QMC=Chem, a massively parallel Quantum Monte Carlo program

Anthony Scemama, Michel Caffarel

Laboratoire de Chimie et Physique Quantiques / IRSAMC,
Toulouse (France)
{scemama,caffarel}@irsamc.ups-tlse.fr
http://qmcchem.ups-tlse.fr

27 April 2011

Outline

QMC in short

Variational Monte Carlo

We want to calculate the energy E associated with a wave function Ψ:

$$E = \frac{\int \Psi(r_1, \ldots, r_N) \mathcal{H} \Psi(r_1, \ldots, r_N) \, \mathrm{d}r_1, \ldots, r_N}{\int \Psi^2(r_1, \ldots, r_N) \, \mathrm{d}r_1, \ldots, r_N} \tag{1}$$

If Ψ is normalized, one can re-write E in terms of a probability density Ψ^2:

$$E = \int \Psi^2(r_1, \ldots, r_N) \frac{\mathcal{H}\Psi(r_1, \ldots, r_N)}{\Psi(r_1, \ldots, r_N)} \, \mathrm{d}r_1, \ldots, r_N \tag{2}$$

E will be computed as the average of the *local energies* $\frac{\mathcal{H}\Psi}{\Psi}$ computed at points drawn randomly from the density Ψ^2.

QMC in short

Diffusion Monte Carlo

We want to compute the exact energy E_0 associated with the exact wave function Φ_0:

$$E_0 = \frac{\int \Phi_0(r_1, \ldots, r_N) \mathcal{H} \Phi_0(r_1, \ldots, r_N) \, \mathrm{d}r_1, \ldots, r_N}{\int \Phi_0^2(r_1, \ldots, r_N) \, \mathrm{d}r_1, \ldots, r_N} \tag{3}$$

Φ_0 is unknown, but a very good approximation ("fixed-node") of $\Psi\Phi_0$ can be sampled (Ψ is the trial wave function and $\tilde{\Phi}_0$ is the approximation of Φ_0). If $\Psi\tilde{\Phi}_0$ is normalized, one can re-write $\tilde{E}_0$ as

$$\tilde{E}_0 = \int \tilde{\Phi}_0(r_1, \ldots, r_N) \Psi(r_1, \ldots, r_N) \frac{\mathcal{H}\Psi(r_1, \ldots, r_N)}{\Psi(r_1, \ldots, r_N)} \tag{4}$$

E will be computed as the average of the local energies $\frac{\mathcal{H}\Psi}{\Psi}$ evaluated at points drawn from the density $\tilde{\Phi}_0\Psi$.

QMC in short

- Electronic trajectories to sample Ψ^2 or $\tilde{\Phi}_0 \Psi$
- At every step, the local energy is computed $\frac{\mathcal{H}\Psi}{\Psi}$
- The average of the local energies is the electronic energy of the system
- Many independent trajectories can be started in parallel with different initial conditions and random seeds.

QMC as an alternative method

Positive aspects

- The usual methods of comparable quality [CCSD(T)] have an $\mathcal{O}(N^7)$ scaling for CPU and $\mathcal{O}(N^4)$ scaling for disk space
- The cost of a Monte Carlo step is of order $\mathcal{O}(N^3)$, and can be $\mathcal{O}(N)$ for large systems
- The evaluation of one Monte Carlo step is fast (4 milliseconds for 60 electrons)

But

- The statistical error decreases as $1/\sqrt{M}$
- A huge number of Monte Carlo steps have to be realized
- So it is not a routine method (yet)

QMC as an alternative method

Positive aspects

- The usual methods of comparable quality [CCSD(T)] have an $\mathcal{O}(N^7)$ scaling for CPU and $\mathcal{O}(N^4)$ scaling for disk space
- The cost of a Monte Carlo step is of order $\mathcal{O}(N^3)$, and can be $\mathcal{O}(N)$ for large systems
- The evaluation of one Monte Carlo step is fast (4 milliseconds for 60 electrons)

But

- The statistical error decreases as $1/\sqrt{M}$
- A huge number of Monte Carlo steps have to be realized
- So it is not a routine method (yet)

QMC as an alternative method

These algorithms are perfect for massive parallelism

- Parallelism is easy to implement (SPMD)
- No necessary synchronization between the processes
- Little I/O, little communication, little RAM, a lot of CPU (number crunching)
- Adapted to super-computers, grids, GPUs, blue-gene, and even cloud computing
- Fault-tolerance easy to implement

Today, a very large number of Monte Carlo steps can be realized in a short time on expensive architectures. In a near future, these calculations will be accessible to laboratoiries.

QMC as an alternative method

These algorithms are perfect for massive parallelism

- Parallelism is easy to implement (SPMD)
- No necessary synchronization between the processes
- Little I/O, little communication, little RAM, a lot of CPU (number crunching)
- Adapted to super-computers, grids, GPUs, blue-gene, and even cloud computing
- Fault-tolerance easy to implement

Today, a very large number of Monte Carlo steps can be realized in a short time on expensive architectures. In a near future, these calculations will be accessible to laboratoiries.

Outline

Constraints

Provide a code with the following properties

- Fast (Fortran, linear scaling algorithms, etc)
- Easy to install (Standard Fortran and native Python)
- Easy to maintain (IRPF90 Fortran pre-processor)
- Easy to use (GUI, scripts, etc)
- Can run with no effort on any computing facility (security, libraries, filesystem,...)
- Can run on grids (unknown and heterogeneous hardware, slow network,...)
- Can run on as many cores as possible
- Fault tolerance
- A (almost) perfect parallel efficiency

Constraints

Provide a code with the following properties

- Fast (Fortran, linear scaling algorithms, etc)
- Easy to install (Standard Fortran and native Python)
- Easy to maintain (IRPF90 Fortran pre-processor)
- Easy to use (GUI, scripts, etc)
- Can run with no effort on any computing facility (security, libraries, filesystem,...)
- Can run on grids (unknown and heterogeneous hardware, slow network,...)
- Can run on as many cores as possible
- Fault tolerance
- A (almost) perfect parallel efficiency

Constraints

Provide a code with the following properties

- Fast (Fortran, linear scaling algorithms, etc)
- Easy to install (Standard Fortran and native Python)
- Easy to maintain (IRPF90 Fortran pre-processor)
- Easy to use (GUI, scripts, etc)
- Can run with no effort on any computing facility (security, libraries, filesystem,...)
- Can run on grids (unknown and heterogeneous hardware, slow network,...)
- Can run on as many cores as possible
- Fault tolerance
- A (almost) perfect parallel efficiency

Constraints

Provide a code with the following properties

- Fast (Fortran, linear scaling algorithms, etc)
- Easy to install (Standard Fortran and native Python)
- Easy to maintain (IRPF90 Fortran pre-processor)
- Easy to use (GUI, scripts, etc)
- Can run with no effort on any computing facility (security, libraries, filesystem,...)
- Can run on grids (unknown and heterogeneous hardware, slow network,...)
- Can run on as many cores as possible
- Fault tolerance
- A (almost) perfect parallel efficiency

Constraints

Provide a code with the following properties

- Fast (Fortran, linear scaling algorithms, etc)
- Easy to install (Standard Fortran and native Python)
- Easy to maintain (IRPF90 Fortran pre-processor)
- Easy to use (GUI, scripts, etc)
- Can run with no effort on any computing facility (security, libraries, filesystem,...)
- Can run on grids (unknown and heterogeneous hardware, slow network,...)
- Can run on as many cores as possible
- Fault tolerance
- A (almost) perfect parallel efficiency

Constraints

Provide a code with the following properties

- Fast (Fortran, linear scaling algorithms, etc)
- Easy to install (Standard Fortran and native Python)
- Easy to maintain (IRPF90 Fortran pre-processor)
- Easy to use (GUI, scripts, etc)
- Can run with no effort on any computing facility (security, libraries, filesystem,...)
- Can run on grids (unknown and heterogeneous hardware, slow network,...)
- Can run on as many cores as possible
- Fault tolerance
- A (almost) perfect parallel efficiency

Constraints

Provide a code with the following properties

- Fast (Fortran, linear scaling algorithms, etc)
- Easy to install (Standard Fortran and native Python)
- Easy to maintain (IRPF90 Fortran pre-processor)
- Easy to use (GUI, scripts, etc)
- Can run with no effort on any computing facility (security, libraries, filesystem,...)
- Can run on grids (unknown and heterogeneous hardware, slow network,...)
- Can run on as many cores as possible
- Fault tolerance
- A (almost) perfect parallel efficiency

Constraints

Provide a code with the following properties

- Fast (Fortran, linear scaling algorithms, etc)
- Easy to install (Standard Fortran and native Python)
- Easy to maintain (IRPF90 Fortran pre-processor)
- Easy to use (GUI, scripts, etc)
- Can run with no effort on any computing facility (security, libraries, filesystem,...)
- Can run on grids (unknown and heterogeneous hardware, slow network,...)
- Can run on as many cores as possible
- Fault tolerance
- A (almost) perfect parallel efficiency

Constraints

Provide a code with the following properties

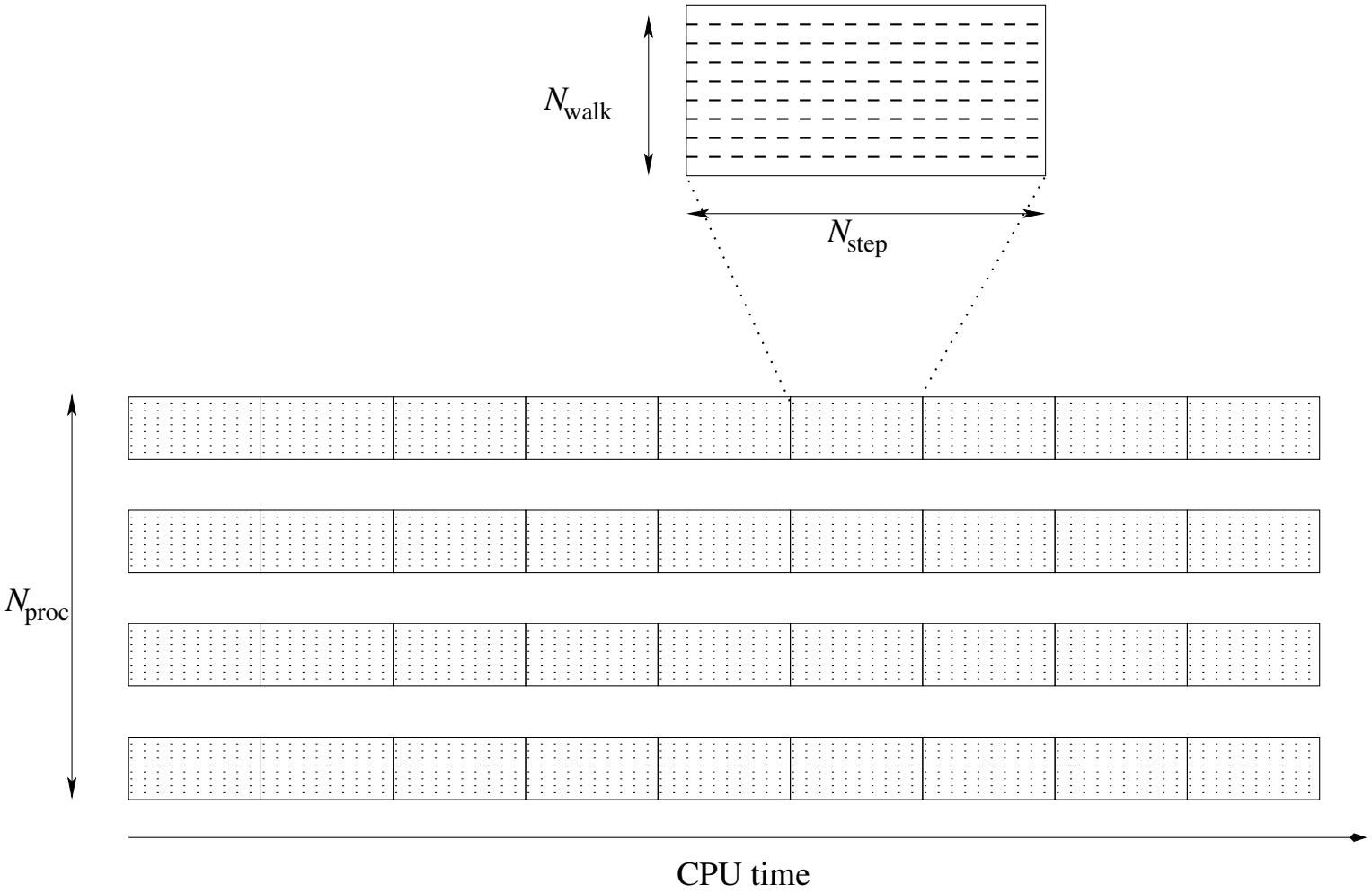
- Fast (Fortran, linear scaling algorithms, etc)
- Easy to install (Standard Fortran and native Python)
- Easy to maintain (IRPF90 Fortran pre-processor)
- Easy to use (GUI, scripts, etc)
- Can run with no effort on any computing facility (security, libraries, filesystem,...)
- Can run on grids (unknown and heterogeneous hardware, slow network,...)
- Can run on as many cores as possible
- Fault tolerance
- A (almost) perfect parallel efficiency

Design

- SPMD Fortran program
- Communication layer handled in Python
- No input, no output. Interaction of the user with a data base.
- GUI, scripts,... can easily read/write in the database

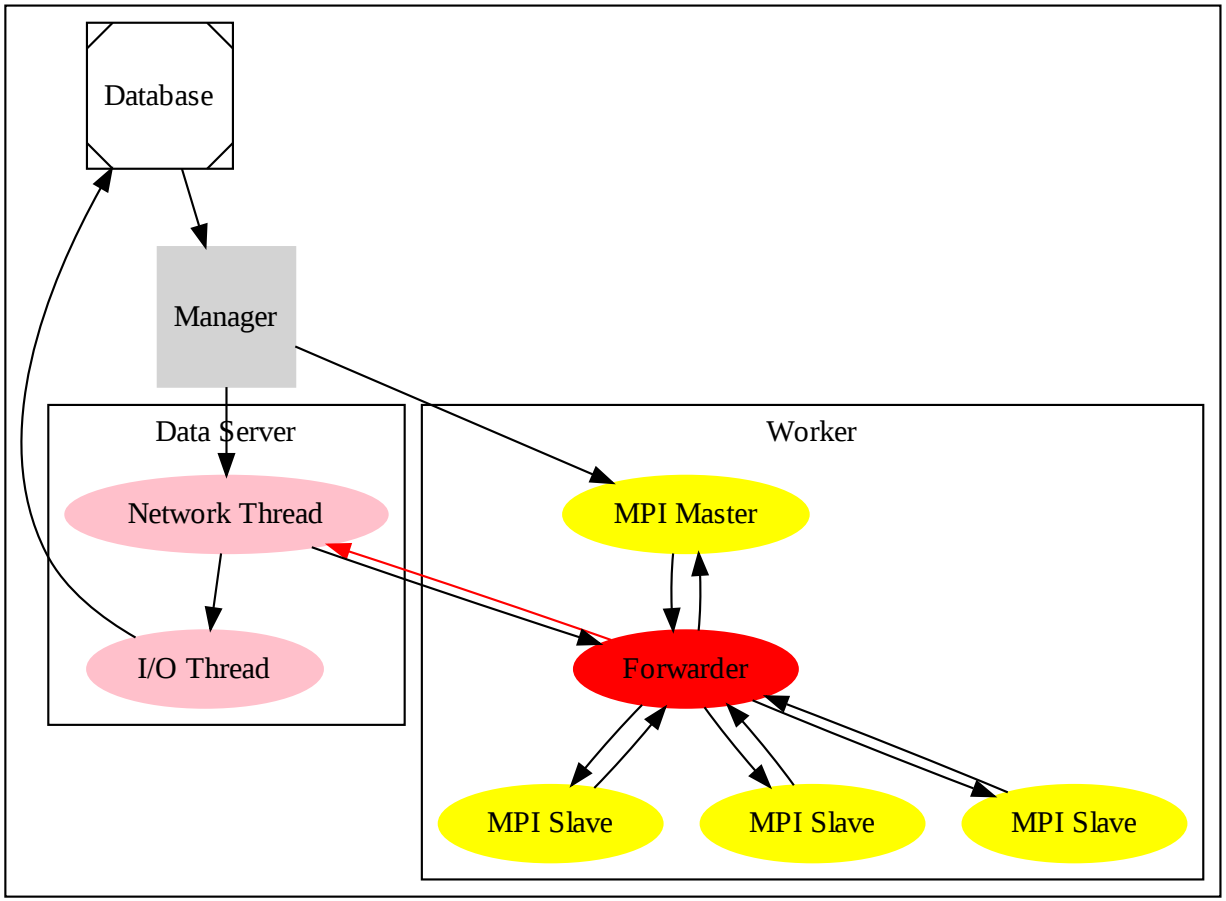
Design

Each trajectory is sliced in blocks:



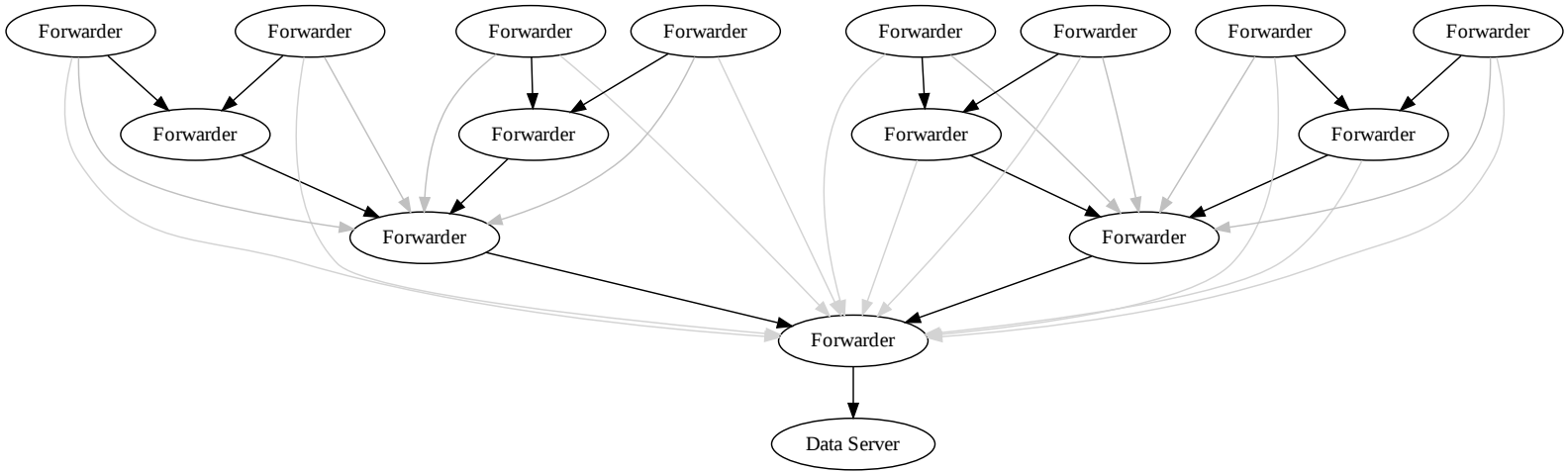
Design

Manager/Worker model with asynchronous I/O.



Design

Forwarders are organized in a binary tree with fault-tolerance.



Used architectures

Supercomputers

- CALMIP, Hyperion (2 816 cores) : 512 cores, routine runs
- CEA-TGCC, Curie (11 000 cores) : 10 000 cores, benchmarks

Grids

- EGI (140 000 cores) : 1 000 tasks runs in 2010
- Grid 5000 (5 000 cores) : Will be invaded soon....

Used architectures

Supercomputers

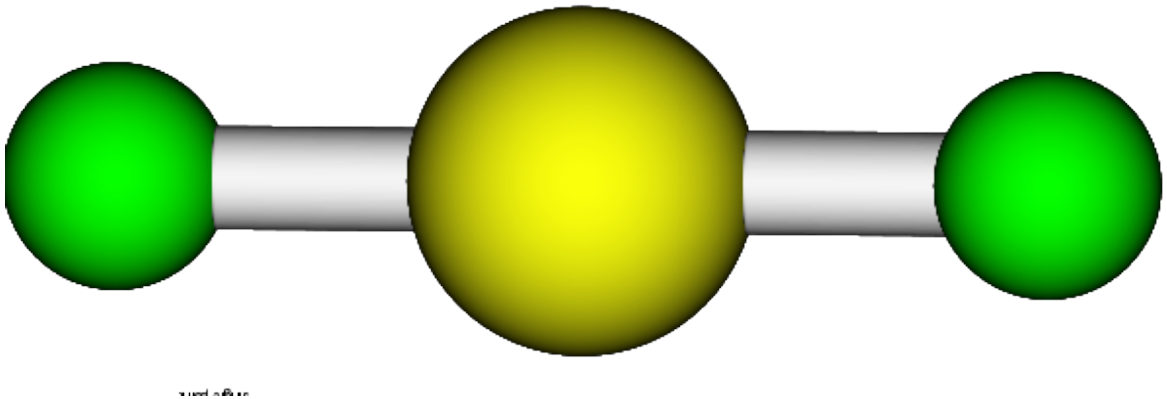
- CALMIP, Hyperion (2 816 cores) : 512 cores, routine runs
- CEA-TGCC, Curie (11 000 cores) : 10 000 cores, benchmarks

Grids

- EGI (140 000 cores) : 1 000 tasks runs in 2010
- Grid 5000 (5 000 cores) : Will be invaded soon....

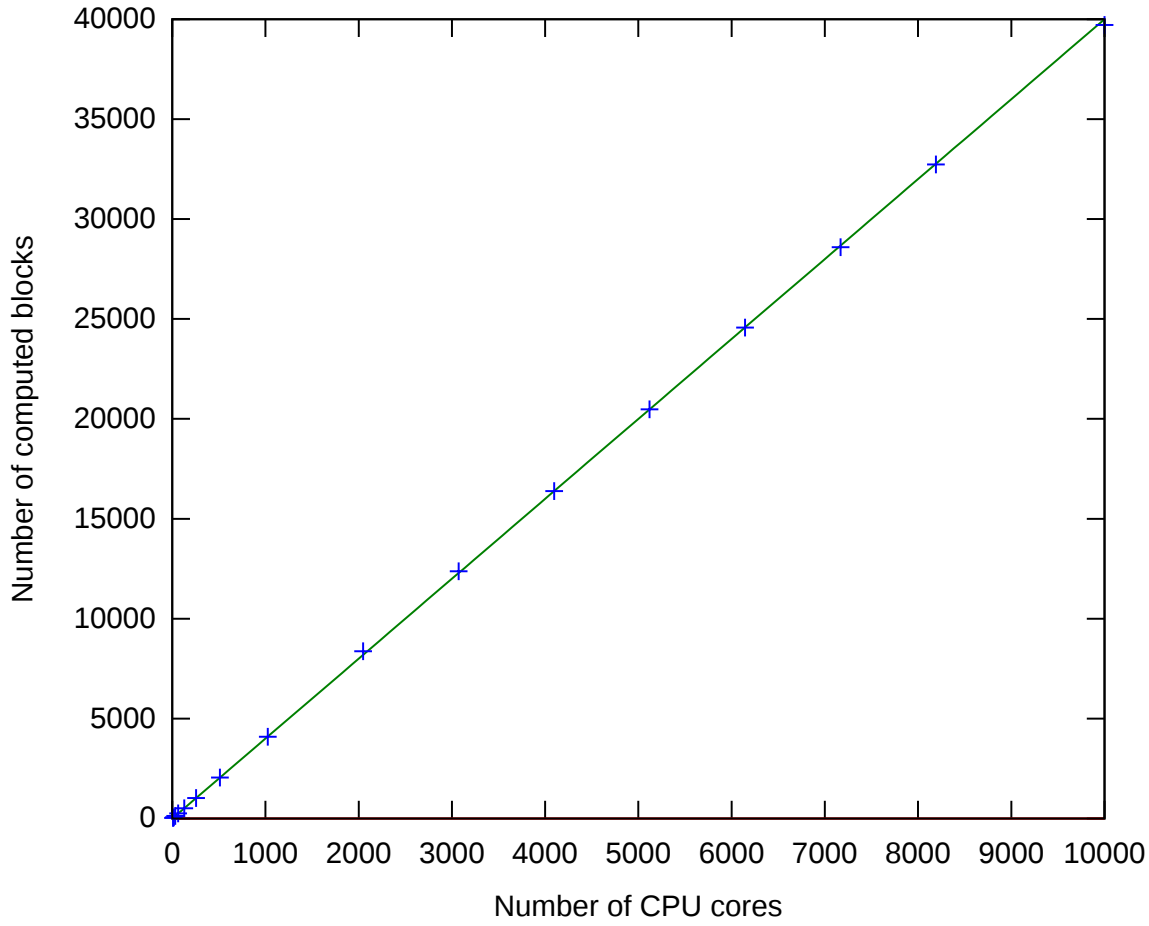
Outline

Description



- $CuCl_2$ molecule: 63 electrons, 336 Gaussian basis functions
- 1 block: 2 000 steps, 10 walkers (too small blocks, 84 seconds)
- 32 MB RAM, 1.5 MB disk (input), 1.2 KB / computed block
- Stopping condition: wall time > 5 minutes

Parallel part



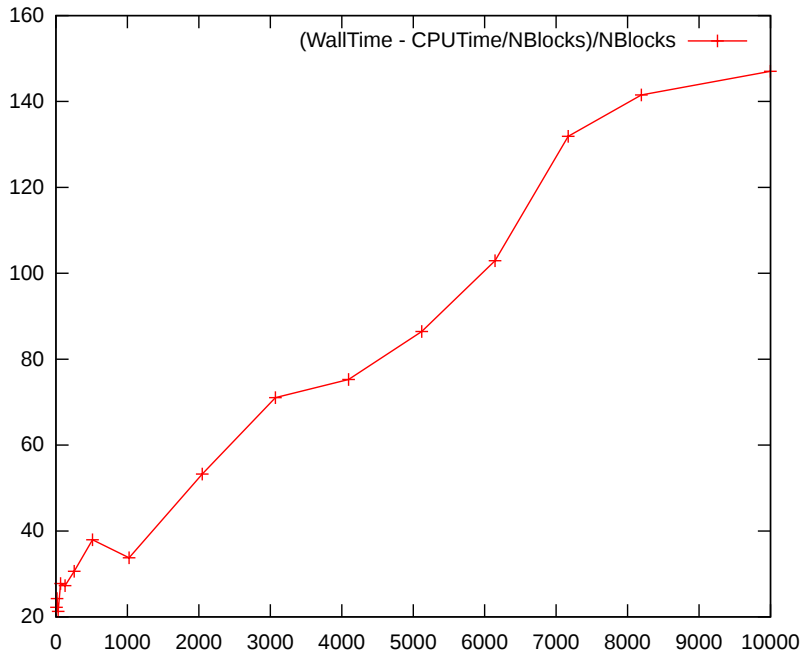
Curie, 1 block = 84 seconds, Stopping condition = wall time > 300 sec

Serial part

- Check consistency of input
- Compute CRC32 key associated with input (to check incoming blocks)
- MPI initialization
- MPI broadcast of input
- Pre-conditioning (matrix sparsifications, sorting, etc)
- *Send to the data server the last walker positions and random seeds
- MPI Finalization
- Wait for every forwarder to finish
- Wait for I/O thread to finish

* Longest part.

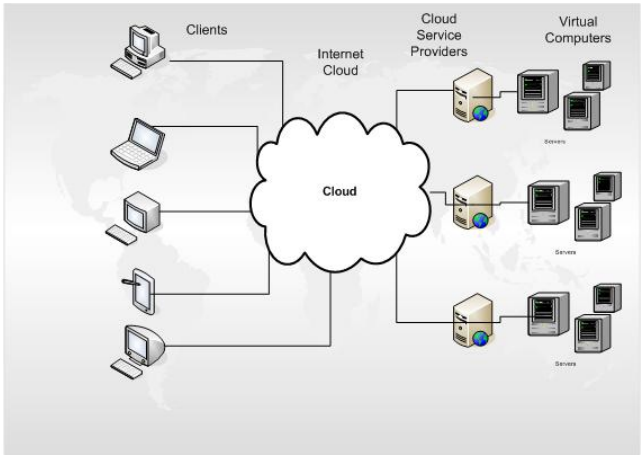
Serial part

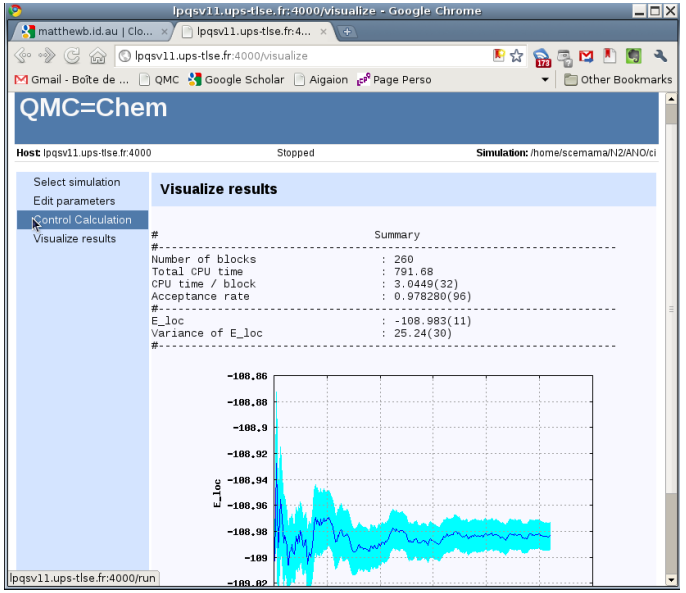


Does not depend on the block length. Not too bad.
For blocks of 100 walkers, 10 000 steps and "wall time > 8 hours", the parallel efficiency is 98.9%

Outline

Cloud computing





- SAAS (web)
- The users don’t worry about resources
- Well adapted to CPU-intensive programs
- ANR Project submitted / collaboration with a software company (ASA)

Future improvements

- Delegate some work to the forwarders to parallelize the finalization
- The forwarders will start the workers : possibility to plug new forwarders to a running simulation
- Deploy the program with TakTuk instead of MPI $\longrightarrow$ Full fault-tolerance implementation
- Better integration with grid environments