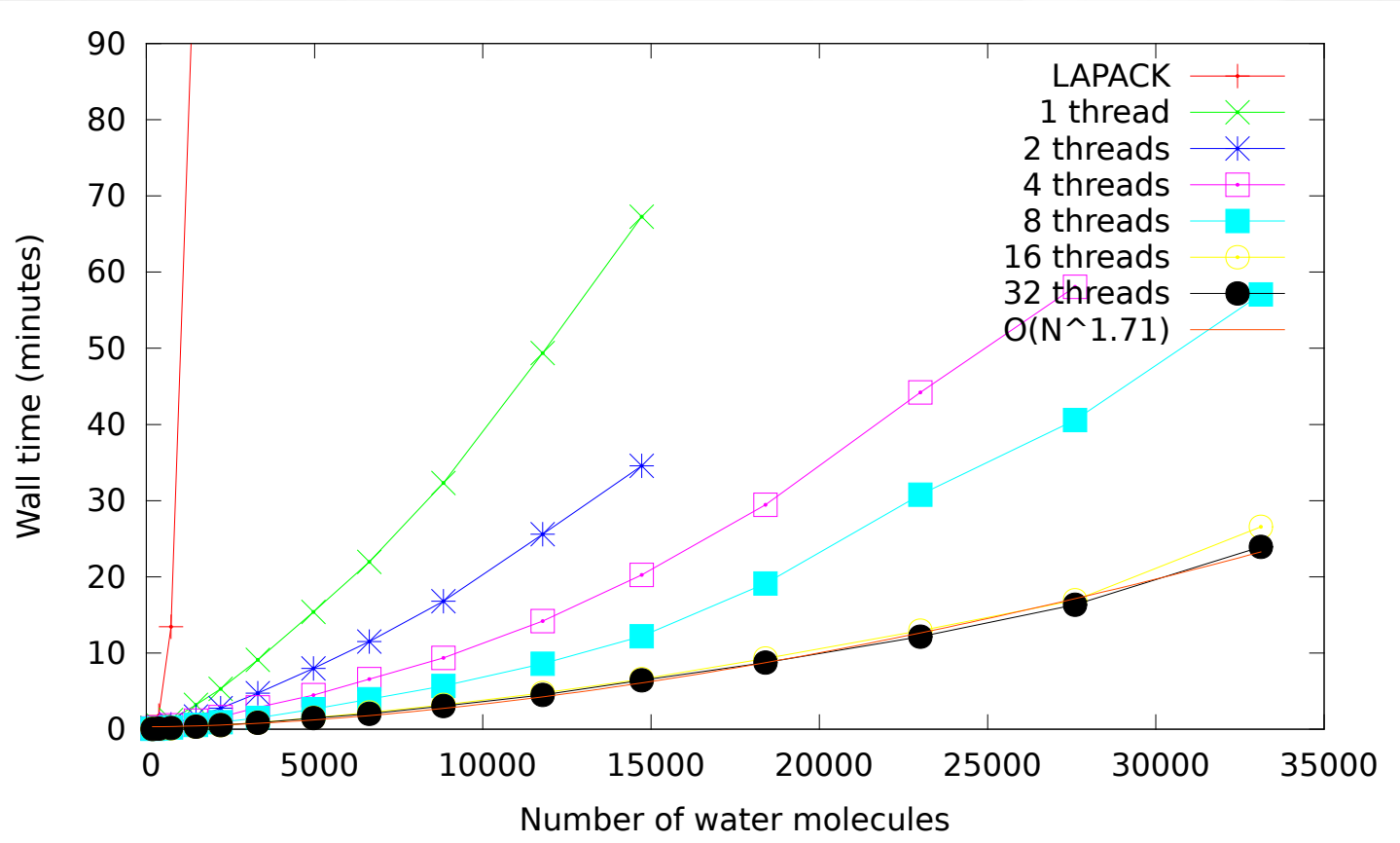


Altix-UV :

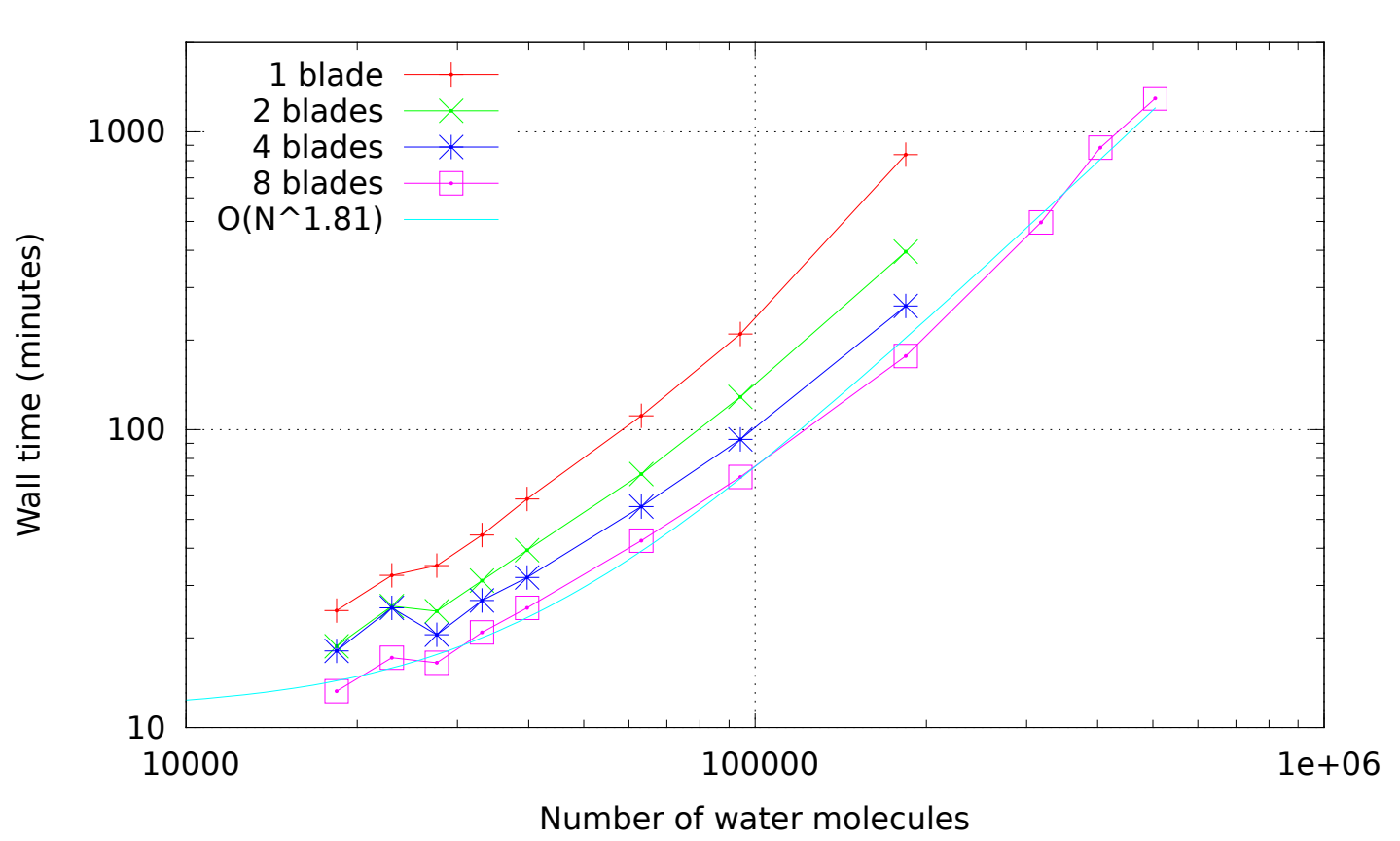

1000
100
10
1
Wall time (minutes)
10000
100000
1e+06
Number of water molecules
8 blades
4 blades
2 blades
1 blade
O(N^1.49)

Global scaling

Dual-socket server



Altix-UV:


1 blade
2 blades
4 blades
8 blades
O(N^1.81)
1000
100
10
Wall time (minutes)
10000
100000
1e+06
Number of water molecules

Observed scaling is not linear:

- The $\mathcal{O}(N^2)$ behavior comes from the $\gamma_{\alpha\xi} \sim 1/|R_\alpha - R_\xi|$ terms in the Hamiltonian:

$$H_{\mu\nu} = H^0_{\mu\nu} + \tfrac{1}{2} S_{\mu\nu} \sum_{\xi}^{N_{\text{atoms}}} (\gamma_{\alpha\xi} + \gamma_{\beta\xi})(q_\xi - q^0_\xi)$$

- For accurate results, it is important *not* to truncate 1/R.
- Very efficiently parallelized
- However, this quadratic scaling appears for large sizes

16 cores, 100 000 atoms	10% of total time
128 cores, 1 500 000 atoms	18% of total time

- We see it because all the rest is very fast!
- Could be improved by computing only the contributions where the charges have changed

Error control

For 3312 water molecules

SCF convergence	$\epsilon = 10^{-5}$	$\epsilon = 10^{-6}$
E(Lapack)	-13495.30553928	-13495.30617832
E(deMon-Nano)	-13495.30553764	-13495.30617828
Error	1.2e-10	3.0e-12
t(Lapack) (s)	9 636.1	10 612.1
t(deMon-Nano) (s)	149.8	259.0

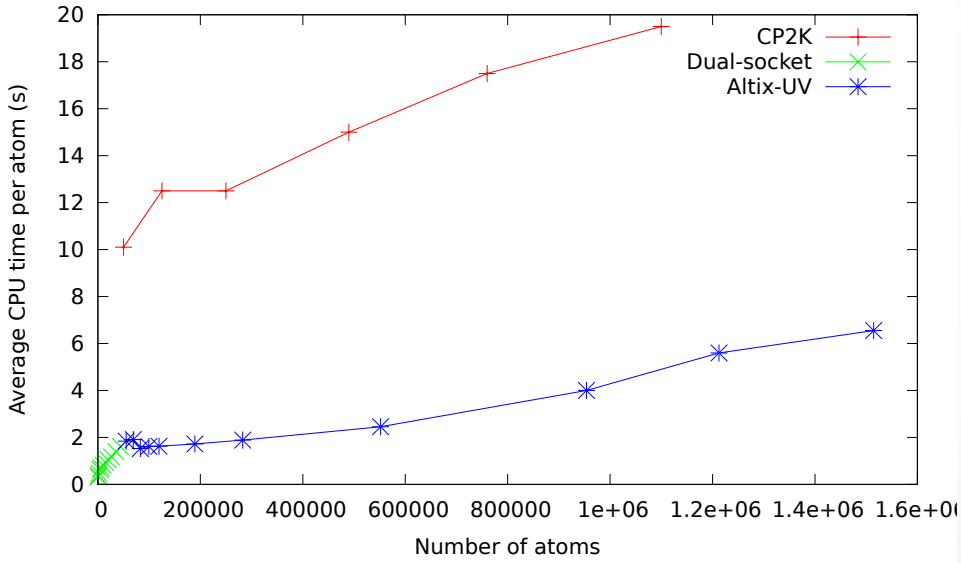
- Converges to the correct value
- Error of the method below the SCF convergence error

For 1 million water molecules, the total energy is ~ -4.7e6 a.u. To get the chemical accuracy with 1 million water molecules, we need a precision of 1e-9 a.u per molecule: an absolute error of 2.5e-10

Comparison with CP2K

Linear Scaling Self-Consistent Field Calculations with Millions of Atoms in the Condensed Phase

J. VandeVondele, U. Borštnik, J. Hutter, JCTC, **8** (10), 3565-3573 (2012)



Comparison is not quantitative: different architectures, different method, different number of SCF cycles

- deMon-Nano exploits very well the hardware, especially for medium-sized systems
- Wall time is better with CP2K for a single point with millions of atoms, because it can use >9000 cores
- MPI communications are indeed important in CP2K (2s / atom)
- Latency of Numalink of Altix-UV is 1.5-3x lower than MPI/Infiniband
- Hardware memory prefetchers make automatically asynchronous inter-blade communications

Conclusion

- We have accelerated deMon-Nano
- Parallel speedup is satisfactory
- Scaling is not linear, but our implementation is very efficient in the useful range (->100 000 atoms)
- More and more cores/node -> Buying a newer machine will make the code run faster
- Doesn't require petascale computers and expensive hardware for standard simulations
- Results equivalent to diagnonalization, error below SCF threshold.
- Approximations below chemical accuracy for one million atoms.

Project for the next years:

Distributed large scale computations (EGI grid?, PRACE?, etc) for Monte Carlo simulations of DNA in water.