Quantum Monte Carlo with QMC=Chem : Basic concepts

Michel Caffarel and Anthony Scemama

CNRS and Université de Toulouse

Presentation of the basic notions required for the tutorial.

More information on our scientific website :

http :// qmcchem.ups-tlse.fr

Variational Monte Carlo (VMC)

In the tutorial method = "VMC"

- N electrons with positions : $\mathbf{r}_1, ..., \mathbf{r}_N$
- Given a trial approximate wavefunction $\Psi_T(\mathbf{r}_1, ..., \mathbf{r}_N)$

we want to compute the variational energy

$$E = \frac{\langle \Psi_T | H | \Psi_T \rangle}{\langle \Psi_T | \Psi_T \rangle}$$

or any property

$$\langle A \rangle = \frac{\langle \Psi_T | A | \Psi_T \rangle}{\langle \Psi_T | \Psi_T \rangle}$$

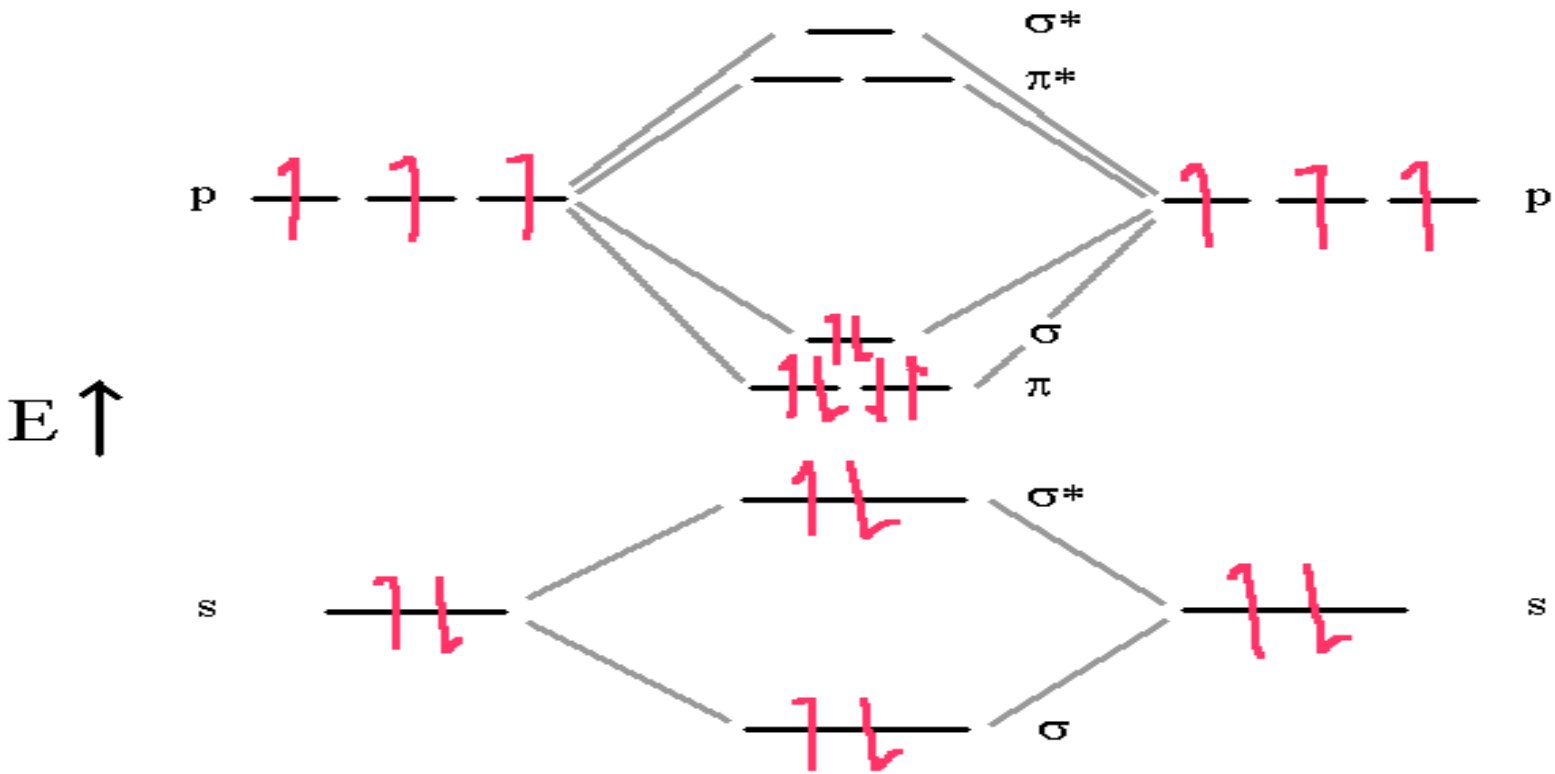
Variational Monte Carlo (VMC)

In the tutorial the trial wavefunction used is either a Hartree-Fock or a CAS-SCF wavefunction

The molecule treated will be N_2

14 electrons and 7 double-occupied molecular orbitals : $\phi_1, ..., \phi_7$ in the $S = 0$ ground-state

N2 molecule



$\phi_1 = 1\sigma_{1s}$ $\phi_2 = 1\sigma^*_{1s}$(core orbitals not shown on the diagram)

$\phi_3 = 2\sigma_{2s}$ $\phi_4 = 2\sigma^*_{2s}$ $\phi_5 = 1\pi_{2px}$ $\phi_6 = 1\pi_{2py}$ $\phi_7 = 3\sigma_{2pz}$

Variational Monte Carlo (VMC)

In quantum chemistry Ψ_{SCF} is usually expressed using space $\mathbf{r}$ and spin variables, $\sigma = \alpha$ (or $\uparrow$) and $\sigma = \beta$ (or $\downarrow$)

$$\Psi_{SCF}(\mathbf{r}_1, \sigma_1, ..., \mathbf{r}_N, \sigma_N) = \begin{vmatrix} \phi_1\alpha & & & \\ & \phi_1\beta & & \\ \vdots & & \vdots & \vdots \\ & & \phi_7\alpha & \\ & & & \phi_7\beta \end{vmatrix}$$

Variational Monte Carlo (VMC)

In QMC methods we use an equivalent spin-free formalism (averages are the same) = the wavefunction Ψ_T depends only on the spatial positions of electrons.

Here, electrons 1 to 7 have been chosen as α electrons and electrons 8 to 14 as β electrons (arbitrary)

$$\Psi_T^{SCF}(\mathbf{r}_1, ..., \mathbf{r}_N) = \begin{vmatrix} \phi_1(\mathbf{r}_1) & \dots & \phi_1(\mathbf{r}_7) \\ \vdots & \vdots & \vdots \\ \phi_7(\mathbf{r}_1) & \dots & \phi_7(\mathbf{r}_7) \end{vmatrix} \begin{vmatrix} \phi_1(\mathbf{r}_8) & \dots & \phi_1(\mathbf{r}_{14}) \\ \vdots & \vdots & \vdots \\ \phi_7(\mathbf{r}_8) & \dots & \phi_7(\mathbf{r}_{14}) \end{vmatrix}$$

Variational Monte Carlo (VMC)

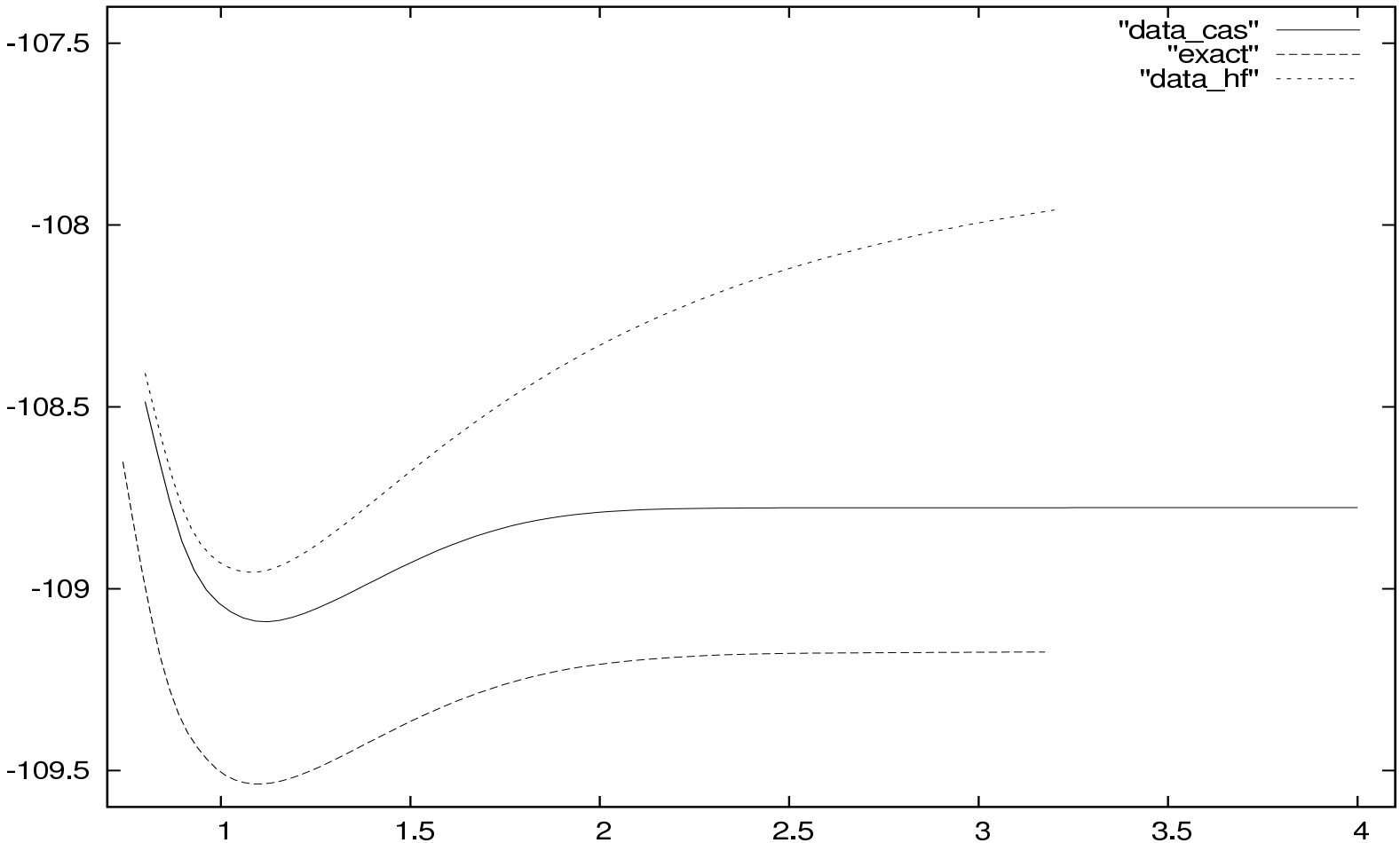
CAS-SCF wavefunction based on the determinantal expansion over all possible excitations within a set of N_{act} active orbitals chosen among the M molecular orbitals (M = total size of the basis set)

$$\Psi_T^{CAS-SCF} = \sum_{all\ \ excitations\ \ K=(k_1^\alpha,\ldots,k_1^\beta,\ldots)} c_K$$

$$\begin{vmatrix} \phi_{k_1^\alpha}(\mathbf{r}_1) & \ldots & \phi_{k_1^\alpha}(\mathbf{r}_7) \\ \vdots & \vdots & \vdots \\ \phi_{k_7^\alpha}(\mathbf{r}_1) & \ldots & \phi_{k_7^\alpha}(\mathbf{r}_7)(\mathbf{r}_7) \end{vmatrix} \begin{vmatrix} \phi_{k_1^\beta}(\mathbf{r}_8) & \ldots & \phi_{k_1^\beta}(\mathbf{r}_{14}) \\ \vdots & \vdots & \vdots \\ \phi_{k_7^\beta}(\mathbf{r}_8) & \ldots & \phi_{k_7^\beta}(\mathbf{r}_{14}) \end{vmatrix}$$

Variational Monte Carlo (VMC)

In this tutorial HF and CASSCF wave functions from GAMESS.



Variational Monte Carlo (VMC)

How to compute the variational energy ?

$$E = \frac{\langle \Psi_T | H | \Psi_T \rangle}{\langle \Psi_T | \Psi_T \rangle}$$

Standard *ab initio* wft approaches

- Ψ_T = expansion over a sum of determinants
- Each determinant is an antisymmetrized product of molecular orbitals ϕ_i
- Each molecular orbital is expressed as a sum over primitive gaussian functions : $\phi_i = \sum_j c_{ij} \chi_j$

$\Rightarrow E$ is obtained as a huge sum of elementary one-electron and bielectronic integrals

Variational Monte Carlo (VMC)

VMC= computing the very same variational energy using Monte Carlo techniques

Big advantage = no need of calculating the numerous one-electron and bielectronic integrals.

Variational Monte Carlo (VMC)

VMC= generate in a probabilistic way billions of "electronic configurations" $\mathbf{R} = (\mathbf{r}_1, ..., \mathbf{r}_N)$ distributed in $3N$-dimensional space according to the density probability

$$\Pi(\mathbf{R}) = \frac{\Psi_T^2(\mathbf{r}_1, ..., \mathbf{r}_N)}{\int d\mathbf{r}_1, ..., d\mathbf{r}_N \Psi_T^2(\mathbf{r}_1, ..., \mathbf{r}_N)}$$

The average energy can be expressed as

$$E = \frac{\langle \Psi_T | H | \Psi_T \rangle}{\langle \Psi_T | \Psi_T \rangle} = \frac{\int d\mathbf{r}_1 ... d\mathbf{r}_N \Psi_T^2(\mathbf{r}_1, ..., \mathbf{r}_N) \frac{H\Psi_T}{\Psi_T}}{\int d\mathbf{r}_1 ... d\mathbf{r}_N \Psi_T^2(\mathbf{r}_1, ..., \mathbf{r}_N)}$$

$$E = \int d\mathbf{R} \Pi(R) \frac{H\Psi_T}{\Psi_T}$$

Variational Monte Carlo (VMC)

Here, let us define the local energy E_L as

$$E_L(\mathbf{r}_1, ..., \mathbf{r}_N) \equiv \frac{H\Psi_T}{\Psi_T}$$

we have :

$$E = \lim_{K \to +\infty} = \frac{1}{K} \sum_{k=1}^{K} E_L(\mathbf{R}^{(k)})$$

K = number of electronic configurations generated by Monte Carlo

Metropolis algorithm

How to generate electronic configurations according to Π?

Answer : Thanks to a Metropolis algorithm (many variants of it)

In a Metropolis algorithm there are two steps :

1) Move the electrons with a certain probability

2) Accept or reject this move according to some probability

Metropolis algorithm

A first type of Metropolis algorithm : sampling="Brownian"

1) Electronic configurations are moved according to :

$$p(\mathbf{R} \rightarrow \mathbf{R}') = \frac{1}{(2\pi\tau)^{3N/2}} \exp[-\frac{(\mathbf{R}' - \mathbf{R} - \mathbf{b}(\mathbf{R})\tau)^2}{2\tau}]$$

$\mathbf{b}$ = drift vector= $\frac{\nabla\psi_T}{\psi_T}$ and τ = time-step

2) Configurations are accepted/rejected with probability q

$$q = Min[1, \frac{\Pi(\mathbf{R}')\mathbf{p}(\mathbf{R}' \rightarrow \mathbf{R})}{\Pi(\mathbf{R})\mathbf{p}(\mathbf{R} \rightarrow \mathbf{R}')}]$$

A random number χ is drawn $\in$ (0,1). If q > χ move accepted, if not rejected.

Metropolis algorithm

In practice, drawing new configurations $\mathbf{R}'$ according to $p(\mathbf{R} \rightarrow \mathbf{R}')$ is very simple. The probability is a product of $3N$ independent 1D probability density

$$p(\mathbf{R} \rightarrow \mathbf{R}') = \prod_{i=1}^{N} \frac{1}{\sqrt{2\pi\tau}} \exp\left[-\frac{(x_i' - x_i - b^x{}_i(\mathbf{R})\tau)^2)}{2\tau}\right]$$

$$\frac{1}{\sqrt{2\pi\tau}} \exp\left[-\frac{(y_i' - y_i - b^y{}_i(\mathbf{R})\tau)^2}{2\tau}\right]$$

$$\frac{1}{\sqrt{2\pi\tau}} \exp\left[-\frac{(z_i' - z_i - b^z{}_i(\mathbf{R})\tau)^2}{2\tau}\right]$$

Metropolis algorithm

Simply draw $3N$ independent gaussian numbers η with zero mean and variance 1

$$\frac{x_i' - x_i - b_i^x(\mathbf{R})\tau}{\sqrt{\tau}} = \eta_i^x$$

$$\frac{y_i' - y_i - b_i^y(\mathbf{R})\tau}{\sqrt{\tau}} = \eta_i^y$$

$$\frac{z_i' - z_i - b_i^z(\mathbf{R})\tau}{\sqrt{\tau}} = \eta_i^z$$

or

$$x_i' = x_i + b_i^x(\mathbf{R})\tau + \sqrt{\tau}\eta_i^x$$

$$y_i' = y_i + b_i^y(\mathbf{R})\tau + \sqrt{\tau}\eta_i^y$$

$$z_i' = z_i + b_i^z(\mathbf{R})\tau + \sqrt{\tau}\eta_i^z$$

Metropolis algorithm

Second type : sampling="Langevin"

Use of the true Langevin equation :

$$dR(t) = P(t)/mdt$$

$$dP(t) = -gradV(R(t))dt - \gamma P(t)dt + \sigma dW(t)$$

Instead of

$$dR(t) = b(R(t))dt + \sigma dW(t)$$

See, A. Scemama, T. Lelièvre, G. Stoltz, E. Cancès, and M.C J. Chem. Phys. vol.125, 114105 (2006).

Summary about VMC

To make a VMC calculation we need to compute at each Monte Carlo step :

- Values of Ψ_T, $\mathbf{b} = \frac{\nabla\psi_T}{\psi_T}$, and $E_L = H\psi_T/\psi_T = -1/2\nabla^2\Psi_T/\Psi_T$
- Draw gaussian and random numbers

Very easy to do, no one- or two-electron integrals to compute.

Important consequence : any form for the trial wavefunction can be used

In the tutorial : HF and CASSCF wavefunctions so we can compare the standard approach and the Monte Carlo one.

In practice, more sophisticated wavefunctions whose averages cannot be calculated using the standard approach are used.

Slater-Jastrow trial wavefunction

Most popular form

$$\Psi_T = e^{J(\mathbf{r}_1,\ldots,\mathbf{r}_N)} \sum_{k=1}^{N_{det}} c_K Det_K(\{\phi^{\alpha}_{k_i}\}) Det_k(\{\phi^{\beta}_{k_i}\})$$

with (typically)

$$J(\mathbf{r}_1,\ldots,\mathbf{r}_N) = \sum_{i<j} v_{e-e}(r_{ij}) + \sum_{\alpha} v_{e-n}(r_{i\alpha}) + \sum_{i<j}\sum_{\alpha} v_{e-e-n}(r_{ij}, r_{i\alpha}, r_{j\alpha})$$

Electron-electron CUSP conditions : when electrons i and j are very close $v_{e-e}(r_{ij}) \rightarrow 1/2\ r_{ij}$(spin-unlike) or $1/4\ r_{ij}$ (spin-like)

Other trial wavefunctions

- Jastrow-Geminal wavefunction (Casula, Sorella and coll.)
- Wavefunctions with backflow (Drummond, Rios, Needs, Mitas, and coll.)
- Pfaffian wavefunction (Mitas and coll.)
- MultiJastrow wavefunction (Bouabça, MC and coll.)
- VB-type wavefunctions (Braida, MC, Goddard and coll.)
- etc.

Fixed-Node Diffusion Monte Carlo (DMC)

DMC = VMC (move + acceptance/rejection step) + Birth-death (branching) process

Birth-death process : After VMC step the electronic configuration is duplicated into M copies ($M = 0, 1, 2, ..$)

$$M = \text{Integer\_part}(\exp\left[-\tau(E_L - E_{ref})\right] + u)$$

where u is a uniform random number in (0,1) (so that the average number of copies is equal to $\exp\left[-\tau(E_L - E_{ref})\right]$)

Population of walkers in DMC

Because of the possible variation of the number of electronic configurations, to perform a DMC calculations we need to introduce a population of electronic configurations (or walkers). In QMC=Chem this number is denoted walk_num.

It can be shown that by adjusting $E_{ref} = E_0$ the size of population of walkers can be kept constant in average.

Note that in VMC it is not necessary to introduce a population of walkers but in general we also consider walk_num walkers.

Fixed-Node Diffusion Monte Carlo (DMC)

Using DMC rules the density obtained is

$$\Pi(\mathbf{R}) = \frac{\Psi_T(\mathbf{r}_1, ..., \mathbf{r}_N)\Phi_0^{FN}(\mathbf{r}_1, ..., \mathbf{r}_N)}{\int d\mathbf{r}_1, ..., d\mathbf{r}_N \Psi_T(\mathbf{r}_1, ..., \mathbf{r}_N)\Phi_0(\mathbf{r}_1, ..., \mathbf{r}_N)}$$

where $\Phi_0^{FN}(\mathbf{r}_1, ..., \mathbf{r}_N)$ = solution of the Schrödinger equation :

$$H\Phi_0^{FN}(\mathbf{r}_1, ..., \mathbf{r}_N) = E_0^{FN}\Phi_0^{FN}(\mathbf{r}_1, ..., \mathbf{r}_N)$$

with the Fixed-Node (FN) constraint : $\Phi_0^{FN}(\mathbf{r}_1, ..., \mathbf{r}_N) = 0$
whenever $\Psi_T(\mathbf{r}_1, ..., \mathbf{r}_N) = 0$

Fixed-Node Diffusion Monte Carlo (DMC)

The estimate of the exact energy is obtained as in VMC by averaging the local energy

$$<< E_L >>= \frac{\int d\mathbf{r}_1 ... d\mathbf{r}_N \Psi_T \phi_0^{FN} \frac{H\Psi_T}{\Psi_T}}{\int d\mathbf{r}_1 ... d\mathbf{r}_N \Psi_T \phi_0^{FN}} = \frac{\int d\mathbf{r}_1 ... d\mathbf{r}_N \phi_0^{FN} H\Psi_T}{\int d\mathbf{r}_1 ... d\mathbf{r}_N \Psi_T \phi_0^{FN}} = E_0^{FN}$$

- In general the fixed-node error is small (a few percents of the correlation energy)
- Variational property : $E_0^{FN} \geq E_0$

Fixed-Node Diffusion Monte Carlo (DMC)

The expectation value of an observable A is biased in DMC

$$< A >= \frac{\int d\mathbf{r}_1 ... d\mathbf{r}_N \Psi_T \phi_0 A}{\int d\mathbf{r}_1 ... d\mathbf{r}_N \Psi_T \phi_0} \neq \frac{\int d\mathbf{r}_1 ... d\mathbf{r}_N \phi_0^2 A}{\int d\mathbf{r}_1 ... d\mathbf{r}_N \phi_0^2}$$

A simple way of improving the average

$$< A > \sim 2 < A >_{DMC} - < A >_{VMC}$$

Indeed : $\langle \phi_0 | A | \phi_0 \rangle = \langle \Psi_T + \delta\phi | A | \Psi_T + \delta\phi \rangle$

where $\delta\phi = \phi_0 - \Psi_T$

$\langle \phi_0 | A | \phi_0 \rangle = \langle \Psi_T | A | \Psi_T \rangle + 2\langle \delta\phi | A | \Psi_T \rangle + O(\delta\phi^2)$

$\langle \phi_0 | A | \phi_0 \rangle = 2\langle \phi_0 | A | \Psi_T \rangle - \langle \Psi_T | A | \Psi_T \rangle + O(\delta\phi^2)$

QMC is fully parallelizable

Input:

$\{\mathbf{R}_{\rm nucl}\}, \mathbf{N}_\alpha, \mathbf{N}_\beta$

$\Psi_T(\vec{r}_1, \ldots, \vec{r}_N)$

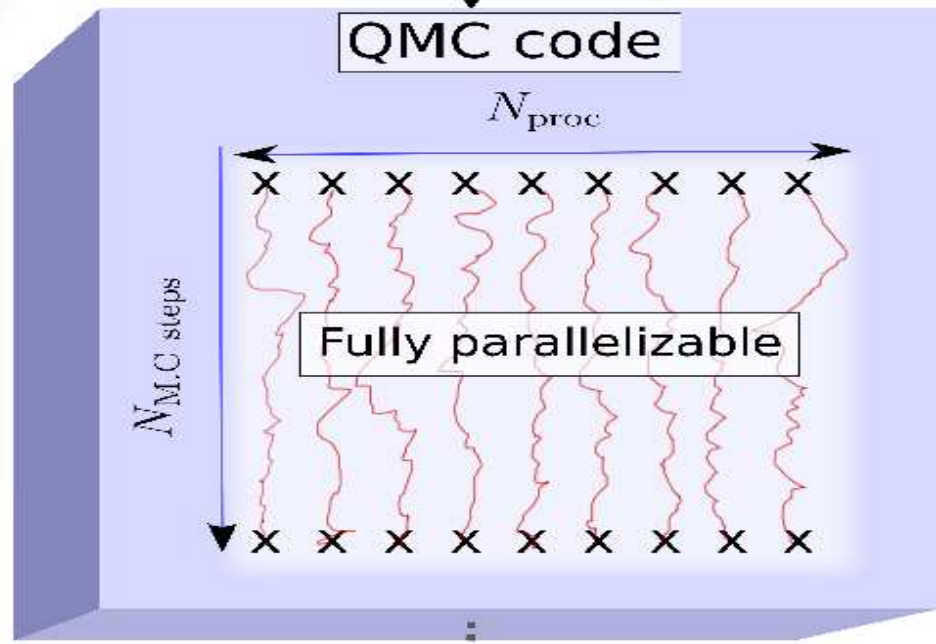
Stochastic dynamics of *electrons*:

$$\begin{cases} \Psi_T(\vec{r}_1, \ldots, \vec{r}_N) \\ \vec{\nabla}\Psi_T(\vec{r}_1, \ldots, \vec{r}_N) \\ \vec{\nabla}^2\Psi_T(\vec{r}_1, \ldots, \vec{r}_N) \end{cases}$$

Expectation values:

$E_0(\mathbf{R}_{\rm nucl}), \quad \Delta E_0(\mathbf{R}_{\rm nucl}),$

$\vec{\nabla} E(\mathbf{R}_{\rm nucl}), \quad \rho(\vec{r}), \quad \ldots$



Do not forget to show the video !

Benchmarks

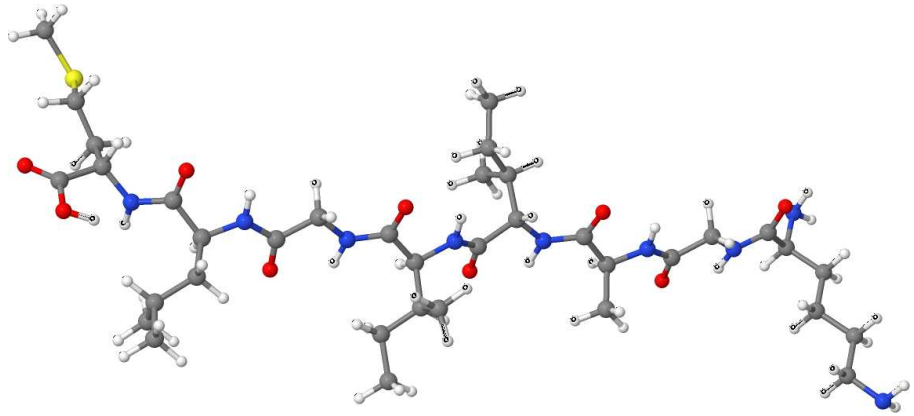
- S. Manten and A. Luchow, J.Chem.Phys. **115**, 5362 (2001).
- J.C. Grossman J.Chem.Phys. **117**, 1434 (2002).
- Nemec et al., J.Chem.Phys. **132**, 034111 (2010).

G1 set Pople et collab. (1990) = 55 molecules. Atomization energies.

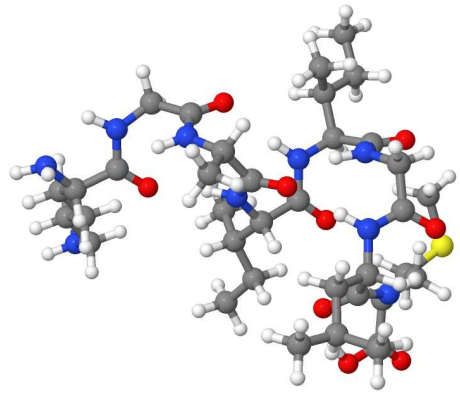
Mean Absolute Deviation ϵ_{MAD}

- FN-DMC : $\epsilon_{MAD} = 2.9 kcal/mol$
- LDA : $\epsilon_{MAD} \sim 40 kcal/mol$
- (B3LYP et B3PW91) $\epsilon_{MAD} \sim 2.5 kcal/mol$
- CCSD(T)/aug-cc-pVQZ $\epsilon_{MAD} \sim 2.8 kcal/mol$

QMC simulations for Amyloid β



Aβ(28-35) β-Strand structure simulated with QMC



Aβ(28-35) α-Helix structure simulated with QMC

Petascale QMC simulations

- Simulations done on CURIE supercomputer at CEA, France, december 2011. Key people : Anthony Scemama (LCPQ) and Bull engineers

- QMC simulations involving 122 atoms and 432 electrons (largest chemical system to date).

- **Use of about 80 000 cores** during about 24h, $\sim$ **960 Tflops/s** **(real performance)**

- 32.5% **of the peak performance** of the INTEL SANDY BRIDGE processor thanks to optimization tools (Exascale Computing Research Lab.)

Practical aspects : Convergence of energy

As we have seen in VMC and DMC the energy is estimated as the average of the local energy over K electronic configurations

$$\overline{E} \sim \frac{1}{K} \sum_{k=1}^{K} E_L(\mathbf{R}^{(k)})$$

How to get an estimation of the statistical error for E ?.

Practical aspects : Convergence of energy

1) Generation of a certain number of "blocks" N_b defined as a set of walk_num walkers having made num_step moves

2) Each block k contains about $N_k = walk\_num \times num\_step$ values of the local energy leading to

$$\overline{E}_k = \frac{1}{N_k} \sum_{i=1}^{N_k} E_L(\mathbf{R}^{(i)})$$

3) If num_step is sufficiently large, such blocks must be statistically independent and gaussian distributed (central-limit theorem for Markovian processes)

Practical aspects : Convergence of energy

4) The energy is then estimated as the average over the N_b blocks

$$\overline{E} = \frac{1}{N_b} \sum_{k=1}^{N_b} \overline{E}_k$$

and the error by

$$\delta E = \frac{\sigma}{\sqrt{N_b}}$$

where σ is an estimate of the standard deviation of the gaussian distribution

$$\sigma = \sqrt{\frac{1}{N_b - 1} \sum_{k=1}^{N_b} (\overline{E}_k - \overline{E})^2}$$

Practical aspects : Convergence of energy

The final result is written under the form

$$\overline{E} \pm \delta E$$

Exact result within one standard deviation ($\overline{E} \pm \delta E$) : probability = 68.2%

Exact result within two standard deviations ($\overline{E} \pm 2\delta E$) : probability = 95.4%

Practical aspects : Convergence of energy

It is important to check

1) The transient regime is attained. For that the evolution of $\overline{E}$ as a function of the number of blocks must be looked at.

2) The distribution of blocks is indeed gaussian.

Other practical aspects

1) Choice of the value of the time-step

2) Treatment of electron-nucleus CUSP

Other practical aspects

1) Choice of the value of the time-step

2) Treatment of electron-nucleus CUSP

Implementation of parallelism in QMC=Chem

Anthony Scemama <scemama@irsamc.ups-tlse.fr>
Michel Caffarel <michel.caffarel@irsamc.ups-tlse.fr>

Labratoire de Chimie et Physique Quantiques
IRSAMC (Toulouse)

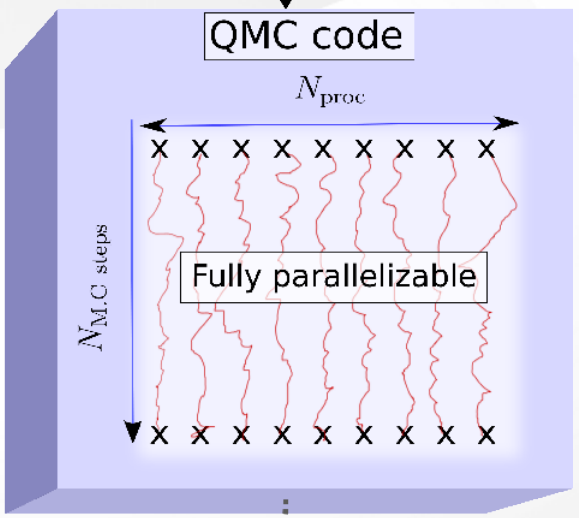
Parallelization of VMC

In VMC, all the trajectories are completely independent:

Input:

$\{\mathbf{R}_{\text{nucl}}\}, \mathbf{N}_\alpha, \mathbf{N}_\beta$

$\Psi_T(\vec{r}_1, \ldots, \vec{r}_N)$



Stochastic dynamics of *electrons*:

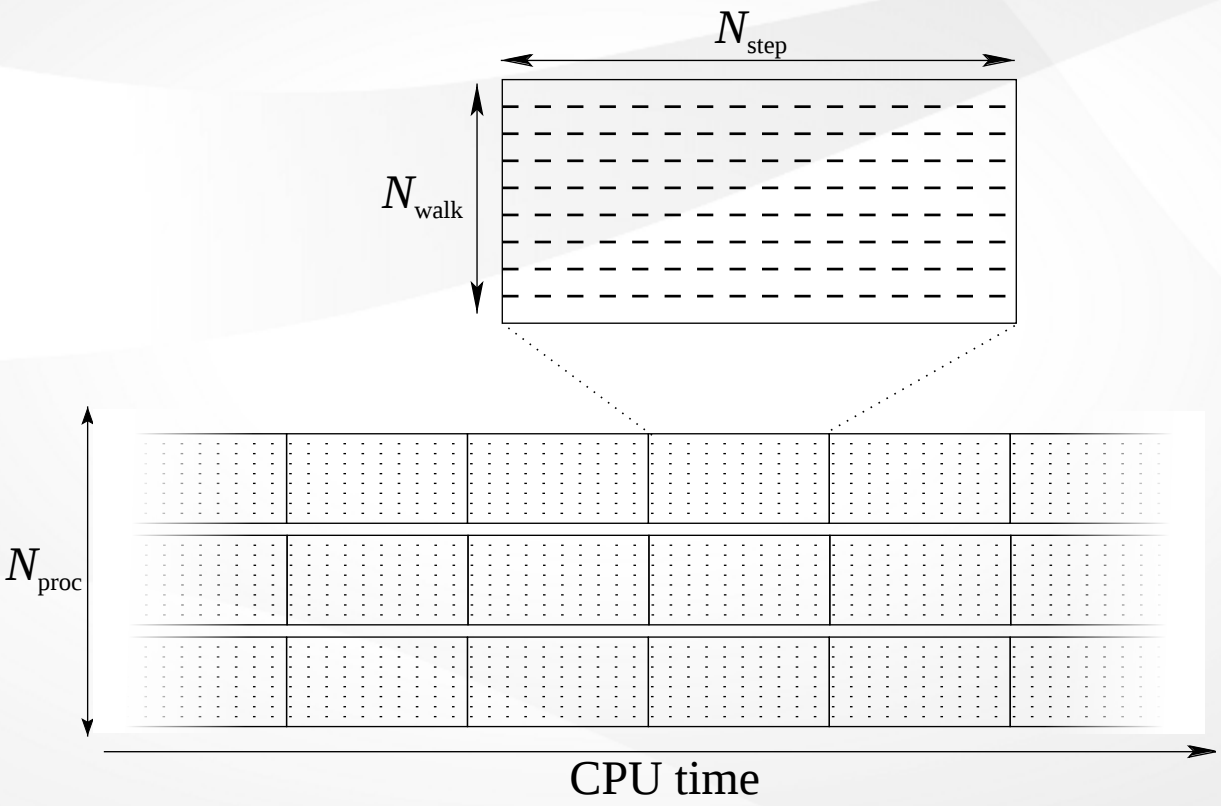
$$\begin{cases} \Psi_T(\vec{r}_1, \ldots, \vec{r}_N) \\ \vec{\nabla}\Psi_T(\vec{r}_1, \ldots, \vec{r}_N) \\ \vec{\nabla}^2\Psi_T(\vec{r}_1, \ldots, \vec{r}_N) \end{cases}$$

Expectation values:

$E_0(\mathbf{R}_{\text{nucl}}), \quad \Delta E_0(\mathbf{R}_{\text{nucl}}),$

$\vec{\nabla} E(\mathbf{R}_{\text{nucl}}), \quad \rho(\vec{r}), \ \ldots$

- Pack together a pool of N_{walk} walkers
- Cut the trajectories in smaller pieces of equal size (N_{step})
- Each CPU computes a block: N_{walk} executing N_{step}



Naive implementation:

- Parallelize with MPI
- At the end of each block, call *MPI_all_reduce* to update the running averages
- If too much memory is used, eventually add an OpenMP layer

This approach is not optimal [*]:

- At every synchronization, all processes will wait until the slowest has finished. Perfect parallel speed-up is impossible to obtain.
- The calculation can't start until all resources are available
- If one compute node crashes, all the simulation is crashed.
- If more resources become available, it is impossible to attach more CPUs to a running calculation

---

[*] "**Manager–worker-based model for the parallelization of quantum Monte Carlo on heterogeneous and homogeneous networks**", M. T. Feldmann, J. C. Cummings, D. R. Kent, R. P. Muller, W. A. Goddard III, J. *Comput. Chem.* 29, 8–16 (2008).

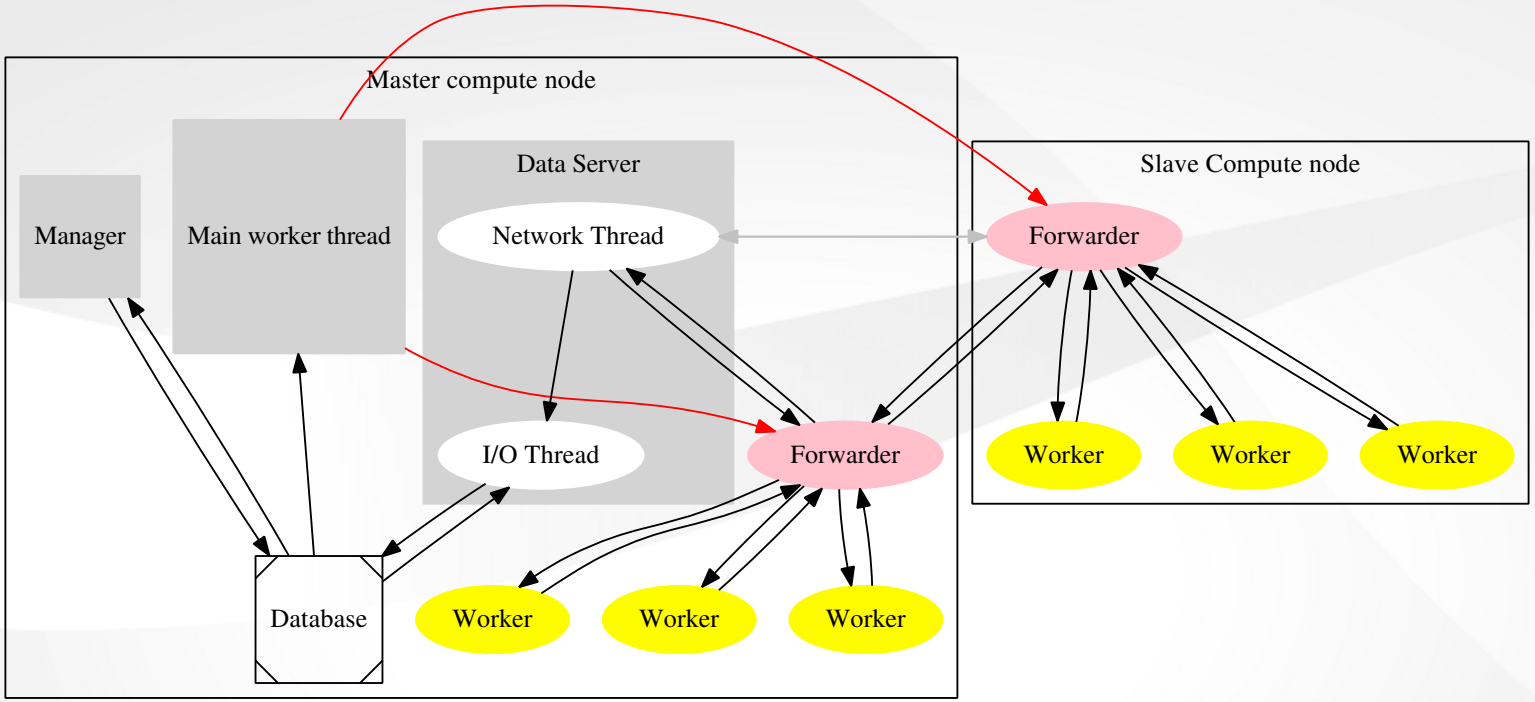
Our approach [†] :

- Manager/worker model: all CPUs are desynchronized, they start immediately
- The length of the block is not fixed: termination is immediate
- Use as less memory/core as possible (<100 MiB / core)
- Implement a client/server model (in Python):
  - allows client nodes to crash
  - allows to dynamically add/remove clients
- Avoid the traditional input/output file model. All data is stored in a database, and data is post-processed.
- Possibility to use computing grids (EGI [‡])

---

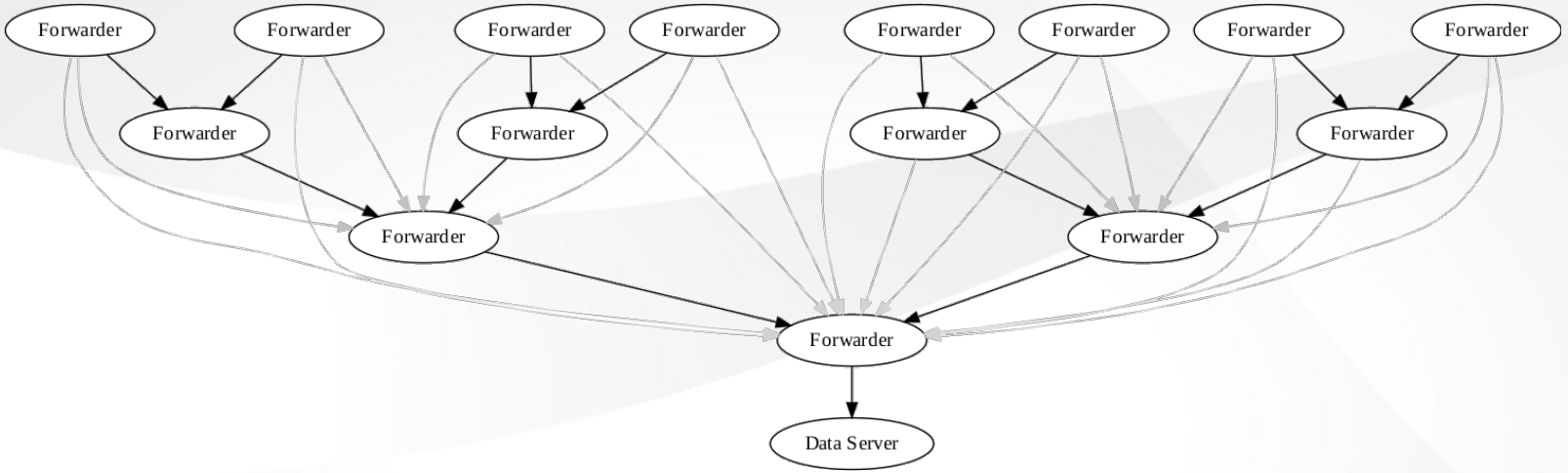
† "**Quantum monte carlo for large chemical systems: Implementing efficient strategies for petascale platforms and beyond**", A. Scemama, M. Caffarel, E. Oseret and W. Jalby, *J. Comput. Chem* 34, 938–951 (2013).

‡ "**Large-scale quantum Monte Carlo electronic structure calculations on the EGEE grid**", A. Monari, A. Scemama and M. Caffarel, *Remote Instrumentation for eScience and Related Aspects*, 195--207, Springer (2012).



- All I/O and network communications are non-blocking
- Worker: Single-core Fortran executable piped to a forwarder
- A Worker stops cleanly when its receives the *SIGTERM* signal

Fault-tolerance



- No access to the filesystem: scripts, binary and input data are broadcasted to the client nodes and stored in */dev/shm*. Local disks can crash.
- Blocks have a Gaussian distribution. Losing blocks doesn't change the average. Any worker can be removed.
- Every forwarder can always reach the data server. Any node can be removed.
- If the data server is lost, it is always possible to continue the calculation in another run.

Parallelization of DMC

- In the standard DMC algorithm, the walkers are no more independent.
- Communications are expected to kill the ideal speed-up.
- The Pure Diffusion MC algorithm [§] allows to obtain the DMC energy with re-weighting instead of branching: no more communication.
- PDMC introduces too much fluctuations in the total energies
- We use an algorithm that combines branching and re-weighting. [¶] Small populations can be used, and multiple independent runs can be done.

---

§ "**Development of a pure diffusion quantum Monte Carlo method using a full generalized Feynman–Kac formula.**", M. Caffarel, *J. Chem. Phys.* 88, 1088 (1988)

¶ "**Diffusion monte carlo methods with a fixed number of walkers**", R. Assaraf, M. Caffarel, A. Khelif, *Phys Rev E* 61(4 Pt B), 4566-75 (2000).

Why a database?

- Input and output data are tightly linked
- An output file can be generated on demand
- Easy connection to GUI
- An API simplifies scripting to manipulate results
- Checkpoint/restart is trivial
- Additional calculation can be done even if the calculation is finished. No need to re-run.
- Combining results obtained on different datacenters is trivial
- Multiple independent runs can write in the same database : dynamic number of CPUs.
- The name of the database is an MD5 key, corresponding to critical input data.

Initial conditions

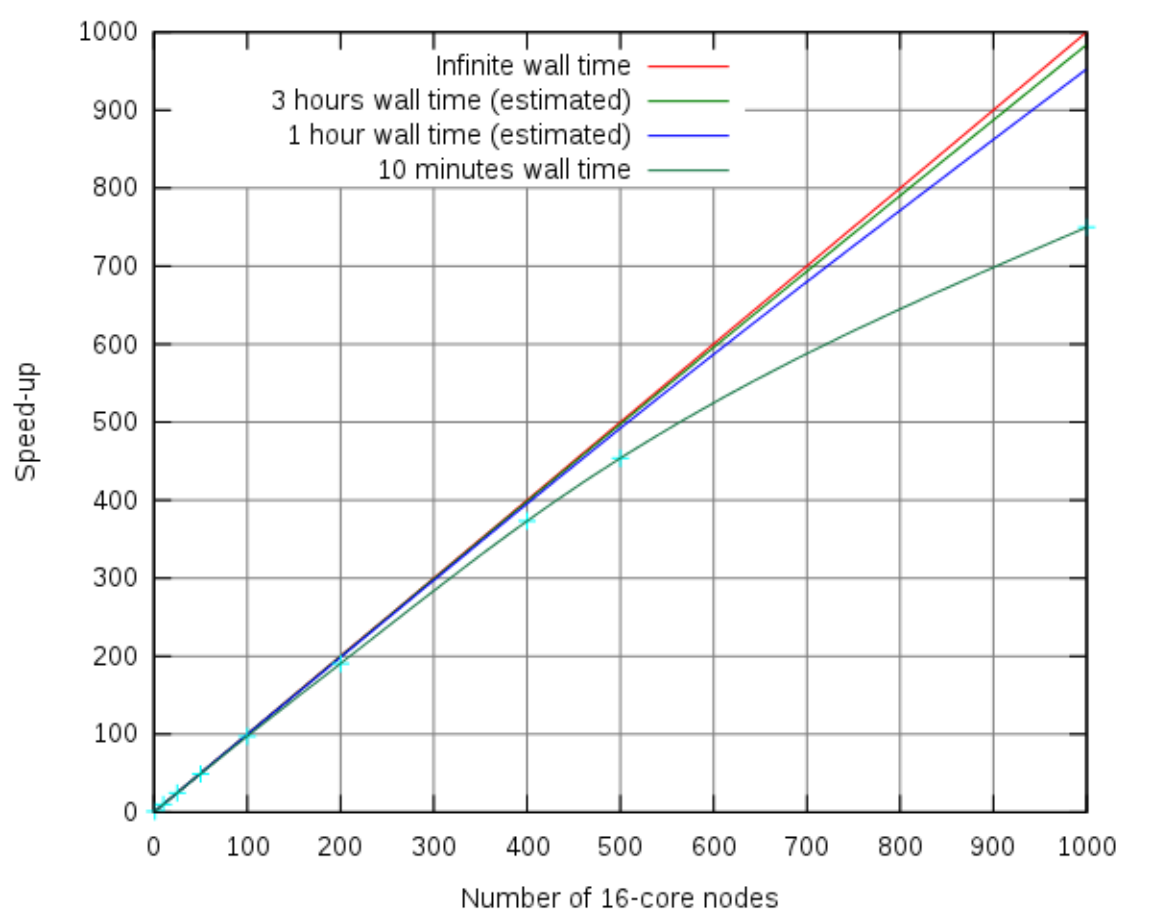
- Different initial walker positions are needed
- At the end of each block, the final positions are sent to the forwarder
- Each forwarder keeps a sample of the populations of its workers
- Sometimes, the forwarder sends its walkers to its parent in the tree
- The data server receives a sample of the population of all the walkers and merges it with its population
- Periodically, the population is saved to disk
- When a new run is started, each worker gets N_{walk} walkers drawn randomly from this population

Termination

When the manager wants to terminate the calculation (catches *SIGTERM*, or termination condition reached):

- It sends to the leaves of the tree a termination signal
- The leaves send a *SIGTERM* to the workers
- Each forwarder gets the data of the last blocks from the workers
- When the workers have terminated, the forwarder sends the data to its parent with a termination signal
- When the data server receives the termination signal, it the calculation is finished

Parallel speed-up



Estimation checked on 100 nodes/1 hour. Accuracy of 0.9992

The parallel speed-up is almost ideal.

Single-core optimization is crucial : Every percent gained on one core will be gained on the parallel simulation

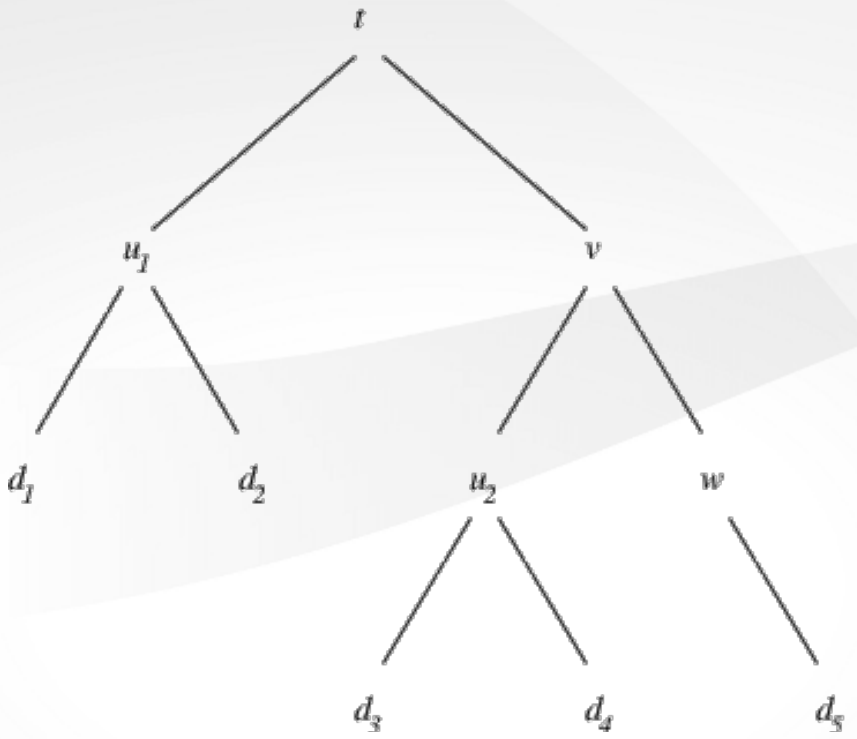
Easy and effficient programming with IRPF90

Anthony Scemama <scemama@irsamc.ups-tlse.fr>
Michel Caffarel <michel.caffarel@irsamc.ups-tlse.fr>

Labratoire de Chimie et Physique Quantiques
IRSAMC (Toulouse)

Introduction

- A program is a function of its input data: output = program(input)
- A program can be represented as a tree where:
- the vertices are the variables
- the edges represent the relation '*depends on*'
- The root of the tree is the output of the program
- The leaves are the input data



u (x, y) = x + y + 1
v (x, y) = x + y + 2
w (x) = x + 3
t (x, y) = x + y + 4
This production tree computes
t(u(d_1, d_2), v(u(d_3, d_4), w(d_5)))

Usual programming

```
program exemple_1
   implicit none
   integer :: d1,d2,d3,d4,d5  ! Input data
   integer :: u1, u2, w, v    ! Temporary variables
   integer :: t               ! Output data

   call read_data(d1,d2,d3,d4,d5)
   call compute_u(d1,d2,u1)
   call compute_u(d3,d4,u2)
   call compute_w(d5,w)
   call compute_v(u2,w,v)
   call compute_t(u1,v,t)

   print * , 't=', t
end program
```

Alternative way with functions

```
program exemple_2
   implicit none
   integer :: d1,d2,d3,d4,d5  ! Input data
   integer :: u1, u2, w, v, t ! Variables
   integer :: compute_u,compute_t,compute_w,compute_w

   call read_data(d1,d2,d3,d4,d5)
   u1 = compute_u(d1,d2)
   u2 = compute_u(d3,d4)
   w  = compute_w(d5)
   v  = compute_v(u2,w)
   t  = compute_t(u1,v)

   print * , 't=', t
end program
```

Single-line with functions

```
program exemple_3
   implicit none
   integer :: d1,d2,d3,d4,d5  ! Input data
   integer :: u, v, w, t

   call read_data(d1,d2,d3,d4,d5)

   print * , 't=', &
     t( u(d1,d2), v( u(d3,d4), w(d5) ) )
end program
```

Now, the sequence of execution is handled by the compiler.

Same example with IRPF90

```
program exemple_4
   implicit none
   print * , 't=', t
end program
```

That's it!

- Using *t* triggers the exploration of the production tree
- Completely equivalent to the previous example, but the parameters of the function *t* are not expressed
- IRP : Implicit Reference to Parameters

Definition of the nodes of the tree

For each node, we write a **provider**. This is a subroutine whose role is to build the variable *and* guarantee that it is built properly.

file: *uvwt.irp.f*

```
BEGIN_PROVIDER [ integer, t ]
  t = u1+v+4
END_PROVIDER

BEGIN_PROVIDER [integer,w]
  w = d5+3
END_PROVIDER

BEGIN_PROVIDER [ integer, v ]
  v = u2+w+2
END_PROVIDER
```

```
BEGIN_PROVIDER [ integer, u1 ]
  u1 = d1+d2+1
END_PROVIDER

BEGIN_PROVIDER [ integer, u2 ]
  u2 = d3+d4+1
END_PROVIDER
```

file : *input.irp.f*

```
 BEGIN_PROVIDER [ integer, d1 ]
&BEGIN_PROVIDER [ integer, d2 ]
&BEGIN_PROVIDER [ integer, d3 ]
&BEGIN_PROVIDER [ integer, d4 ]
&BEGIN_PROVIDER [ integer, d5 ]
 read(*,*) d1
 read(*,*) d2
 read(*,*) d3
 read(*,*) d4
 read(*,*) d5
END_PROVIDER
```

When you write a provider for *x*, you **only** have to focus on

- How do I build *x*?
- What are the variables that I need to build *x*?
- Am I sure that *x* is built correctly when I exit the provider?

Using this method:

- You don't have to know the execution sequence
- If you need a variable (node), you are *sure* that it has been built properly when you use it
- You will never break other parts of the program
- Many people can work simultaneously on the same program with minimal effort
- If a node has already been built, it will not be built again. The correct value will be returned by the provider.

Fortran code generation

- Run *irpf90* in the current directory
- *irpf90* reads all the **.irp.f* files
- All the providers are identified
- All the corresponding variables (IRP entities) are searched for in the code
- The dependence tree is built
- Providers are transformed to subroutines (*subroutine provide_**)
- Calls to *provide_** are inserted in the code
- Each file **.irp.f* generates a module containing the IRP entities, and a Fortran file containing the subroutines/functions
- As the dependence tree is built, the dependences between the files are known and the *makefile* is built automatically

Generated code example

```
! -*- F90 -*-
!
!-----------------------------------------------!
! This file was generated with the irpf90 tool. !
!                                               !
!           DO NOT MODIFY IT BY HAND            !
!-----------------------------------------------!

program irp_program                              ! irp_example1:   0
 call irp_example1                               ! irp_example1.irp.f:   0
 call irp_finalize_742559343()                   ! irp_example1.irp.f:   0
end program                                      ! irp_example1.irp.f:   0
subroutine irp_example1                          ! irp_example1.irp.f:   1
  use uvwt_mod
  implicit none                                  ! irp_example1.irp.f:   2
  character*(12) :: irp_here = 'irp_example1' ! irp_example1.irp.f:   1
```

```
  if (.not.t_is_built) then
    call provide_t
  endif
  print *, 't = ', t                                  ! irp_example1.irp.f:   3
end                                                   ! irp_example1.irp.f:   4
```

```
! -*- F90 -*-
!
!-----------------------------------------------!
! This file was generated with the irpf90 tool. !
!                                               !
!           DO NOT MODIFY IT BY HAND            !
!-----------------------------------------------!

module uvwt_mod
  integer :: u1
  logical :: u1_is_built = .False.
  integer :: u2
  logical :: u2_is_built = .False.
```

```
  integer :: t
  logical :: t_is_built = .False.
  integer :: w
  logical :: w_is_built = .False.
  integer :: v
  logical :: v_is_built = .False.
end module uvwt_mod
```

```
! -*- F90 -*-
!
!-----------------------------------------------!
! This file was generated with the irpf90 tool. !
!                                               !
!           DO NOT MODIFY IT BY HAND            !
!-----------------------------------------------!

subroutine provide_u1
  use uvwt_mod
```

```
  use input_mod
  implicit none
  character*(10) :: irp_here = 'provide_u1'
  integer                    :: irp_err
  logical                    :: irp_dimensions_OK
  if (.not.d1_is_built) then
    call provide_d1
  endif
 if (.not.u1_is_built) then
  call bld_u1
  u1_is_built = .True.

 endif
end subroutine provide_u1

subroutine bld_u1
  use uvwt_mod
  use input_mod
```

```
  character*(2) :: irp_here = 'u1'                    ! uvwt.irp.f:  13
  u1 = d1+d2+1                                        ! uvwt.irp.f:  14
end subroutine bld_u1
subroutine provide_u2
  use uvwt_mod
  use input_mod
  implicit none
  character*(10) :: irp_here = 'provide_u2'
  integer                    :: irp_err
  logical                    :: irp_dimensions_OK
  if (.not.d1_is_built) then
    call provide_d1
  endif
 if (.not.u2_is_built) then
  call bld_u2
  u2_is_built = .True.

 endif
```

```
end subroutine provide_u2

subroutine bld_u2
  use uvwt_mod
  use input_mod
  character*(2) :: irp_here = 'u2'                ! uvwt.irp.f:  17
  u2 = d3+d4+1                                    ! uvwt.irp.f:  18
end subroutine bld_u2
subroutine provide_t
  use uvwt_mod
  implicit none
  character*(9) :: irp_here = 'provide_t'
  integer                        :: irp_err
  logical                        :: irp_dimensions_OK
  if (.not.u1_is_built) then
    call provide_u1
  endif
  if (.not.v_is_built) then
```

```
    call provide_v
  endif
 if (.not.t_is_built) then
  call bld_t
  t_is_built = .True.

 endif
end subroutine provide_t

subroutine bld_t
  use uvwt_mod
  character*(1) :: irp_here = 't'                    ! uvwt.irp.f:   1
  t = u1+v+4                                         ! uvwt.irp.f:   2
end subroutine bld_t
subroutine provide_w
  use uvwt_mod
  use input_mod
  implicit none
```

```
  character*(9) :: irp_here = 'provide_w'
  integer                   :: irp_err
  logical                   :: irp_dimensions_OK
  if (.not.d1_is_built) then
    call provide_d1
  endif
 if (.not.w_is_built) then
  call bld_w
  w_is_built = .True.

 endif
end subroutine provide_w

subroutine bld_w
  use uvwt_mod
  use input_mod
  character*(1) :: irp_here = 'w'                ! uvwt.irp.f:   5
  w = d5+3                                       ! uvwt.irp.f:   6
```

```
end subroutine bld_w
subroutine provide_v
  use uvwt_mod
  implicit none
  character*(9) :: irp_here = 'provide_v'
  integer                    :: irp_err
  logical                    :: irp_dimensions_OK
  if (.not.w_is_built) then
    call provide_w
  endif
  if (.not.u2_is_built) then
    call provide_u2
  endif
 if (.not.v_is_built) then
  call bld_v
  v_is_built = .True.

 endif
```

```
end subroutine provide_v

subroutine bld_v
  use uvwt_mod
  character*(1) :: irp_here = 'v'                    ! uvwt.irp.f:   9
  v = u2+w+2                                         ! uvwt.irp.f:  10
end subroutine bld_v
```

Code execution with debug mode on:

```
$ ./irp_example1
            0 : -> provide_t
            0 :  -> provide_u1
            0 :   -> provide_d1
            0 :    -> d1
1
2
3
4
```

```
5
          0 :    <- d1   0.000000000000000
          0 :   <- provide_d1   0.000000000000000
          0 :   -> u1
          0 :   <- u1   0.000000000000000
          0 :  <- provide_u1   0.000000000000000
          0 :  -> provide_v
          0 :   -> provide_w
          0 :    -> w
          0 :    <- w   0.000000000000000
          0 :   <- provide_w   0.000000000000000
          0 :   -> provide_u2
          0 :    -> u2
          0 :    <- u2   0.000000000000000
          0 :   <- provide_u2   0.000000000000000
          0 :   -> v
          0 :   <- v   0.000000000000000
          0 :  <- provide_v   0.000000000000000
```

```
           0 :  -> t
           0 :  <- t   0.0000000000000000
           0 : <- provide_t   0.0000000000000000
           0 : -> irp_example1
t =           26
           0 : <- irp_example1   0.0000000000000000
```

Using subroutines/functions

```
BEGIN_PROVIDER [ integer, u1 ]
  integer :: fu
  u1 = fu(d1,d2)
END_PROVIDER

BEGIN_PROVIDER [ integer, u2 ]
  integer :: fu
  u2 = fu(d3,d4)
END_PROVIDER

integer function fu(x,y)
  integer :: x,y
  fu = x+y+1
end function
```

Providing arrays

An array is considered built when all its elements are built. Its dimensions can be provided variables, constants and intervals (a:b).

```
BEGIN_PROVIDER [ integer, fact_max ]
 fact_max = 10
END_PROVIDER

BEGIN_PROVIDER [ double precision, fact, (0:fact_max) ]
  integer :: i

  fact(0) = 1.d0
  do i=1,fact_max
    fact(i) = fact(i-1)*dble(i)
  enddo
END_PROVIDER
```

```
program test
  print *,  fact(5)
end
```

```
$ ./test
           0 : -> provide_fact
           0 :  -> provide_fact_max
           0 :   -> fact_max
           0 :   <- fact_max   0.000000000000000
           0 :  <- provide_fact_max   0.000000000000000
           0 :  -> fact
           0 :  <- fact   0.000000000000000
           0 : <- provide_fact   0.000000000000000
           0 : -> test
   120.00000000000000
           0 : <- test   0.000000000000000
```

The allocation behaves as follows:

- If the array is not already allocated, it is allocated

- If the array already allocated, check if the dimensions have changed
- If the dimensions have not changed, then OK.
- Else deallocate the array and re-allocate it with the correct dimensions
- All allocations/deallocations are checked with *stat=err*

```
! -*- F90 -*-
!
!-----------------------------------------------!
! This file was generated with the irpf90 tool. !
!                                               !
!           DO NOT MODIFY IT BY HAND            !
!-----------------------------------------------!

subroutine provide_fact_max
  use fact_mod
  implicit none
  character*(16) :: irp_here = 'provide_fact_max'
  integer                    :: irp_err
```

```
  logical                        :: irp_dimensions_OK
 if (.not.fact_max_is_built) then
  call bld_fact_max
  fact_max_is_built = .True.

 endif
end subroutine provide_fact_max

subroutine bld_fact_max
  use fact_mod
  character*(8) :: irp_here = 'fact_max'      ! fact.irp.f:   1
 fact_max = 10                                ! fact.irp.f:   2
end subroutine bld_fact_max
subroutine provide_fact
  use fact_mod
  implicit none
  character*(12) :: irp_here = 'provide_fact'
  integer                        :: irp_err
```

```
 logical                         :: irp_dimensions_OK
 if (.not.fact_max_is_built) then
   call provide_fact_max
 endif
if (allocated (fact) ) then
 irp_dimensions_OK = .True.
 irp_dimensions_OK = irp_dimensions_OK.AND. &
   (SIZE(fact,1)==(fact_max - (-1)))
 if (.not.irp_dimensions_OK) then
  deallocate(fact,stat=irp_err)
  if (irp_err /= 0) then
    print *, irp_here//': Deallocation failed: fact'
    print *, ' size: (0:fact_max)'
  endif
  if ((fact_max - (-1)>0)) then
   allocate(fact(0:fact_max),stat=irp_err)
   if (irp_err /= 0) then
    print *, irp_here//': Allocation failed: fact'
```

```
     print *, ' size: (0:fact_max)'
    endif
   endif
  endif
else
   if ((fact_max - (-1)>0)) then
    allocate(fact(0:fact_max),stat=irp_err)
    if (irp_err /= 0) then
     print *, irp_here//': Allocation failed: fact'
     print *, ' size: (0:fact_max)'
    endif
   endif
endif
if (.not.fact_is_built) then
 call bld_fact
 fact_is_built = .True.

endif
```

```
end subroutine provide_fact

subroutine bld_fact
  use fact_mod
  character*(4) :: irp_here = 'fact'          ! fact.irp.f:   5
  integer :: i                                ! fact.irp.f:   6
  fact(0) = 1.d0                              ! fact.irp.f:   8
  do i=1,fact_max                             ! fact.irp.f:   9
    fact(i) = fact(i-1)*dble(i)               ! fact.irp.f:  10
  enddo                                       ! fact.irp.f:  11
end subroutine bld_fact
```

Modifying a variable outside of its provider

In iterative processes, a variable needs to be modified outside of its provider. If it is the case, IRPF90 has to be informed of the change by the **TOUCH** keyword.

Example: computing numerical derivatives

```
BEGIN_PROVIDER [ real, dPsi ]
  x += 0.5*delta_x
  TOUCH x
  dPsi = Psi
  x -= delta_x
  TOUCH x
  dPsi = (dPsi - Psi)/delta_x
  x += 0.5*delta_x
  SOFT_TOUCH x
END_PROVIDER
```

Generated code:

```
! -*- F90 -*-
!
!------------------------------------------!
! This file was generated with the irpf90 tool. !
!                                          !
!           DO NOT MODIFY IT BY HAND       !
!------------------------------------------!

subroutine provide_dpsi
  use y_mod
  use x_mod
  implicit none
  character*(12) :: irp_here = 'provide_dpsi'
  integer                    :: irp_err
  logical                    :: irp_dimensions_OK
  if (.not.x_is_built) then
    call provide_x
```

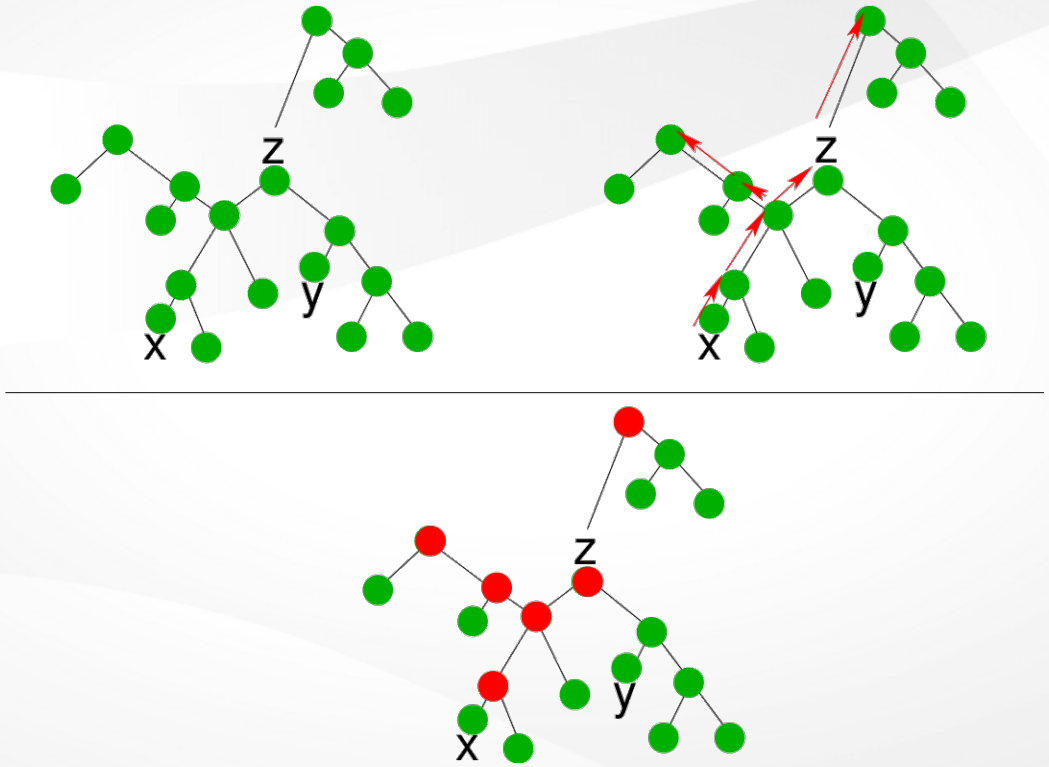
```
  endif
  if (.not.psi_is_built) then
    call provide_psi
  endif
  if (.not.delta_x_is_built) then
    call provide_delta_x
  endif
 if (.not.dpsi_is_built) then
  call bld_dpsi
  dpsi_is_built = .True.

 endif
end subroutine provide_dpsi

subroutine bld_dpsi
  use y_mod
  use x_mod
 use y_mod                                    ! x.irp.f:   3
```

```
 use y_mod                                                  ! x.irp.f:   6
 use y_mod                                                  ! x.irp.f:   9
  character*(4) :: irp_here = 'dpsi'                        ! x.irp.f:   1
  x =x +( 0.5*delta_x)                                      ! x.irp.f:   2
!                                                           ! x.irp.f:   3
! >>> TOUCH x                                              ! x.irp.f:   3
 call touch_x                                               ! x.irp.f:   3
! <<< END TOUCH                                            ! x.irp.f:   3
  if (.not.x_is_built) then
    call provide_x
  endif
  if (.not.psi_is_built) then
    call provide_psi
  endif
  if (.not.delta_x_is_built) then
    call provide_delta_x
  endif
  dPsi = Psi                                                ! x.irp.f:   4
```

```
  x =x -( delta_x)                                    ! x.irp.f:   5
!                                                     ! x.irp.f:   6
! >>> TOUCH x                                        ! x.irp.f:   6
 call touch_x                                         ! x.irp.f:   6
! <<< END TOUCH                                      ! x.irp.f:   6
  if (.not.x_is_built) then
    call provide_x
  endif
  if (.not.psi_is_built) then
    call provide_psi
  endif
  if (.not.delta_x_is_built) then
    call provide_delta_x
  endif
  dPsi = (dPsi - Psi)/delta_x                         ! x.irp.f:   7
  x =x +( 0.5*delta_x)                                ! x.irp.f:   8
!                                                     ! x.irp.f:   9
! >>> TOUCH x                                        ! x.irp.f:   9
```

```
 call touch_x                                          ! x.irp.f:   9
! <<< END TOUCH (Soft)                                 ! x.irp.f:   9
end subroutine bld_dpsi
```

How this works:



Templates

When pieces of code are very similar, it is possible to use a template:

```
BEGIN_TEMPLATE

 subroutine insertion_$Xsort (x,iorder,isize)
  implicit none
  $type,intent(inout)    :: x(isize)
  integer,intent(inout)  :: iorder(isize)
  integer,intent(in)     :: isize
  $type                  :: xtmp
  integer                :: i, i0, j, jmax

  do i=1,isize
   xtmp = x(i)
   i0 = iorder(i)
   j = i-1
```

```
   do j=i-1,1,-1
    if ( x(j) > xtmp ) then
     x(j+1) = x(j)
     iorder(j+1) = iorder(j)
    else
     exit
    endif
   enddo
   x(j+1) = xtmp
   iorder(j+1) = i0
  enddo

 end

SUBST [ X, type ]

   ; real ;;
 d ; double precision ;;
```

```
 i ; integer ;;

END_TEMPLATE
```

Generated code:

```
! -*- F90 -*-
!
!-----------------------------------------------!
! This file was generated with the irpf90 tool. !
!                               !
!           DO NOT MODIFY IT BY HAND  !
!-----------------------------------------------!

 subroutine insertion_sort (x,iorder,isize)   !x.irp.f_tpl_35:  3
  implicit none                               !x.irp.f_tpl_35:  4
  character*(14) :: irp_here='insertion_sort' !x.irp.f_tpl_35:  3
  real,intent(inout)    :: x(isize)           !x.irp.f_tpl_35:  5
  integer,intent(inout)  :: iorder(isize)     !x.irp.f_tpl_35:  6
```

```
integer,intent(in)     :: isize                !x.irp.f_tpl_35:  7
real                   :: xtmp                 !x.irp.f_tpl_35:  8
integer                :: i, i0, j, jmax       !x.irp.f_tpl_35:  9
do i=1,isize                                   !x.irp.f_tpl_35: 11
 xtmp = x(i)                                   !x.irp.f_tpl_35: 12
 i0 = iorder(i)                                !x.irp.f_tpl_35: 13
 j = i-1                                       !x.irp.f_tpl_35: 14
 do j=i-1,1,-1                                 !x.irp.f_tpl_35: 15
  if ( x(j) > xtmp ) then                      !x.irp.f_tpl_35: 16
   x(j+1) = x(j)                               !x.irp.f_tpl_35: 17
   iorder(j+1) = iorder(j)                     !x.irp.f_tpl_35: 18
  else                                         !x.irp.f_tpl_35: 19
   exit                                        !x.irp.f_tpl_35: 20
  endif                                        !x.irp.f_tpl_35: 21
 enddo                                         !x.irp.f_tpl_35: 22
 x(j+1) = xtmp                                 !x.irp.f_tpl_35: 23
 iorder(j+1) = i0                              !x.irp.f_tpl_35: 24
enddo                                          !x.irp.f_tpl_35: 25
```

```
end                                                    !x.irp.f_tpl_35: 27
subroutine insertion_dsort (x,iorder,isize)            !x.irp.f_tpl_35: 32
 implicit none                                         !x.irp.f_tpl_35: 33
 character*(15) :: irp_here='insertion_dsort'          !x.irp.f_tpl_35: 32
 double precision,intent(inout)   :: x(isize)          !x.irp.f_tpl_35: 34
 integer,intent(inout)  :: iorder(isize)               !x.irp.f_tpl_35: 35
 integer,intent(in)     :: isize                       !x.irp.f_tpl_35: 36
 double precision                  :: xtmp             !x.irp.f_tpl_35: 37
 integer                :: i, i0, j, jmax              !x.irp.f_tpl_35: 38
 do i=1,isize                                          !x.irp.f_tpl_35: 40
  xtmp = x(i)                                          !x.irp.f_tpl_35: 41
  i0 = iorder(i)                                       !x.irp.f_tpl_35: 42
  j = i-1                                              !x.irp.f_tpl_35: 43
  do j=i-1,1,-1                                        !x.irp.f_tpl_35: 44
   if ( x(j) > xtmp ) then                             !x.irp.f_tpl_35: 45
    x(j+1) = x(j)                                      !x.irp.f_tpl_35: 46
    iorder(j+1) = iorder(j)                            !x.irp.f_tpl_35: 47
   else                                                !x.irp.f_tpl_35: 48
```

```
    exit                                                !x.irp.f_tpl_35: 49
   endif                                                !x.irp.f_tpl_35: 50
  enddo                                                 !x.irp.f_tpl_35: 51
  x(j+1) = xtmp                                         !x.irp.f_tpl_35: 52
  iorder(j+1) = i0                                      !x.irp.f_tpl_35: 53
 enddo                                                  !x.irp.f_tpl_35: 54
end                                                     !x.irp.f_tpl_35: 56
subroutine insertion_isort (x,iorder,isize)             !x.irp.f_tpl_35: 61
 implicit none                                          !x.irp.f_tpl_35: 62
 character*(15) :: irp_here='insertion_isort'           !x.irp.f_tpl_35: 61
 integer,intent(inout)    :: x(isize)                   !x.irp.f_tpl_35: 63
 integer,intent(inout)  :: iorder(isize)                !x.irp.f_tpl_35: 64
 integer,intent(in)     :: isize                        !x.irp.f_tpl_35: 65
 integer                  :: xtmp                       !x.irp.f_tpl_35: 66
 integer                :: i, i0, j, jmax               !x.irp.f_tpl_35: 67
 do i=1,isize                                           !x.irp.f_tpl_35: 69
  xtmp = x(i)                                           !x.irp.f_tpl_35: 70
  i0 = iorder(i)                                        !x.irp.f_tpl_35: 71
```

```
   j = i-1                                                !x.irp.f_tpl_35: 72
   do j=i-1,1,-1                                          !x.irp.f_tpl_35: 73
    if ( x(j) > xtmp ) then                               !x.irp.f_tpl_35: 74
     x(j+1) = x(j)                                        !x.irp.f_tpl_35: 75
     iorder(j+1) = iorder(j)                              !x.irp.f_tpl_35: 76
    else                                                  !x.irp.f_tpl_35: 77
     exit                                                 !x.irp.f_tpl_35: 78
    endif                                                 !x.irp.f_tpl_35: 79
   enddo                                                  !x.irp.f_tpl_35: 80
   x(j+1) = xtmp                                          !x.irp.f_tpl_35: 81
   iorder(j+1) = i0                                       !x.irp.f_tpl_35: 82
  enddo                                                   !x.irp.f_tpl_35: 83
end                                                       !x.irp.f_tpl_35: 85
```

Metaprogramming

Shell scripts can be inserted in the IRPF90 code, and the output of the script will be inserted in the generated Fortran. For example:

```
program test
 BEGIN_SHELL [ /bin/bash ]
  echo print *, \'Compiled by $(whoami) on $(date)\'
 END_SHELL
end
```

Generated code:

```
! -*- F90 -*-
!
!-----------------------------------------------!
! This file was generated with the irpf90 tool. !
!                                               !
!           DO NOT MODIFY IT BY HAND            !
```

```
!-----------------------------------------------!

program irp_program                                                    ! test:   0
 call test                                                             ! test.irp.f:   0
 call irp_finalize_491024427()                                         ! test.irp.f:   0
end program                                                            ! test.irp.f:   0
subroutine test                                                        ! test.irp.f:   1
  character*(4) :: irp_here = 'test'                                   ! test.irp.f:   1
print *, 'Compiled by scemama on Mon Jul 8 11:28:16 CEST 2013'         ! test.irp.f_shell_4:   1
end                                                                    ! test.irp.f:   5
```

Example: Computing powers of x

```
BEGIN_SHELL [ /usr/bin/python ]

POWER_MAX = 20

def compute_x_prod(n,d):
  if n == 0:
    d[0] = None
    return d
  if n == 1:
    d[1] = None
```

```
    return d
  if n in d:
    return d
  m = n/2
  d = compute_x_prod(m,d)
  d[n] = None
  d[2*m] = None
  return d

def print_subroutine(n):
  keys = compute_x_prod(n,{}).keys()
  keys.sort()
  output = []
  print "real function power_%d(x1)"%n
  print " real, intent(in) :: x1"
  for i in range(1,len(keys)):
     output.append( "x%d"%keys[i] )
  if output != []:
```

```
    print " real :: "+', '.join(output)
  for i in range(1,len(keys)):
   ki = keys[i]
   ki1 = keys[i-1]
   if ki == 2*ki1:
     print " x%d"%ki + " = x%d * x%d"%(ki1,ki1)
   else:
     print " x%d"%ki + " = x%d * x1"%(ki1)
  print " power_%d = x%d"%(n,n)
  print "end"

for i in range(POWER_MAX):
  print_subroutine (i+1)
  print ''

END_SHELL
```

```
! -*- F90 -*-
!
!-----------------------------------------------!
! This file was generated with the irpf90 tool. !
!                                               !
!           DO NOT MODIFY IT BY HAND            !
!-----------------------------------------------!

real function power_1(x1)                    ! power.irp.f_shell_44:   1
  character*(7) :: irp_here = 'power_1'      ! power.irp.f_shell_44:   1
 real, intent(in) :: x1                      ! power.irp.f_shell_44:   2
 power_1 = x1                                ! power.irp.f_shell_44:   3
end                                          ! power.irp.f_shell_44:   4
real function power_2(x1)                    ! power.irp.f_shell_44:   6
  character*(7) :: irp_here = 'power_2'      ! power.irp.f_shell_44:   6
 real, intent(in) :: x1                      ! power.irp.f_shell_44:   7
 real :: x2                                  ! power.irp.f_shell_44:   8
 x2 = x1 * x1                                ! power.irp.f_shell_44:   9
```

```
 power_2 = x2                                  ! power.irp.f_shell_44:  10
end                                            ! power.irp.f_shell_44:  11
real function power_3(x1)                      ! power.irp.f_shell_44:  13
  character*(7) :: irp_here = 'power_3'        ! power.irp.f_shell_44:  13
 real, intent(in) :: x1                        ! power.irp.f_shell_44:  14
 real :: x2, x3                                ! power.irp.f_shell_44:  15
 x2 = x1 * x1                                  ! power.irp.f_shell_44:  16
 x3 = x2 * x1                                  ! power.irp.f_shell_44:  17
 power_3 = x3                                  ! power.irp.f_shell_44:  18
end                                            ! power.irp.f_shell_44:  19
real function power_4(x1)                      ! power.irp.f_shell_44:  21
  character*(7) :: irp_here = 'power_4'        ! power.irp.f_shell_44:  21
 real, intent(in) :: x1                        ! power.irp.f_shell_44:  22
 real :: x2, x4                                ! power.irp.f_shell_44:  23
 x2 = x1 * x1                                  ! power.irp.f_shell_44:  24
 x4 = x2 * x2                                  ! power.irp.f_shell_44:  25
 power_4 = x4                                  ! power.irp.f_shell_44:  26
end                                            ! power.irp.f_shell_44:  27
```

```
real function power_5(x1)                    ! power.irp.f_shell_44:  29
  character*(7) :: irp_here = 'power_5'      ! power.irp.f_shell_44:  29
 real, intent(in) :: x1                      ! power.irp.f_shell_44:  30
 real :: x2, x4, x5                          ! power.irp.f_shell_44:  31
 x2 = x1 * x1                                ! power.irp.f_shell_44:  32
 x4 = x2 * x2                                ! power.irp.f_shell_44:  33
 x5 = x4 * x1                                ! power.irp.f_shell_44:  34
 power_5 = x5                                ! power.irp.f_shell_44:  35
end                                          ! power.irp.f_shell_44:  36
real function power_6(x1)                    ! power.irp.f_shell_44:  38
  character*(7) :: irp_here = 'power_6'      ! power.irp.f_shell_44:  38
 real, intent(in) :: x1                      ! power.irp.f_shell_44:  39
 real :: x2, x3, x6                          ! power.irp.f_shell_44:  40
 x2 = x1 * x1                                ! power.irp.f_shell_44:  41
 x3 = x2 * x1                                ! power.irp.f_shell_44:  42
 x6 = x3 * x3                                ! power.irp.f_shell_44:  43
 power_6 = x6                                ! power.irp.f_shell_44:  44
end                                          ! power.irp.f_shell_44:  45
```

```
real function power_7(x1)                    ! power.irp.f_shell_44:  47
  character*(7) :: irp_here = 'power_7'      ! power.irp.f_shell_44:  47
 real, intent(in) :: x1                      ! power.irp.f_shell_44:  48
 real :: x2, x3, x6, x7                      ! power.irp.f_shell_44:  49
 x2 = x1 * x1                                ! power.irp.f_shell_44:  50
 x3 = x2 * x1                                ! power.irp.f_shell_44:  51
 x6 = x3 * x3                                ! power.irp.f_shell_44:  52
 x7 = x6 * x1                                ! power.irp.f_shell_44:  53
 power_7 = x7                                ! power.irp.f_shell_44:  54
end                                          ! power.irp.f_shell_44:  55
real function power_8(x1)                    ! power.irp.f_shell_44:  57
  character*(7) :: irp_here = 'power_8'      ! power.irp.f_shell_44:  57
 real, intent(in) :: x1                      ! power.irp.f_shell_44:  58
 real :: x2, x4, x8                          ! power.irp.f_shell_44:  59
 x2 = x1 * x1                                ! power.irp.f_shell_44:  60
 x4 = x2 * x2                                ! power.irp.f_shell_44:  61
 x8 = x4 * x4                                ! power.irp.f_shell_44:  62
 power_8 = x8                                ! power.irp.f_shell_44:  63
```

```
end                                                ! power.irp.f_shell_44:  64
real function power_9(x1)                          ! power.irp.f_shell_44:  66
  character*(7) :: irp_here = 'power_9'            ! power.irp.f_shell_44:  66
 real, intent(in) :: x1                            ! power.irp.f_shell_44:  67
 real :: x2, x4, x8, x9                            ! power.irp.f_shell_44:  68
 x2 = x1 * x1                                      ! power.irp.f_shell_44:  69
 x4 = x2 * x2                                      ! power.irp.f_shell_44:  70
 x8 = x4 * x4                                      ! power.irp.f_shell_44:  71
 x9 = x8 * x1                                      ! power.irp.f_shell_44:  72
 power_9 = x9                                      ! power.irp.f_shell_44:  73
end                                                ! power.irp.f_shell_44:  74
real function power_10(x1)                         ! power.irp.f_shell_44:  76
  character*(8) :: irp_here = 'power_10'           ! power.irp.f_shell_44:  76
 real, intent(in) :: x1                            ! power.irp.f_shell_44:  77
 real :: x2, x4, x5, x10                           ! power.irp.f_shell_44:  78
 x2 = x1 * x1                                      ! power.irp.f_shell_44:  79
 x4 = x2 * x2                                      ! power.irp.f_shell_44:  80
 x5 = x4 * x1                                      ! power.irp.f_shell_44:  81
```

```
 x10 = x5 * x5                                  ! power.irp.f_shell_44:  82
 power_10 = x10                                 ! power.irp.f_shell_44:  83
end                                             ! power.irp.f_shell_44:  84
real function power_11(x1)                      ! power.irp.f_shell_44:  86
  character*(8) :: irp_here = 'power_11'        ! power.irp.f_shell_44:  86
 real, intent(in) :: x1                         ! power.irp.f_shell_44:  87
 real :: x2, x4, x5, x10, x11                   ! power.irp.f_shell_44:  88
 x2 = x1 * x1                                   ! power.irp.f_shell_44:  89
 x4 = x2 * x2                                   ! power.irp.f_shell_44:  90
 x5 = x4 * x1                                   ! power.irp.f_shell_44:  91
 x10 = x5 * x5                                  ! power.irp.f_shell_44:  92
 x11 = x10 * x1                                 ! power.irp.f_shell_44:  93
 power_11 = x11                                 ! power.irp.f_shell_44:  94
end                                             ! power.irp.f_shell_44:  95
real function power_12(x1)                      ! power.irp.f_shell_44:  97
  character*(8) :: irp_here = 'power_12'        ! power.irp.f_shell_44:  97
 real, intent(in) :: x1                         ! power.irp.f_shell_44:  98
 real :: x2, x3, x6, x12                        ! power.irp.f_shell_44:  99
```

```
 x2 = x1 * x1                                    ! power.irp.f_shell_44: 100
 x3 = x2 * x1                                    ! power.irp.f_shell_44: 101
 x6 = x3 * x3                                    ! power.irp.f_shell_44: 102
 x12 = x6 * x6                                   ! power.irp.f_shell_44: 103
 power_12 = x12                                  ! power.irp.f_shell_44: 104
end                                              ! power.irp.f_shell_44: 105
real function power_13(x1)                       ! power.irp.f_shell_44: 107
  character*(8) :: irp_here = 'power_13'         ! power.irp.f_shell_44: 107
 real, intent(in) :: x1                          ! power.irp.f_shell_44: 108
 real :: x2, x3, x6, x12, x13                    ! power.irp.f_shell_44: 109
 x2 = x1 * x1                                    ! power.irp.f_shell_44: 110
 x3 = x2 * x1                                    ! power.irp.f_shell_44: 111
 x6 = x3 * x3                                    ! power.irp.f_shell_44: 112
 x12 = x6 * x6                                   ! power.irp.f_shell_44: 113
 x13 = x12 * x1                                  ! power.irp.f_shell_44: 114
 power_13 = x13                                  ! power.irp.f_shell_44: 115
end                                              ! power.irp.f_shell_44: 116
real function power_14(x1)                       ! power.irp.f_shell_44: 118
```

```
  character*(8) :: irp_here = 'power_14' ! power.irp.f_shell_44: 118
 real, intent(in) :: x1                  ! power.irp.f_shell_44: 119
 real :: x2, x3, x6, x7, x14             ! power.irp.f_shell_44: 120
 x2 = x1 * x1                            ! power.irp.f_shell_44: 121
 x3 = x2 * x1                            ! power.irp.f_shell_44: 122
 x6 = x3 * x3                            ! power.irp.f_shell_44: 123
 x7 = x6 * x1                            ! power.irp.f_shell_44: 124
 x14 = x7 * x7                           ! power.irp.f_shell_44: 125
 power_14 = x14                          ! power.irp.f_shell_44: 126
end                                      ! power.irp.f_shell_44: 127
real function power_15(x1)               ! power.irp.f_shell_44: 129
  character*(8) :: irp_here = 'power_15' ! power.irp.f_shell_44: 129
 real, intent(in) :: x1                  ! power.irp.f_shell_44: 130
 real :: x2, x3, x6, x7, x14, x15        ! power.irp.f_shell_44: 131
 x2 = x1 * x1                            ! power.irp.f_shell_44: 132
 x3 = x2 * x1                            ! power.irp.f_shell_44: 133
 x6 = x3 * x3                            ! power.irp.f_shell_44: 134
 x7 = x6 * x1                            ! power.irp.f_shell_44: 135
```

```
 x14 = x7 * x7                                    ! power.irp.f_shell_44: 136
 x15 = x14 * x1                                   ! power.irp.f_shell_44: 137
 power_15 = x15                                   ! power.irp.f_shell_44: 138
end                                               ! power.irp.f_shell_44: 139
real function power_16(x1)                        ! power.irp.f_shell_44: 141
  character*(8) :: irp_here = 'power_16'          ! power.irp.f_shell_44: 141
 real, intent(in) :: x1                           ! power.irp.f_shell_44: 142
 real :: x2, x4, x8, x16                          ! power.irp.f_shell_44: 143
 x2 = x1 * x1                                     ! power.irp.f_shell_44: 144
 x4 = x2 * x2                                     ! power.irp.f_shell_44: 145
 x8 = x4 * x4                                     ! power.irp.f_shell_44: 146
 x16 = x8 * x8                                    ! power.irp.f_shell_44: 147
 power_16 = x16                                   ! power.irp.f_shell_44: 148
end                                               ! power.irp.f_shell_44: 149
real function power_17(x1)                        ! power.irp.f_shell_44: 151
  character*(8) :: irp_here = 'power_17'          ! power.irp.f_shell_44: 151
 real, intent(in) :: x1                           ! power.irp.f_shell_44: 152
 real :: x2, x4, x8, x16, x17                     ! power.irp.f_shell_44: 153
```

```
 x2 = x1 * x1                                    ! power.irp.f_shell_44: 154
 x4 = x2 * x2                                    ! power.irp.f_shell_44: 155
 x8 = x4 * x4                                    ! power.irp.f_shell_44: 156
 x16 = x8 * x8                                   ! power.irp.f_shell_44: 157
 x17 = x16 * x1                                  ! power.irp.f_shell_44: 158
 power_17 = x17                                  ! power.irp.f_shell_44: 159
end                                              ! power.irp.f_shell_44: 160
real function power_18(x1)                       ! power.irp.f_shell_44: 162
  character*(8) :: irp_here = 'power_18'         ! power.irp.f_shell_44: 162
 real, intent(in) :: x1                          ! power.irp.f_shell_44: 163
 real :: x2, x4, x8, x9, x18                     ! power.irp.f_shell_44: 164
 x2 = x1 * x1                                    ! power.irp.f_shell_44: 165
 x4 = x2 * x2                                    ! power.irp.f_shell_44: 166
 x8 = x4 * x4                                    ! power.irp.f_shell_44: 167
 x9 = x8 * x1                                    ! power.irp.f_shell_44: 168
 x18 = x9 * x9                                   ! power.irp.f_shell_44: 169
 power_18 = x18                                  ! power.irp.f_shell_44: 170
end                                              ! power.irp.f_shell_44: 171
```

```
real function power_19(x1)                    ! power.irp.f_shell_44: 173
  character*(8) :: irp_here = 'power_19'      ! power.irp.f_shell_44: 173
 real, intent(in) :: x1                       ! power.irp.f_shell_44: 174
 real :: x2, x4, x8, x9, x18, x19             ! power.irp.f_shell_44: 175
 x2 = x1 * x1                                 ! power.irp.f_shell_44: 176
 x4 = x2 * x2                                 ! power.irp.f_shell_44: 177
 x8 = x4 * x4                                 ! power.irp.f_shell_44: 178
 x9 = x8 * x1                                 ! power.irp.f_shell_44: 179
 x18 = x9 * x9                                ! power.irp.f_shell_44: 180
 x19 = x18 * x1                               ! power.irp.f_shell_44: 181
 power_19 = x19                               ! power.irp.f_shell_44: 182
end                                           ! power.irp.f_shell_44: 183
real function power_20(x1)                    ! power.irp.f_shell_44: 185
  character*(8) :: irp_here = 'power_20'      ! power.irp.f_shell_44: 185
 real, intent(in) :: x1                       ! power.irp.f_shell_44: 186
 real :: x2, x4, x5, x10, x20                 ! power.irp.f_shell_44: 187
 x2 = x1 * x1                                 ! power.irp.f_shell_44: 188
 x4 = x2 * x2                                 ! power.irp.f_shell_44: 189
```

```
 x5 = x4 * x1                                    ! power.irp.f_shell_44: 190
 x10 = x5 * x5                                   ! power.irp.f_shell_44: 191
 x20 = x10 * x10                                 ! power.irp.f_shell_44: 192
 power_20 = x20                                  ! power.irp.f_shell_44: 193
end                                              ! power.irp.f_shell_44: 194
```

IRPF90 for HPC

Using the *--align* option, IRPF90 can introduce compiler directives for the Intel Fortran compiler, such that *all* the arrays are aligned. The *$IRP_ALIGN* variable corresponds to this alignment.

For example,

```
integer function align_double(i)
 integer, intent(in) :: i
 integer :: j
 j = mod(i,max($IRP_ALIGN,4)/4)
 if (j==0) then
   align_double = i
 else
   align_double = i+4-j
 endif
end
```

```
 BEGIN_PROVIDER [ integer, n ]
&BEGIN_PROVIDER [ integer, n_aligned ]
  integer :: align_double
  n = 19
  n_aligned = align_double(19)
END_PROVIDER

BEGIN_PROVIDER [ double precision, Matrix, (n_aligned,n) ]
 Matrix = 0.d0
END_PROVIDER
```

```
program test
  print *,  size(Matrix,1), size(Matrix,2)
end
```

Generated code without alignment:

```
! -*- F90 -*-
!
```

```
!-----------------------------------------------!
! This file was generated with the irpf90 tool. !
!                                               !
!           DO NOT MODIFY IT BY HAND            !
!-----------------------------------------------!

module matrix_mod
  double precision, allocatable :: matrix (:,:)
  logical :: matrix_is_built = .False.
  integer :: n_aligned
  integer :: n
  logical :: n_is_built = .False.
end module matrix_mod
```

```
! -*- F90 -*-
!
!-----------------------------------------------!
! This file was generated with the irpf90 tool. !
```

```
!                                                    !
!           DO NOT MODIFY IT BY HAND                 !
!----------------------------------------------------!

subroutine provide_matrix
  use matrix_mod
  implicit none
  character*(14) :: irp_here = 'provide_matrix'
  integer                    :: irp_err
  logical                    :: irp_dimensions_OK
  if (.not.n_is_built) then
    call provide_n
  endif
 if (allocated (matrix) ) then
  irp_dimensions_OK = .True.
  irp_dimensions_OK = irp_dimensions_OK.AND.(SIZE(matrix,1)==(n_aligned))
  irp_dimensions_OK = irp_dimensions_OK.AND.(SIZE(matrix,2)==(n))
  if (.not.irp_dimensions_OK) then
```

```
    deallocate(matrix,stat=irp_err)
    if (irp_err /= 0) then
```

```
      print *, irp_here//': Deallocation failed: matrix'
      print *, ' size: (n_aligned,n)'
    endif
    if ((n_aligned>0).and.(n>0)) then
     allocate(matrix(n_aligned,n),stat=irp_err)
     if (irp_err /= 0) then
      print *, irp_here//': Allocation failed: matrix'
      print *, ' size: (n_aligned,n)'
     endif
    endif
   endif
  else
    if ((n_aligned>0).and.(n>0)) then
     allocate(matrix(n_aligned,n),stat=irp_err)
     if (irp_err /= 0) then
      print *, irp_here//': Allocation failed: matrix'
      print *, ' size: (n_aligned,n)'
     endif
```

```
   endif
 endif
 if (.not.matrix_is_built) then
  call bld_matrix
  matrix_is_built = .True.

 endif
end subroutine provide_matrix

subroutine bld_matrix
  use matrix_mod
  character*(6) :: irp_here = 'matrix'                    ! matrix.irp.f:  19
 Matrix = 0.d0                                            ! matrix.irp.f:  20
end subroutine bld_matrix
subroutine provide_n
  use matrix_mod
  implicit none
  character*(9) :: irp_here = 'provide_n'
```

```
  integer                        :: irp_err
  logical                        :: irp_dimensions_OK
 if (.not.n_is_built) then
  call bld_n
  n_is_built = .True.
```

```
 endif
end subroutine provide_n

subroutine bld_n
  use matrix_mod
  character*(1) :: irp_here = 'n'                ! matrix.irp.f:  12
  integer :: align_double                        ! matrix.irp.f:  14
  n = 19                                         ! matrix.irp.f:  15
  n_aligned = align_double(19)                   ! matrix.irp.f:  16
end subroutine bld_n
integer function align_double(i)                 ! matrix.irp.f:   1
  character*(12) :: irp_here = 'align_double'    ! matrix.irp.f:   1
 integer, intent(in) :: i                        ! matrix.irp.f:   2
 integer :: j                                    ! matrix.irp.f:   3
 j = mod(i,max(1,4)/4)                           ! matrix.irp.f:   4
 if (j==0) then                                  ! matrix.irp.f:   5
   align_double = i                              ! matrix.irp.f:   6
```

```
 else                                                  ! matrix.irp.f:   7
   align_double = i+4-j                                ! matrix.irp.f:   8
 endif                                                 ! matrix.irp.f:   9
end                                                    ! matrix.irp.f:  10
```

Output:

```
$ ./test
        19          19
```

Generated code with an alignment of 32 bytes:

```
! -*- F90 -*-
!
!-----------------------------------------------!
! This file was generated with the irpf90 tool. !
!                                               !
!           DO NOT MODIFY IT BY HAND            !
!-----------------------------------------------!
```

```
module matrix_mod
  double precision, allocatable :: matrix (:,:)
  !DIR$ ATTRIBUTES ALIGN: 32 :: matrix
  logical :: matrix_is_built = .False.
  integer :: n_aligned
  integer :: n
  logical :: n_is_built = .False.
end module matrix_mod
```

```
! -*- F90 -*-
!
!-----------------------------------------------!
! This file was generated with the irpf90 tool. !
!                                               !
!           DO NOT MODIFY IT BY HAND            !
!-----------------------------------------------!

subroutine provide_matrix
```

```
 use matrix_mod
 implicit none
 character*(14) :: irp_here = 'provide_matrix'
 integer                    :: irp_err
 logical                    :: irp_dimensions_OK
 if (.not.n_is_built) then
   call provide_n
 endif
if (allocated (matrix) ) then
 irp_dimensions_OK = .True.
 irp_dimensions_OK = irp_dimensions_OK.AND.(SIZE(matrix,1)==(n_aligned))
 irp_dimensions_OK = irp_dimensions_OK.AND.(SIZE(matrix,2)==(n))
 if (.not.irp_dimensions_OK) then
  deallocate(matrix,stat=irp_err)
  if (irp_err /= 0) then
    print *, irp_here//': Deallocation failed: matrix'
    print *, ' size: (n_aligned,n)'
  endif
```

```
   if ((n_aligned>0).and.(n>0)) then
    allocate(matrix(n_aligned,n),stat=irp_err)
```

```
    if (irp_err /= 0) then
     print *, irp_here//': Allocation failed: matrix'
     print *, ' size: (n_aligned,n)'
    endif
   endif
  endif
 else
   if ((n_aligned>0).and.(n>0)) then
    allocate(matrix(n_aligned,n),stat=irp_err)
    if (irp_err /= 0) then
     print *, irp_here//': Allocation failed: matrix'
     print *, ' size: (n_aligned,n)'
    endif
   endif
 endif
 if (.not.matrix_is_built) then
  call bld_matrix
  matrix_is_built = .True.
```

```
 endif
end subroutine provide_matrix

subroutine bld_matrix
  use matrix_mod
  character*(6) :: irp_here = 'matrix'                                  ! matrix.irp.f:  19
 Matrix = 0.d0                                                          ! matrix.irp.f:  20
end subroutine bld_matrix
subroutine provide_n
  use matrix_mod
  implicit none
  character*(9) :: irp_here = 'provide_n'
  integer                   :: irp_err
  logical                   :: irp_dimensions_OK
 if (.not.n_is_built) then
  call bld_n
  n_is_built = .True.
```

```
 endif
end subroutine provide_n

subroutine bld_n
```

```
  use matrix_mod
  character*(1) :: irp_here = 'n'                ! matrix.irp.f:  12
  integer :: align_double                        ! matrix.irp.f:  14
  n = 19                                         ! matrix.irp.f:  15
  n_aligned = align_double(19)                   ! matrix.irp.f:  16
end subroutine bld_n
integer function align_double(i)                 ! matrix.irp.f:   1
  character*(12) :: irp_here = 'align_double'    ! matrix.irp.f:   1
 integer, intent(in) :: i                        ! matrix.irp.f:   2
 integer :: j                                    ! matrix.irp.f:   3
 j = mod(i,max(32,4)/4)                          ! matrix.irp.f:   4
 if (j==0) then                                  ! matrix.irp.f:   5
   align_double = i                              ! matrix.irp.f:   6
 else                                            ! matrix.irp.f:   7
   align_double = i+4-j                          ! matrix.irp.f:   8
 endif                                           ! matrix.irp.f:   9
end                                              ! matrix.irp.f:  10
```

Output:

```
$ ./test
        20          19
```

To remove all compiler directives introduced by the programmer, it is possible to use *irpf90 --no-directives*.

More about IRPF90

- ArXiv: http://arxiv.org/abs/0909.5012
- Web site: http://irpf90.ups-tlse.fr

Single core optimization in QMC=Chem

Anthony Scemama <scemama@irsamc.ups-tlse.fr>
Michel Caffarel <michel.caffarel@irsamc.ups-tlse.fr>

Labratoire de Chimie et Physique Quantiques
IRSAMC (Toulouse)

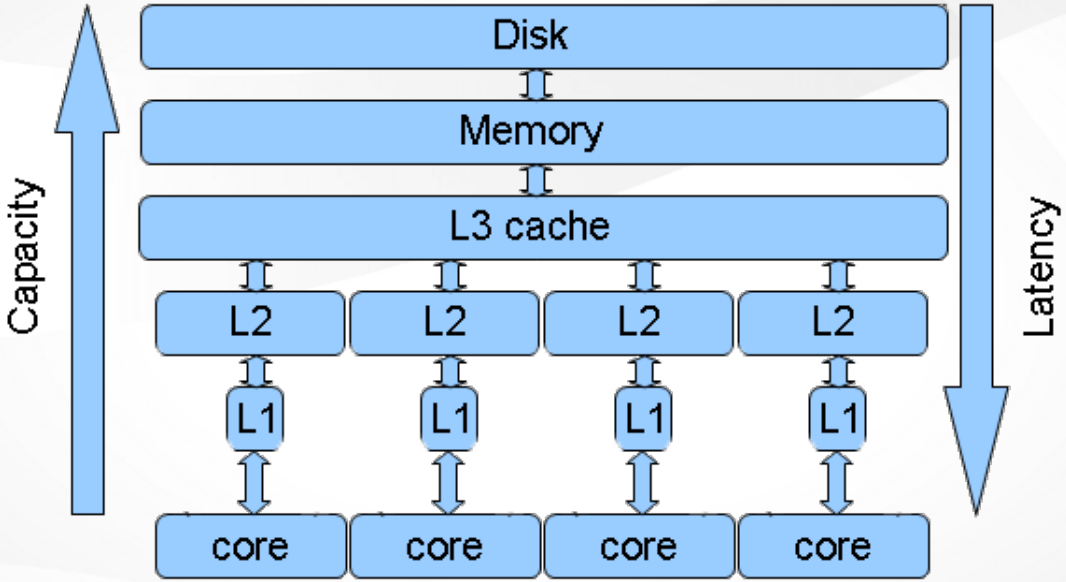
Hardware considerations

Intel(R) Xeon(R) CPU E31220 @ 3.10GHz
3.4GHz turbo, 8 MiB shared L3, 256 KiB L2, 32 KiB L1

- The AVX instruction set allows to perform vector operations on 256 bits : 8 single precision (SP) elements or 4 double precision (DP) elements
- The vector ADD and MUL operations have a throughput of 1 per cycle (pipelining)
- One vector ADD, one vector MUL and one integer ADD (loop count) can be performed simultaneously
- Therefore, the peak performance of an Intel Sandy/Ivy Bridge core is 16 floating point operations (flops) per cycle (SP) or 8 flops/cycle (DP)
- One E31220 core has a peak performance of 54.4 Gflops/s (SP), 27.2 Gflops/s (DP)
- In the peak regime, one flop takes in average 0.018 ns (SP) or 0.037 ns (DP)

On modern architectures, reducing the number of flops does **not** systematically reduce the execution time.

Memory access is much more critical. Understanding how the data arrives to the CPU helps to write efficient code [*].



---

* "**What Every Programmer Should Know About Memory**, U. Drepper, (2007), http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.91.957

Measures obtained with LMbench †

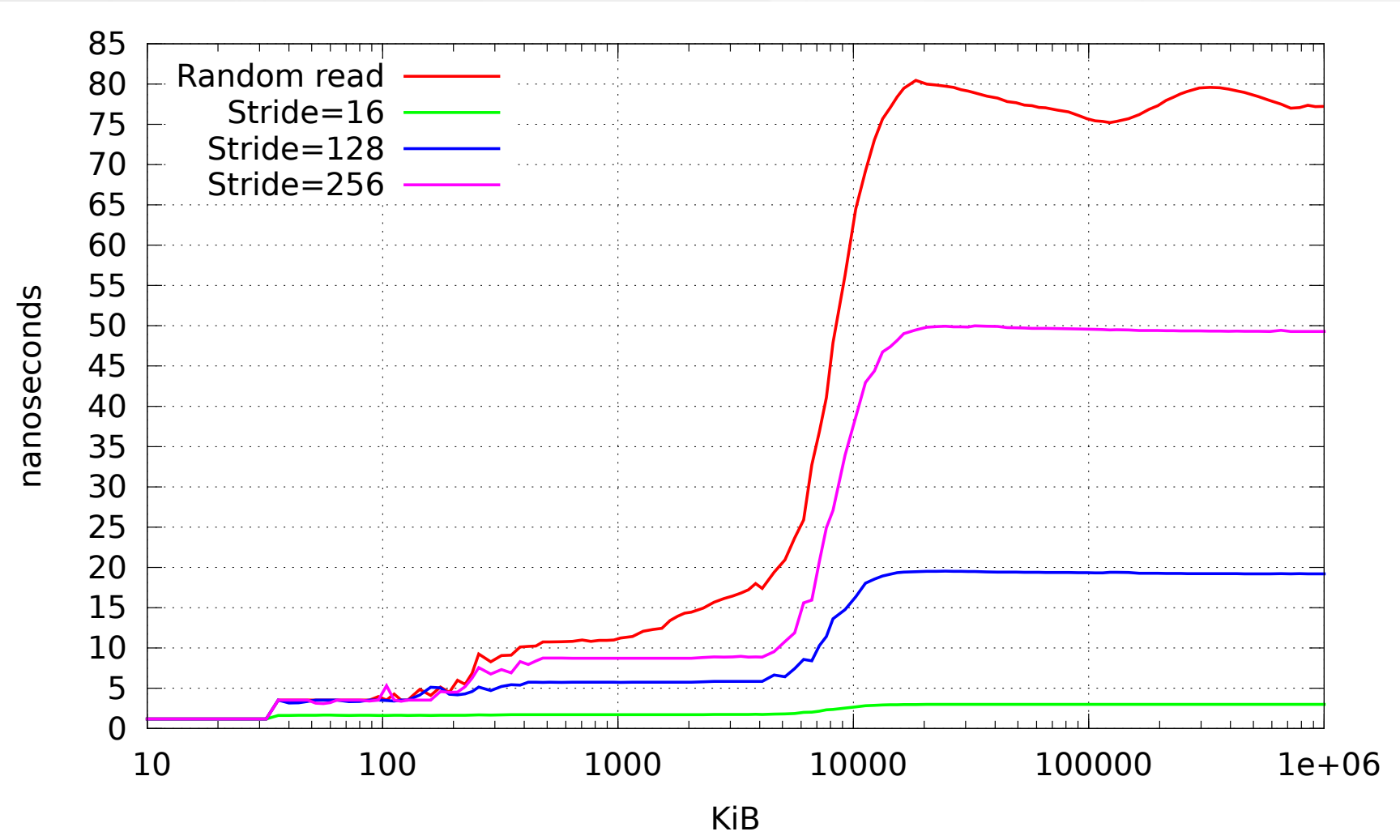
1 cycle = 0.29 ns, 1 peak flop SP = 0.018 ns

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	1.18	1.18
L2	3.5	1.6
L3	13	1.7
Memory	75-80	3.

† http://www.bitmover.com/lmbench/


Random read
Stride=16
Stride=128
Stride=256
nanoseconds
KiB
0
5
10
15
20
25
30
35
40
45
50
55
60
65
70
75
80
85
10
100
1000
10000
100000
1e+06

- Random access to memory is **very** slow : 79 ns = 270 CPU cycles : 4300 flops (Peak SP)
- Strided access to memory with a stride < 4096 KiB (1 page) triggers the hardware prefetchers, reducing the memory latencies. Smaller strides are better, and give latencies comparable to L2 latencies.

Recomputing data may be faster than fetching it randomly in memory

Other important numbers:

Mutex lock/unlock	~100 ns
Infiniband	~1 200 ns
Ethernet	~50 000 ns
Disk seek (SSD)	~50 000 ns
Disk seek (15k rpm)	~2 000 000 ns

Example : squared distance matrix

```
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

```
do j=1,n
 do i=1,j
  dist2(i,j) = X(i,1)*X(j,1) + X(i,2)*X(j,2) + X(i,3)*X(j,3)
  dist2(j,i) = dist2(i,j)
 end do
end do
```

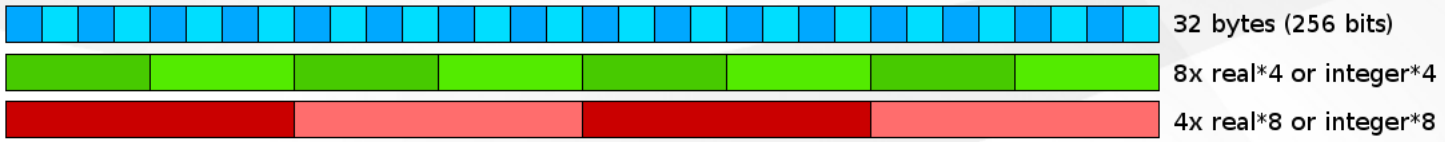
t(n=133) = 11.5 μs : 1.13x speed-up, 3.5 GFlops/s
t(n=4125) = 90.4 ms : 1.05x speed-up, 0.47 GFlops/s

```
do j=1,n
 do i=1,n
  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
```

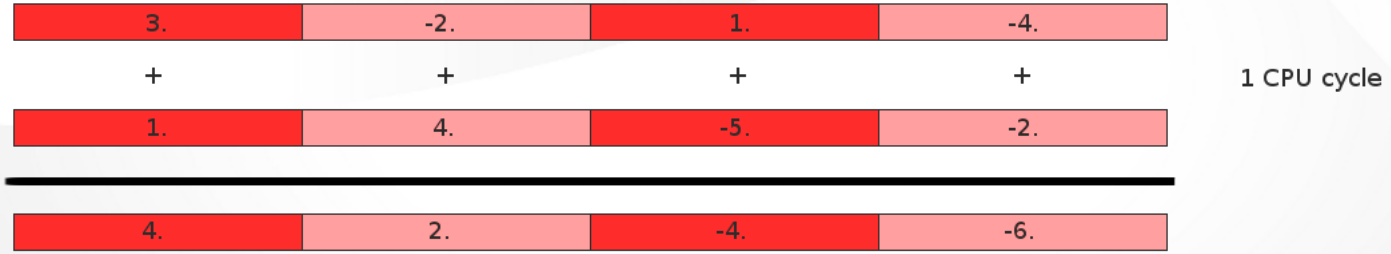
2x more flops!
t(n=133) = 10.3 μs : 1.12x speed up, 8.2 GFlops/s
t(n=4125) = 15.7 ms : 5.75x speed up, 5.4 GFlops/s

Vector operations

AVX single instruction / multiple data (SIMD) instructions operate on 256-bit floating point registers:



Example : vector ADD in double precision:



Requirements:

- The elements of each SIMD vector must be contiguous in memory
- The first element of each SIMD vector must be aligned on a 32 byte boundary

Automatic vectorization

The compiler can generate automatically vector instructions when possible. An auto-vectorized loop generates 3 loops:

Peel loop (scalar)

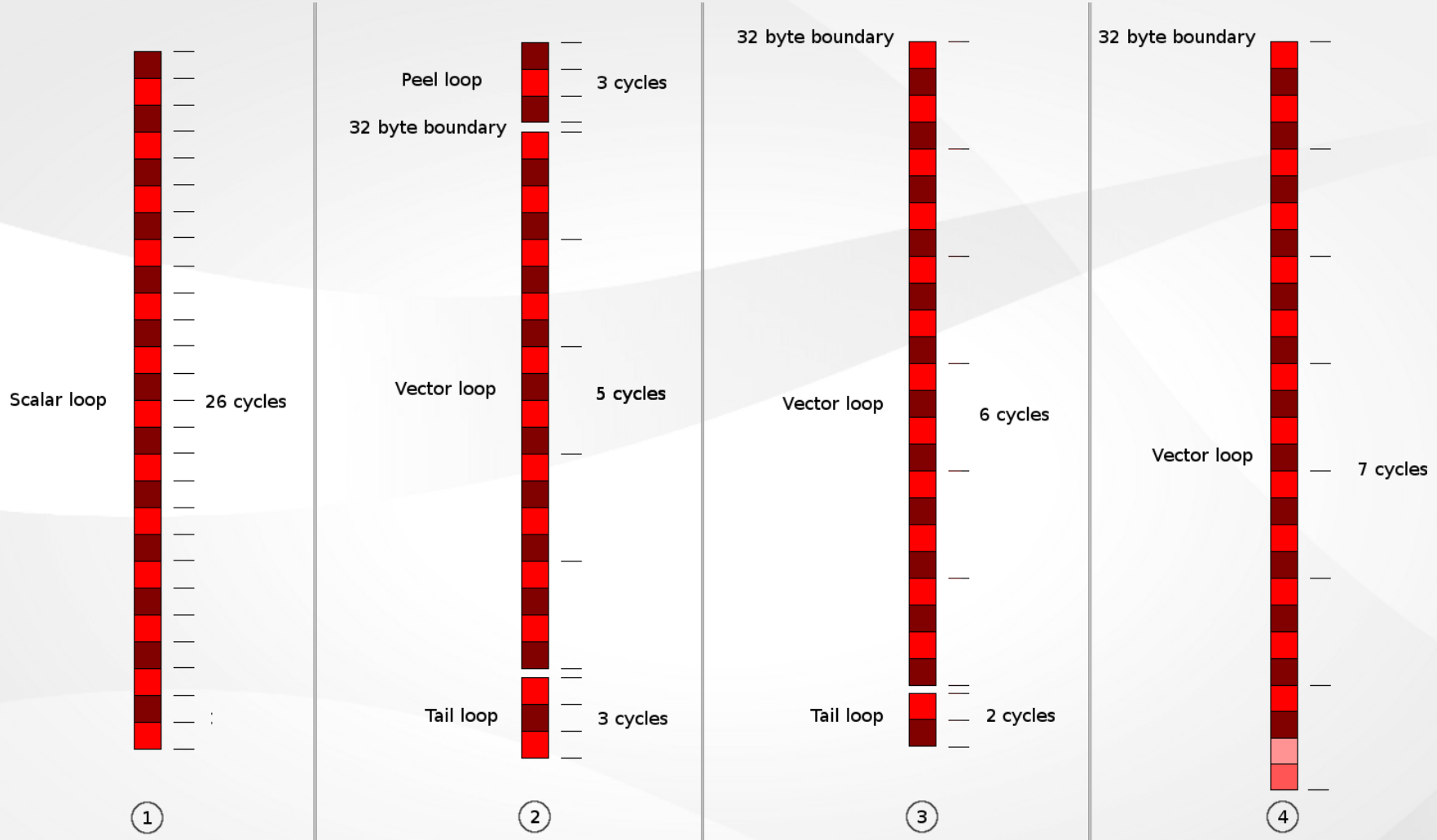
First elements until the 32 byte boundary is met

Vector loop

Vectorized version until the last vector of 4 elements

Tail loop (scalar)

Last elements


Scalar loop
26 cycles
1
Peel loop
3 cycles
32 byte boundary
Vector loop
5 cycles
Tail loop
3 cycles
2
32 byte boundary
Vector loop
6 cycles
Tail loop
2 cycles
3
32 byte boundary
Vector loop
7 cycles
4

Intel specific Compiler directives

To remove the peel loop, you can tell the compiler to align the arrays on a 32 byte boundary using:

```
double precision, allocatable :: A(:), B(:)
!DIR$ ATTRIBUTES ALIGN : 32 :: A, B
```

Then, before using the arrays in a loop, you can tell the compiler that the arrays are aligned. Be careful: if one array is not aligned, this may cause a segmentation fault.

```
!DIR$ VECTOR ALIGNED
do i=1,n
  A(i) = A(i) + B(i)
end do
```

To remove the tail loop, you can allocate A such that its dimension is a multiple of 4 elements:

```
n_4 = mod(n,4)
if (n_4 == 0) then
  n_4 = n
else
  n_4 = n - n_4 + 4
endif
allocate ( A(n_4), B(n_4) )
```

and rewrite the loop as follows:

```
do i=1,n,4
  !DIR$ VECTOR ALIGNED
  !DIR$ VECTOR ALWAYS
  do k=0,3
    A(i+k) = A(i+k) + B(i+k)
  end do
end do
```

In that case, the compiler knows that each inner-most loop cycle can be transformed safely into only vector instructions, and it will not produce the tail and peel loops with the branching. For small arrays, the gain can be significant.

For multi-dimensional arrays, if the 1st dimension is a multiple of 4 elements, all the columns are aligned:

```
double precision, allocatable :: A(:,:)
!DIR$ ATTRIBUTES ALIGN : 32 :: A
allocate( A(n_4,m) )
do j=1,m
 do i=1,n,4
   !DIR$ VECTOR ALIGNED
   !DIR$ VECTOR ALWAYS
   do k=0,3
     A(i+k,j) = A(i+k,j) * B(i+k,j)
   end do
 end do
end do
```

Warning :

In practice, using multiples of 4 elements is not always the best choice. Using multiples of 8 or 16 elements can be better because the inner-most loop may be unrolled by the compiler to improve the efficiency of the pipeline.

Example : squared distance matrix

```
do j=1,n
 do i=1,n,8
  !DIR$ VECTOR ALIGNED
  !DIR$ VECTOR ALWAYS
  do k=0,7
    dist4(i+k,j) = X(i+k,1)*X(j,1) + X(i+k,2)*X(j,2) + X(i+k,3)*X(j,3)
  end do
 end do
end do
```

t(n=133) = 7.2 μs : 1.44x speed-up, 12.1 GFlops/s
t(n=4125) = 15.5 ms : 1.01x speed-up, 7.5 GFlops/s.

Hot spots of QMC algorithms

At every Monte Carlo step, the following quantities have to be computed:

- $\Psi_T \; : \; \Psi_T(\mathbf{r}_1, \ldots, \mathbf{r}_N) = \sum_i c_i \det\left(D_i^\alpha(\mathbf{r}_1, \ldots, \mathbf{r}_{N_\alpha})\right) \det\left(D_i^\beta(\mathbf{r}_{N_\alpha+1}, \ldots, \mathbf{r}_N)\right)$

  Slater matrix:

  $$D_i^\alpha(\mathbf{r}_1, \ldots, \mathbf{r}_{N_\alpha}) = \begin{pmatrix} \phi_1(\mathbf{r}_1) & \ldots & \phi_{N_\alpha}(\mathbf{r}_1) \\ \vdots & \vdots & \vdots \\ \phi_1(\mathbf{r}_{N_\alpha}) & \ldots & \phi_{N_\alpha}(\mathbf{r}_{N_\alpha}) \end{pmatrix}$$

- $\dfrac{\nabla_i \Psi_T}{\Psi_T} \; : \; \dfrac{1}{\Psi_T(\mathbf{r}_1, \ldots, \mathbf{r}_N)} \dfrac{\partial}{\partial \mathbf{r}_i} \Psi_T(\mathbf{r}_1, \ldots, \mathbf{r}_N)$

- $\dfrac{\Delta_i \Psi_T}{\Psi_T} \; : \; \dfrac{1}{\Psi_T(\mathbf{r}_1, \ldots, \mathbf{r}_N)} \Delta_i \Psi_T(\mathbf{r}_1, \ldots, \mathbf{r}_N)$

Calculation of the Slater matrices

The Slater matrices have to be computed, as well as their gradients and Laplacian.

It is necessary to compute the values, gradients and Laplacian of the Molecular Orbitals (MOs) at the electron positions.

$$\phi_i(\mathrm{r}) = \sum_k C_{ik}\chi_k(\mathrm{r})$$

$$\chi_k(\mathrm{r}) = (x - x_A)^{a_k}(y - y_A)^{b_k}(z - z_A)^{c_k}\sum_l d_l e^{-\alpha_{kl}|\mathrm{r}-\mathrm{r}_A|^2}$$

- *C* is the matrix of MO coefficients (constant)
- A_1 : MO values
- B_1 : AO values
- A_2, A_3, A_4 : MO gradients (x,y,z)

- B_2, B_3, B_4 : AO gradients (x,y,z)
- A_5 : of MO Laplacian
- B_5 : of AO Laplacian

We need to compute $A_i = C \times B_i$ efficiently:

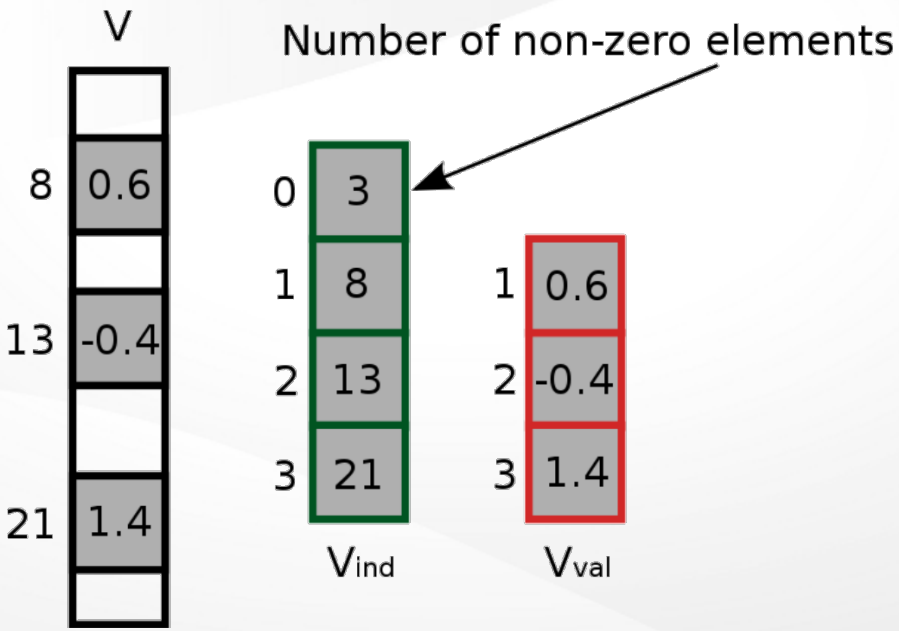
- Single precision is sufficient
- AOs are not orthonormal and centered on nuclei
- All B_i have null elements where $|r-r_i|^2$ is large : only non-zero elements are computed
- All B_i are sparse with non-zero elements at the same indices
- C is constant and dense
- The size of A_i is small (~ $N_{elec} / 2$)
- We have implemented a very efficient dense x sparse matrix product for small matrices

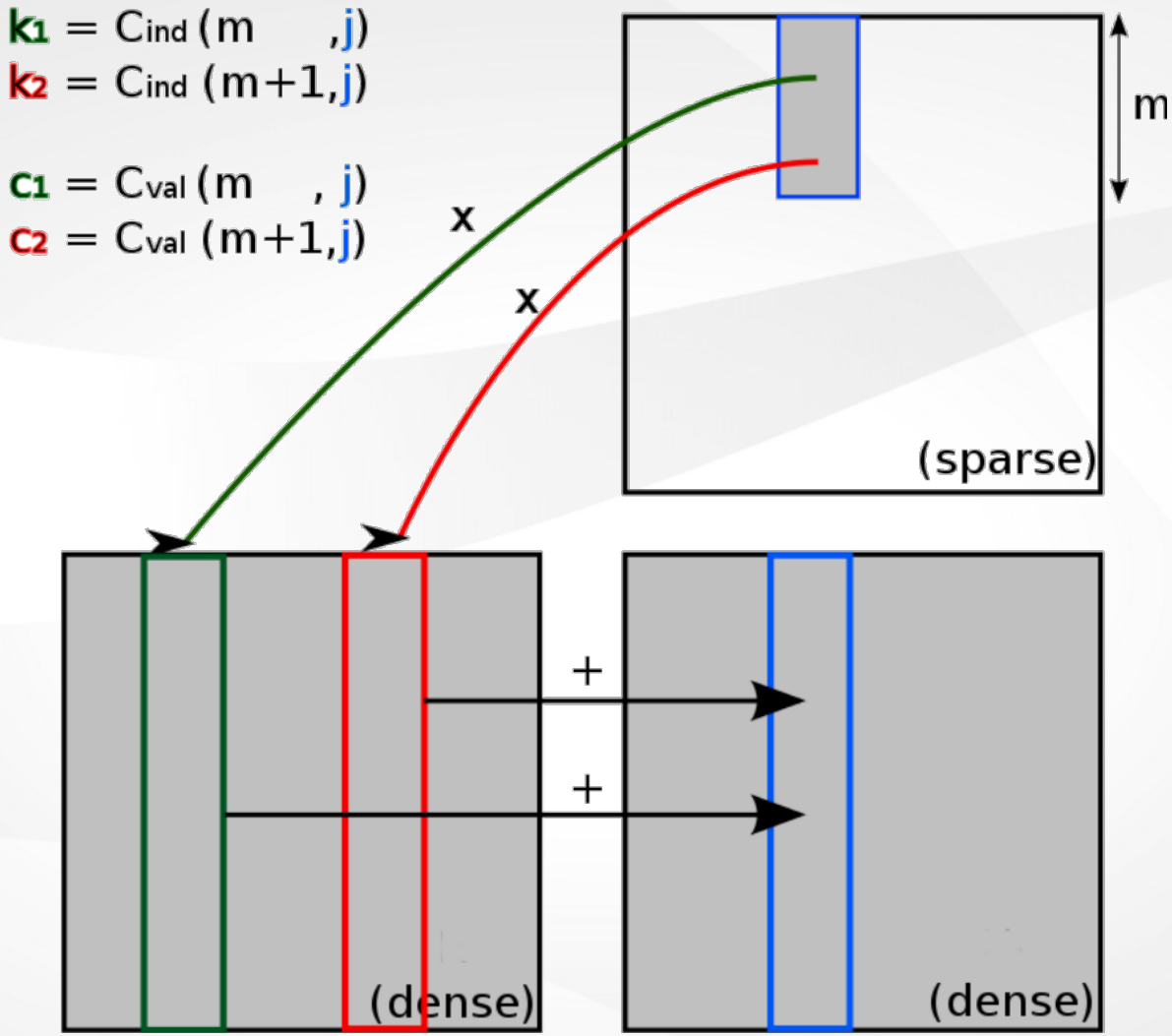
Dense Matrix x Sparse Vector Product

To improve cache locality and reduce memory, we:

- compute one column of all B_i and store them sparse
- make the product of C with these vectors and store all A_i

The sparse vectors are represented as:




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

In QMC=Chem, all arrays are aligned on a 32 byte boundary by IRPF90. The leading dimension is always a multiple of 8 elements.

```
! $IRP_ALIGN = 32
! $IRP_ALIGN/4-1 = 7

! Initialize output vectors
! -------------------------

     !DIR$ VECTOR ALIGNED
     do j=1,LDA,max(1,$IRP_ALIGN/4)
      !DIR$ VECTOR ALIGNED
      A1(j:j+$IRP_ALIGN/4-1) = 0.
      !DIR$ VECTOR ALIGNED
      A2(j:j+$IRP_ALIGN/4-1) = 0.
      !DIR$ VECTOR ALIGNED
      A3(j:j+$IRP_ALIGN/4-1) = 0.
      !DIR$ VECTOR ALIGNED
      A4(j:j+$IRP_ALIGN/4-1) = 0.
```

```
      !DIR$ VECTOR ALIGNED
       A5(j:j+$IRP_ALIGN/4-1) = 0.
      enddo

! Unroll and jam x 4
! -------------------

      kmax2 = indices(0)-mod(indices(0),4)
      do kao=1,kmax2,4

  ! Fetch column indices
  ! --------------------

       k_vec(1) = indices(kao  )
       k_vec(2) = indices(kao+1)
       k_vec(3) = indices(kao+2)
       k_vec(4) = indices(kao+3)
```

```
! Fetch column factors (1,2)
! ---------------------------

    d11 = B1(kao  )
    d21 = B1(kao+1)
    d31 = B1(kao+2)
    d41 = B1(kao+3)

    d12 = B2(kao  )
    d22 = B2(kao+1)
    d32 = B2(kao+2)
    d42 = B2(kao+3)

! A += C x B (1,2)
! ----------------

    !DIR$ VECTOR ALIGNED
    !DIR$ LOOP COUNT (256)
```

```
      do j=1,LDA,max(1,$IRP_ALIGN/4)

        !DIR$ VECTOR ALIGNED
        A1(j:j+$IRP_ALIGN/4-1) = A1(j:j+$IRP_ALIGN/4-1) + &
          C(j:j+$IRP_ALIGN/4-1,k_vec(1))*d11 + &
          C(j:j+$IRP_ALIGN/4-1,k_vec(2))*d21 + &
          C(j:j+$IRP_ALIGN/4-1,k_vec(3))*d31 + &
          C(j:j+$IRP_ALIGN/4-1,k_vec(4))*d41

        !DIR$ VECTOR ALIGNED
        A2(j:j+$IRP_ALIGN/4-1) = A2(j:j+$IRP_ALIGN/4-1) + &
          C(j:j+$IRP_ALIGN/4-1,k_vec(1))*d12 + &
          C(j:j+$IRP_ALIGN/4-1,k_vec(2))*d22 + &
          C(j:j+$IRP_ALIGN/4-1,k_vec(3))*d32 + &
          C(j:j+$IRP_ALIGN/4-1,k_vec(4))*d42

      enddo
```

```
! Fetch column factors (3,4)
! ---------------------------

    d13 = B3(kao  )
    d23 = B3(kao+1)
    d33 = B3(kao+2)
    d43 = B3(kao+3)

    d14 = B4(kao  )
    d24 = B4(kao+1)
    d34 = B4(kao+2)
    d44 = B4(kao+3)

! A = C x B (3,4)
! ---------------

    !DIR$ VECTOR ALIGNED
    do j=1,LDA,max(1,$IRP_ALIGN/4)
```

```
        !DIR$ VECTOR ALIGNED
        A3(j:j+$IRP_ALIGN/4-1) = A3(j:j+$IRP_ALIGN/4-1) + &
          C(j:j+$IRP_ALIGN/4-1,k_vec(1))*d13 + &
          C(j:j+$IRP_ALIGN/4-1,k_vec(2))*d23 + &
          C(j:j+$IRP_ALIGN/4-1,k_vec(3))*d33 + &
          C(j:j+$IRP_ALIGN/4-1,k_vec(4))*d43

        !DIR$ VECTOR ALIGNED
        A4(j:j+$IRP_ALIGN/4-1) = A4(j:j+$IRP_ALIGN/4-1) + &
          C(j:j+$IRP_ALIGN/4-1,k_vec(1))*d14 + &
          C(j:j+$IRP_ALIGN/4-1,k_vec(2))*d24 + &
          C(j:j+$IRP_ALIGN/4-1,k_vec(3))*d34 + &
          C(j:j+$IRP_ALIGN/4-1,k_vec(4))*d44

      enddo

  ! Fetch column factors (5)
```

```
! ---------------------------

    d15 = B5(kao  )
    d25 = B5(kao+1)
    d35 = B5(kao+2)
    d45 = B5(kao+3)


! A += C x B (5), unrolled 2x by compiler
! ---------------------------------------

    !DIR$ VECTOR ALIGNED
    do j=1,LDA,max(1,$IRP_ALIGN/4)  ! Unroll 2 times

      !DIR$ VECTOR ALIGNED
      A5(j:j+$IRP_ALIGN/4-1) = A5(j:j+$IRP_ALIGN/4-1) + &
         C(j:j+$IRP_ALIGN/4-1,k_vec(1))*d15 + &
         C(j:j+$IRP_ALIGN/4-1,k_vec(2))*d25 + &
```

```
            C(j:j+$IRP_ALIGN/4-1,k_vec(3))*d35 + &
            C(j:j+$IRP_ALIGN/4-1,k_vec(4))*d45

       enddo

     enddo

! Tail loop of outer loop
! ------------------------

     do kao = kmax2+1, indices(0)

  ! Fetch column indice
  ! -------------------

       k_vec(1) = indices(kao)
```

```
! Fetch column factors (1-5)
! ---------------------------

      d11 = B1(kao)
      d12 = B2(kao)
      d13 = B3(kao)
      d14 = B4(kao)
      d15 = B5(kao)


! A += B x C (1-5)
! ----------------

      !DIR$ VECTOR ALIGNED
      do j=1,LDA,max(1,$IRP_ALIGN/4)

       !DIR$ VECTOR ALIGNED
       A1(j:j+$IRP_ALIGN/4-1) = A1(j:j+$IRP_ALIGN/4-1) + &
```

```
      C(j:j+$IRP_ALIGN/4-1,k_vec(1))*d11

!DIR$ VECTOR ALIGNED
A2(j:j+$IRP_ALIGN/4-1) = A2(j:j+$IRP_ALIGN/4-1) + &
      C(j:j+$IRP_ALIGN/4-1,k_vec(1))*d12

!DIR$ VECTOR ALIGNED
A3(j:j+$IRP_ALIGN/4-1) = A3(j:j+$IRP_ALIGN/4-1) + &
      C(j:j+$IRP_ALIGN/4-1,k_vec(1))*d13

!DIR$ VECTOR ALIGNED
A4(j:j+$IRP_ALIGN/4-1) = A4(j:j+$IRP_ALIGN/4-1) + &
      C(j:j+$IRP_ALIGN/4-1,k_vec(1))*d14

!DIR$ VECTOR ALIGNED
A5(j:j+$IRP_ALIGN/4-1) = A5(j:j+$IRP_ALIGN/4-1) + &
      C(j:j+$IRP_ALIGN/4-1,k_vec(1))*d15
```

```
      enddo
    enddo
```

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.

Other QMC codes use 3D splines to avoid the computation of AOs, and the matrix products but:

- To do the 3D interpolation, 8 values are needed (corners of a cube)
- This represents 4 random memory accesses : ~320 nanoseconds + the time to compute the interpolation
- In 360 nanoseconds, we can do ~ 12 000 flops.
- The average computation time of 1 element with our matrix product is proportional to the number of non-zero elements in B_i (<500 flops).
- We have shown that our implementation (calculation of A_i and matrix products) is faster than the interpolation by factors of 1.0x to 1.5x
- 3D splines have to be pre-computed on a grid. It takes initialization time
- 3D splines needs many GiB of RAM, so only small systems can be handled, and OpenMP parallelism is often required.

Inverse Slater matrices

To compute $\frac{\nabla_i \Psi_T}{\Psi_T}$ and $\frac{\Delta_i \Psi_T}{\Psi_T}$, one needs the inverse of the Slater matrices:

- $\frac{\nabla_i \Psi_T}{\Psi_T}$ needs $D_k^{-1} \nabla_i D_k$
- $\frac{\Delta_i \Psi_T}{\Psi_T}$ needs $D_k^{-1} \Delta_i D_k$

A unique list of alpha and beta Slater matrices is generated, and the results are combined at the end to produce the alpha x beta determinant products.

To give accurate results, double precision is required. For each spin, the first inverse Slater matrix is fully calculated:

- < 5x5 : hand-written O(N!) algorithm ($5! < 5^3$)
- > 5x5 : MKL library : *dgetrf*, *dgetri*

The next determinants are calculated using the Shermann-Morisson-Woodbury formula : if only one column differs, the new inverse can be computed in $O(N^2)$.

A CAS-SCF wave function with 10 000 determinant products has 100 unique alpha and 100 unique beta determinants. One of those will be computed in $O(N^3)$ and all others will be computed in $O(N^2)$. For a 40 electrons system (20 alpha, 20 beta), computing 10 000 determinants will be only ~6x longer than the single-determinant calculation.

Tutorial: Quantum Monte Carlo with QMC=Chem

In this tutorial, We will study the dissociation energy of N_2 using a Hartree-Fock (HF) trial wave function and a complete active space (CAS-SCF) trial wave function. The HF wave function for dissociated N_2 is computed within restricted open-shell HF with a spin multiplicity of 7. These wave functions were prepared using the GAMESS [1] program. The dissociation energy is evaluated by calculating the energy difference of N_2 at $R = 4\text{Å}$ and at $R = 1.1\text{Å}$, where R is the inter-atomic distance.

In your directory, you should have:

```
$ ls
1.1.CAS/     1.1.HF/                 1.1.HF.1core.sub  4.CAS/     4.HF/
1.1.CAS.out  1.1.HF.160core.sub  1.1.HF.out           4.CAS.out  4.HF.out
```

- The **.out* files are the GAMESS output files
- The **.sub* files are the files needed to submit a job
- The directories are the corresponding QMC=Chem EZFIO database files [2]

Running a VMC calculation

Single core run

To access the input data, run

```
$ qmcchem_edit.py 1.1.HF
```

This command will open a temporary file containing the different parameters of the simulation. Modify them as follows:

```
# Simulation
# --------------------

end_condition  = "wall_time > 300"
jastrow        = False
method         = "VMC"
nucl_fitcusp   = True
num_step       = 10000
sampling       = "Langevin"
time_step      = 0.2
title          = "HF, 1.1 angstroms"
walk_num       = 20
```

end_condition

Stopping condition of the run. 5 minutes is fine.

jastrow

If true, use a Jastrow factor to improve the trial wave function. In this tutorial, we will not use it.

method

VMC: Variational Monte Carlo.

nucl_fitcusp

Impose the correct electron-nucleus cusp at the nucleus to avoid the divergence of the energy at the nuclei. This doesn't change the energy but considerably reduces its variance.

num_step

Number of steps per block. This is usually adjusted such that the time spent to compute one block is not too small or not too large. Typically, for very short runs 20 seconds is OK, and for usual production runs, this parameters is adjusted to 10 minutes per block.

sampling

The Monte Carlo sampling algorithm. Langevin is the best for VMC.

time_step

Simulation time step. Using the Langevin algorithm, 0.2 is usually a good choice.

title

Title of the run. You can put whatever you want.

walk_num

Number of walkers (independent trajectories in VMC).

When you save the fiel and exit the text editor, the EZFIO database has been updated. You can now run the QMC calculation. First, run a single-core run:

```
$ ccc_msub -A <your_curie_account> 1.1.HF.1core.sub
```

In QMC=Chem, there is no output file. At any time, you can see what has been computed by running:

```
$ qmcchem_result.py -s 1.1.HF
```

The output of this command should look like this:

```
#                                  Summary
#---------------------------------------------------------------------
Number of blocks                   : 26
Number of blocks per core          : 26
Total CPU time                     : 0:04:51
CPU time / block                   : 11.192(79)
Acceptance rate                    : 0.92348(10)
#---------------------------------------------------------------------
e_loc                              : -108.9842(52)
Variance of e_loc                  : 25.75(30)
Min of e_loc                       : -351.467898466
Max of e_loc                       : 503.693209743
#---------------------------------------------------------------------
```

Number of blocks

Total number of blocks in the database.

Number of blocks per core

Each CPU core has individually made this number of blocks.

Total CPU time

Sum of the CPU times of all the cores.

CPU time / block

Average CPU time per block.

Acceptance rate

Average Metropolis acceptance rate.

e_loc

Average of the local energy.

Variance of e_loc

Variance (σ^2) of the local energy.

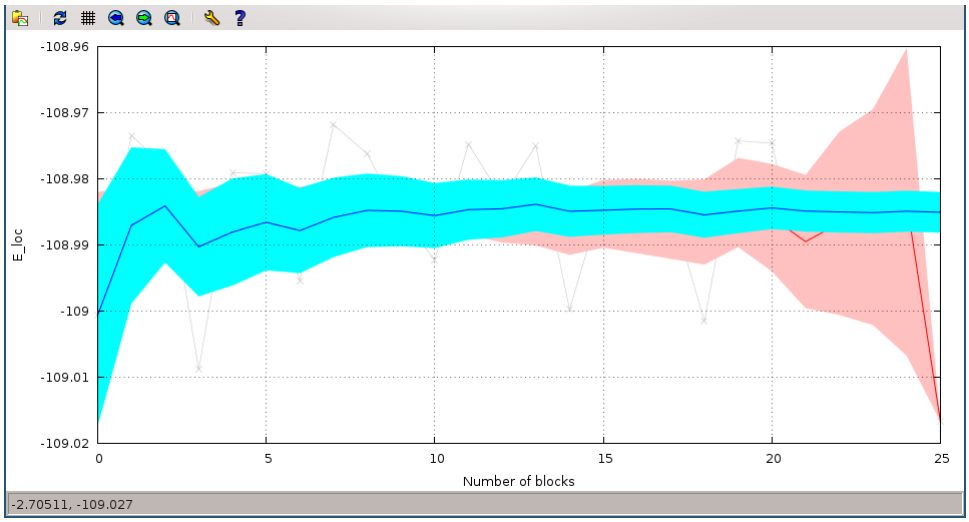
Min/Max of e_loc

Min or Max value of the local energy encountered in the simulation.

At the end of the run, check that the average of the local energy corresponds to the Hartree-Fock energy given by GAMESS (within the error bars).

You can plot the convergence of the local energy using:

```
$ qmcchem_result.py -p E_loc 1.1.HF
```



The blue curve is the convergence plot of the local energy by cumulating blocks from the first block to the last block. The red curve is the same convergence plot but using the blocks from the last one to the first one. If the calculation is converged, the blocks are independent between each other and the shape of the curve should not depend on the order in which the blocks are taken. If the blue and the red convergence plots are not compatible, the QMC run is not converged.

Multi-core run

Your first calculation has finished. If you want, you can add more blocks to the EZFIO database. To do this, run a calculation in parallel using

```
$ ccc_msub -A <your_curie_account> 1.1.HF.160core.sub
```

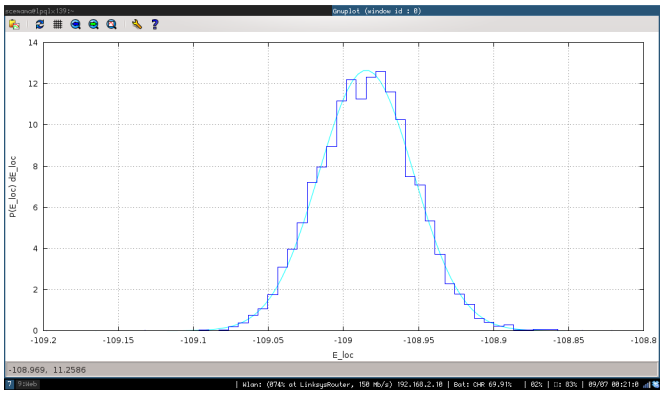
Check that the error bar is significantly reduced, and that the total CPU time is 160x larger:

```
$ qmcchem_result.py -s -c 1.1.HF
#                                  Summary
#-------------------------------------------------------------------------
Number of blocks                   : 3168
Number of blocks per core          : 27
Total CPU time                     : 9:49:16
CPU time / block                   : 11.292(19)
Acceptance rate                    : 0.923571(13)
#-------------------------------------------------------------------------
e_loc                              : -108.98489(56)
Variance of e_loc                  : 26.082(58)
Min of e_loc                       : -397.069319894
```

```
Max of e_loc                          : 4547.98196953
#------------------------------------------------------------------------
```

Now, you have enough blocks to verify that the blocks have a Gaussian distribution:

```
$ qmcchem_result.py -H E_loc 1.1.HF
```



Run VMC calculations for the 3 other trial wave functions and check that the energy corresponds to the energy given by GAMESS.

Running DMC calculations

For each EZFIO directory, modify the simulation parameters as follows:

```
$ qmcchem_edit.py 1.1.HF

method          = "DMC"
sampling        = "Brownian"
time_step       = 0.0001
title           = "1.1 HF DMC"
walk_num        = 40
```

method

Choose DMC to perform a Diffusion Monte Carlo calculation.

sampling

Choose the Brownian motion for DMC. Langevin is not adapted.

time_step

With the Brownian motion, this time step is sufficiently small to obtain a small time step error, and the Metropolis acceptance rate is close to 99.9%.

walk_num

We use a DMC algorithm with a fixed number of walkers with no population control bias. The counter part is that with a small number of walkers, additional fluctuations of the local energy are introduced. It is preferable to increase the number of walkers for the DMC calculation.

This set of parameters is fine for all the runs. As the effective time step is approximately 10 times less than in VMC, the total computational time to obtain an error bar comparable to the error bar obtained in VMC will be 10 times longer.

Run a first short DMC calculation with a small number of cores (typically one node), such that the walkers move from the VMC distribution to the DMC distribution. Clear the computed data by un-commenting *clear(blocks)*, since this calculation it is not well converged:

```
$ qmcchem_edit.py 1.1.HF

# Clear
# --------------------

# clear(all_blocks)
clear(blocks)
# clear(jastrow)
# clear(walkers)
```

Then, run a longer calculation on 10 nodes.

You should obtain these energies:

Nodes	R	<E> (DMC) (a.u)
HF	1.1	-109.4869(63)
	4.0	-109.1498(76)
CAS	1.1	-109.5094(66)
	4.0	-109.1360(61)

Dissociation energies:

W.F.	Delta E (a.u)
HF	0.1883
CAS	0.3246
DMC/HF	0.3371(98)
DMC/CAS	0.3734(90)
Exact	0.3632

Adding a new property

In this section, we will modify the sources of QMC=Chem to compute a new property. The 3D space is partitioned in two subspaces separated by the plane perpendicular to the N-N bond. We will compute the probability to find 1,2,3,4,...,14 electrons in one subspace. The corresponding local operator is implemented as an array P(elec_num). P(m) = 1.d0 where m is the number of electrons in the subspace, and P = 0.d0 eleswhere. The average of this operator will give the probability of finding 1,2,3,4,...,14 electrons in the subspace.

Adding the property to the sources

First, go into the QMC=Chem source directory:

```
$ cd ${QMCCHEM_PATH}/src
```

Create a new file, named *properties_cecam.irp.f* with the following content:

```
!=============================================================================!
! PROPERTIES
!=============================================================================!
```

```
BEGIN_PROVIDER [ double precision, proba_N2, (14) ]
 implicit none
 BEGIN_DOC
! Probability of finding N electrons on one N atom in N2
 END_DOC
 integer :: i, n

 n = 0
 proba_N2 = 0.d0
 do i=1,elec_num
   if (elec_coord(i,3) > 0.d0) then
      n += 1
   endif
 enddo
 if (n>0) then
   proba_N2(n) = 0.5d0
   proba_N2(elec_num-n) = 0.5d0
 endif


END_PROVIDER
```

Do not remove the 3 first commented lines: they are used by an embedded shell script to detect that what follows are properties to compute.

Then, build the program:

```
$ cd ${QMCCHEM_PATH}
$ make
```

Before runnning tests, we will have to restore the VMC parameters in our EZFIO databases.

Restoring the VMC configuration

QMC=Chem keeps track of all the modifications if the EZIO database:

```
$ qmcchem_log.py 1.1.HF
       |         Date         |               MD5                |
 ------------------------------------------------------------------------------
    1 |  2013-07-08 14:05:39 | 97395378eaa00b194de0536dbd172153 | Edit
    2 |  2013-07-08 14:05:42 | 97395378eaa00b194de0536dbd172153 | Generate new walkers
    3 |  2013-07-08 14:05:43 | 97395378eaa00b194de0536dbd172153 | Start run
    4 |  2013-07-08 14:06:08 | 97395378eaa00b194de0536dbd172153 | Stop run
    5 |  2013-07-08 14:06:28 | f622a3fc6e35fc3a75717d43e1b84de2 | Edit
    6 |  2013-07-08 14:06:36 | 9e8b5122372c7f6e7698a8a55861131b | Edit
    7 |  2013-07-08 14:06:43 | 9e8b5122372c7f6e7698a8a55861131b | Clear all_blocks
    8 |  2013-07-08 15:41:20 | 295d77618e1507bbd3152d6e33610ddb | Start run
    9 |  2013-07-08 15:43:28 | 295d77618e1507bbd3152d6e33610ddb | Stop run
   10 |  2013-07-09 11:38:40 | 93b358fb4ead5aa061b40c2807ea0e73 | Edit
   11 |  2013-07-09 11:38:59 | 93b358fb4ead5aa061b40c2807ea0e73 | Start run
   12 |  2013-07-09 11:44:07 | 93b358fb4ead5aa061b40c2807ea0e73 | Stop run
```

From this data, you can identify that the DMC run should be at step number 10, as the MD5 key has changed. To verify this, run:

```
$ qmcchem_log.py log 10 1.1.HF
Date        :  2013-07-09 11:38:40
```

```
MD5         :  93b358fb4ead5aa061b40c2807ea0e73
Description :  Edit

Wave function
=============

N_atoms         = 2
N_electrons     = 14 (7 alpha, 7 beta)
N_det           = 1
N_MOs           = 60
N_AOs           = 70
no Jastrow
nuclear cusp fitting

DMC
====

time_step       = 0.0001
sampling        = Brownian
N_steps         = 10000
N_walkers       = 40

Modified
========

simulation/http_server
simulation/time_step
simulation/sampling
electrons/elec_walk_num
simulation/method
simulation/title
```

You see that it is a DMC run, and that *simulation/method* has been modified from the previous step. This confirms it is the first DMC calculation. Now, you can check out the configuration of the VMC run just before this DMC run:

```
$ qmcchem_log.py checkout 9 1.1.HF

 Date        :  2013-07-09 23:22:29
 MD5         :  295d77618e1507bbd3152d6e33610ddb
 Description :  Checked out 9

 Wave function
 =============

 N_atoms         = 2
 N_electrons     = 14 (7 alpha, 7 beta)
 N_det           = 1
 N_MOs           = 60
 N_AOs           = 70
 no Jastrow
 nuclear cusp fitting

 VMC
 ====
```

```
time_step       = 0.2
sampling        = Langevin
N_steps         = 10000
N_walkers       = 20

Modified
========

electrons/elec_coord.gz
simulation/http_server
simulation/time_step
simulation/print_level
simulation/sampling
electrons/elec_walk_num
simulation/method
simulation/title
```

You can verify that this corresponds to the VMC configuration.

Running the code with the new property to sample

Now, when you run *qmcchem_edit.py*, a new item appears:

```
# Properties
# ----------

...
( ) e_ref_weight
( ) proba_n2
( ) voronoi_charges
...
```

Activate the *proba_n2* property by putting an *X* between the brackets:

```
(X) proba_n2
```

Then, submit VMC calculations for both the HF and the CAS-SCF trial wave functions at $R = 1.1 \text{Å}$ The results can be checked using:

```
$ qmcchem_result.py -t proba_n2 1.1.HF

#                                 proba_n2
#-------------------------------------------------------------------------
#                     Idx                           Average
    1 0.000000
    2 0.000000
    3 0.000156(15)
    4 0.01629(26)
    5 0.09477(52)
    6 0.23309(38)
    7 0.15568(52)
    8 0.23309(38)
    9 0.09477(52)
```

```
10 0.01629(26)
11 0.000156(15)
12 0.000000
13 0.000000
14 0.000000
```

Now, check out the corresponding DMC calculations and sample the histograms. The DMC sampled quantities correspond to the mixed distribution $\Psi_t \Phi_{FN}$. A first-order approximation to the properties computed with Φ_{FN}^2 can by obtained by $\langle O \rangle_{\Phi_{FN}^2} = 2\langle O \rangle_{\Psi_t \Phi_{FN}} - \langle O \rangle_{\Psi_t^2}$

Here are the expected probabilities P(n):

n	HF	DMC(HF)	2DMC-VMC(HF)	CAS-SCF	DMC(CAS-SCF)	2DMC-VMC(CAS-SCF)
4	0.016	0.010	0.004	0.004	0.004	0.004
5	0.095	0.077	0.059	0.054	0.051	0.048
6	0.233	0.240	0.247	0.244	0.244	0.244
7	0.156	0.173	0.190	0.198	0.201	0.204
8	0.233	0.240	0.247	0.244	0.244	0.244
9	0.095	0.077	0.059	0.054	0.051	0.048
10	0.016	0.010	0.010	0.004	0.004	0.004

Going from the HF wave function to the CAS-SCF wave function tends to increase the weight of the neutral components (the probabilities of finding 7 electrons), which is expected. One can also remark that even with HF nodes, this is realized by the DMC algorithm. Using CAS-SCF nodes, the trial wave function has much better probabilities, and the DMC has less work to do. This shows that the the CAS-SCF nodes are much more physical than the HF nodes, and illustrates the difference observed in total energies when going from HF nodes to CAS-SCF nodes.

---

1 http://www.msg.ameslab.gov/gamess/

2 http://ezfio.sourceforge.net . EZFIO is the Easy Fortran I/O library generator written with IRPF90. The data is organized using the filesystem tree in plain text (eventually gzipped) files.

Quantum Monte Carlo Methods in Chemistry

Synonyms and Acronyms

Fixed-node diffusion Monte Carlo (FN-DMC); Green's function Monte Carlo (GFMC); Pure diffusion Monte Carlo (PDMC); Reptation Monte Carlo (RMC); Stochastic reconfiguration Monte Carlo (SRMC); Variational Monte Carlo (VMC)

Description of the Problem

The problem considered here is to obtain accurate solutions of the time-independent Schrödinger equation for a general molecular system described as N electrons moving within the external potential of a set of fixed nuclei. This problem can be considered as the central problem of theoretical and computational chemistry. Using the atomic units adapted to the molecular scale the Schrödinger equation to solve can be written as

$$H\Psi(\mathbf{r}_1,\ldots,\mathbf{r}_N)=E\Psi(\mathbf{r}_1,\ldots,\mathbf{r}_N) \tag{1}$$

where H is the Hamiltonian operator given by

$$H=-\frac{1}{2}\sum_{i=1}^{N}\nabla_i^2+V(\mathbf{r}_1,,\ldots,\mathbf{r}_N), \tag{2}$$

$\{\mathbf{r}_1,\ldots,\mathbf{r}_N\}$ the spatial positions of the N electrons, $\boldsymbol{\nabla}_i^2=\frac{\partial^2}{\partial x_i^2}+\frac{\partial^2}{\partial y_i^2}+\frac{\partial^2}{\partial z_i^2}$ the Laplacian operator for electron i of coordinates $\mathbf{r}_i=(x_i,y_i,z_i)$, Ψ the wavefunction, E the total energy (a real constant), and V the potential energy function expressed as

$$V(\mathbf{r}_1,,\ldots,\mathbf{r}_N)=\sum_{i<j}\frac{1}{r_{ij}}-\sum_{i,\alpha}\frac{Z_\alpha}{r_{i\alpha}}+\sum_{\alpha<\beta}\frac{Z_\alpha Z_\beta}{R_{\alpha\beta}} \tag{3}$$

In this formula $r_{ij}=|\mathbf{r}_i-\mathbf{r}_j|$ is the interelectronic distance, Z_α the charge of nucleus α (a positive integer), $\mathbf{R}_\alpha$ its vector position, $r_{i\alpha}=|\mathbf{r}_i-\mathbf{R}_\alpha|$, and $R_{\alpha\beta}=|\mathbf{R}_\alpha-\mathbf{R}_\beta|$. The Schrödinger equation being invariant under complex conjugation, we can restrict without loss of generality the eigensolutions to be real-valued. The boundary conditions are of Dirichlet-type: Eigenfunctions Ψ are imposed to vanish whenever one electron (or more) goes to infinity

$$\Psi\to 0 \quad \text{as} \quad \sqrt{\mathbf{r}_1^2+\ldots+\mathbf{r}_N^2}\to+\infty \tag{4}$$

In addition, the mathematical constraints resulting from the Pauli principle must be considered. Within a space-only formalism as employed in QMC, two types of electron – usually referred to as the "spin-up" and "spin-down" electrons – are distinguished and the Pauli principle is expressed as follows. Among all eigenfunctions verifying (1)–(4) only those that are antisymmetric under the exchange of any pair of spin-like electrons are physically allowed. Because of the permutational invariance, the $N_\uparrow$ spin-up electrons can be arbitrarily chosen as those having the first labels and the mathematical conditions can be written as

$$\begin{aligned}&\Psi(\ldots,\mathbf{r}_i,\ldots,\mathbf{r}_j,\ldots|\mathbf{r}_{N_\uparrow+1},\ldots,\mathbf{r}_N)\\&=-\Psi(\ldots,\mathbf{r}_j,\ldots,\mathbf{r}_i,\ldots|\mathbf{r}_{N_\uparrow+1},\ldots,\mathbf{r}_N)\end{aligned} \tag{5a}$$

and

$$\begin{aligned} &\Psi(\mathbf{r}_1,\ldots,\mathbf{r}_{N_\uparrow}|\ldots,\mathbf{r}_i,\ldots,\mathbf{r}_j,\ldots) \\ &= -\Psi(\mathbf{r}_1,\ldots,\mathbf{r}_{N_\uparrow}|\ldots,\mathbf{r}_j,\ldots,\mathbf{r}_i,\ldots) \end{aligned} \tag{5b}$$

for all pairs (i, j) of spin-like electrons. Equations 1–5b define the mathematical problem discussed here. Although such a mathematical model results from a number of physical approximations, it contains the bulk of most chemical phenomena and solving it with enough accuracy (=chemical accuracy) can be considered as the major problem of computational chemistry. The two standard approaches to deal with the electronic structure problem in chemistry are the density functional theory (DFT) (Density Functional Theory) and the post-Hartree–Fock wavefunction approaches (Post-Hartree Fock Methods and Excited States Modelling, Coupled-Cluster Methods). Quantum Monte Carlo (QMC) presented here may be viewed as an alternative approach aiming at circumventing the limitations of these two well-established methods (for a detailed presentation of QMC, see, e.g., [1]). In contrast with these deterministic approaches, QMC is based on a stochastic sampling of the electronic configuration space. In the recent years, a number of remarkable applications have been presented, thus establishing QMC as a high potential approach although a number of limitations are still present. Here, we shall present the two most popular approaches used in chemistry, namely, the variational Monte Carlo (VMC) and the fixed-node diffusion Monte Carlo (FN-DMC) methods.

The Variational Monte Carlo (VMC) Method

The variational Monte Carlo (VMC) method is the simpler and the most popular quantum Monte Carlo approach. From a mathematical point of view, VMC is a standard Markov chain Monte Carlo (MCMC) method. Introducing an approximate trial wavefunction $\Psi_T(\mathbf{r}_1,\ldots,\mathbf{r}_N)$ known in an analytic form (a good approximation of the unknown wavefunction), the Metropolis-Hastings algorithm is used to generate sample points distributed in the 3N-dimensional configuration space according to the quantum-mechanical probability density π associated with Ψ_T

$$\pi(\mathbf{R}) = \frac{\Psi_T^2(\mathbf{R})}{\int d\mathbf{R}\Psi_T^2(\mathbf{R})} \tag{6}$$

where $\mathbf{R}$ is a compact notation representing the positions of the N electrons, $\mathbf{R}=(\mathbf{r}_1,\ldots,\mathbf{r}_N)$. Expectation values corresponding to various physical properties can be rewritten as averages over π. As an important example, the total energy defined as

$$E_{VMC}(\Psi_T) \equiv \frac{\int d\mathbf{R}\Psi_T(\mathbf{R})H\Psi_T(\mathbf{R})}{\int d\mathbf{R}\Psi_T^2(\mathbf{R})} \tag{7}$$

may be rewritten under the form

$$E_{VMC}(\Psi_T) = \int d\mathbf{R}\pi(\mathbf{R})E_L(\mathbf{R}) \tag{8}$$

where $E_L(\mathbf{R})$ is the local energy defined as

$$E_L(\mathbf{R}) = \frac{H\Psi_T(\mathbf{R})}{\Psi_T(\mathbf{R})}. \tag{9}$$

In VMC, the total energy is thus estimated as a simple average of the local energy over a sufficiently large number K of configurations $\mathbf{R}^{(k)}$ generated with the Monte Carlo procedure

$$E_{VMC} \simeq \frac{1}{K}\sum_{k=1}^{K} E_L[\mathbf{R}^{(k)}], \tag{10}$$

the estimator becoming exact as K goes to infinity with a statistical error decreasing as $\sim \frac{1}{\sqrt{K}}$. Properties other than the energy can be obtained in a similar way.

In the case of the energy, it can be shown that there exists a variational principle expressed as $E_{VMC}(\Psi_T) \geq E_0$ for any Ψ_T, the equality being obtained for the exact ground-state wavefunction of energy E_0. In addition, there also exists a zero-variance principle stating that the closer the trial wavefunction is from the exact solution, the smaller the fluctuations of the local energy are, the statistical error vanishing in the limit of an exact trial wavefunction. In practice, both principles – minimization of the energy and/or of the fluctuations of the local energy – are at the basis of the various approaches proposed for optimizing the parameters entering the trial wavefunction.

The Diffusion Monte Carlo (DMC) Method

The fundamental idea is to introduce a formal mathematical connection between the quantum and stochastic worlds by introducing a fictitious time dynamics as follows

$$\frac{\partial \Psi(\mathbf{R},t)}{\partial t} = -[H(\mathbf{R}) - E_T]\Psi(\mathbf{R},t) \tag{11}$$

where t plays the role of a time variable, $\Psi(\mathbf{R},t)$, a time-dependent real wavefunction, and E_T, some constant reference energy. The solution of this equation is uniquely defined by the choice of the initial wavefunction, $\Psi(\mathbf{R},t=0)$. Using the spectral decomposition of the self-adjoint (hermitic) Hamiltonian operator, the solution of (11) can be written as

$$\Psi(\mathbf{R},t) = \sum_i c_i e^{-t(E_i - E_T)}\psi_i(\mathbf{R}) \tag{12}$$

where the sum is performed over the complete set of the eigensolutions of the Hamiltonian operator

$$H(\mathbf{R})\psi_i(\mathbf{R}) = E_i\psi_i(\mathbf{R}), \tag{13}$$

and $c_i = \int d\mathbf{R}\psi_i^*(\mathbf{R})\Psi(\mathbf{R},0)$.

As seen from (12) the knowledge of the time-dependent solution of the Schrödinger equation allows to have direct access to information about the time-independent eigensolutions, $\psi_i(\mathbf{R})$. As an important example, the exact ground-state wavefunction (corresponding to the smaller eigenvalue E_0) can be obtained by considering the large-time limit of the time-dependent wavefunction

$$\lim_{t\to+\infty} \Psi(\mathbf{R},t) = \psi_0(\mathbf{R}) \tag{14}$$

up to an unessential multiplicative factor.

In practice, to have an efficient Monte Carlo simulation of the original time-dependent equation, we need to employ some sort of importance sampling, that is, a practical scheme for sampling only the regions of the very high-dimensional configuration space where the quantities to be averaged have a non-vanishing contribution. Here, it is realized by

introducing a trial wavefunction Ψ_T (usually optimized in a preliminary VMC step) and by defining a new time-dependent density as follows

$$\pi(\mathbf{R},t) \equiv \Psi_T(\mathbf{R})\Psi(\mathbf{R},t). \tag{15}$$

The equation that π obeys can be derived without difficulty from (11) and (15), we get

$$\frac{\partial\pi(\mathbf{R},t)}{\partial t} = L\pi(\mathbf{R},t) - [E_L(\mathbf{R}) - E_T]\pi(\mathbf{R},t), \tag{16}$$

where L is a forward Fokker-Planck operator defined as (see, e.g., [2])

$$L\pi = \frac{1}{2}\boldsymbol{\nabla}^2\pi - \boldsymbol{\nabla}[\mathbf{b}(\mathbf{R})\pi] \tag{17}$$

and $\mathbf{b}(\mathbf{R})$ the drift vector given by

$$\mathbf{b}(\mathbf{R}) = \frac{\boldsymbol{\nabla}\Psi_T(\mathbf{R})}{\Psi_T(\mathbf{R})}. \tag{18}$$

In order to define a step-by-step Monte Carlo algorithm, the fundamental equation (16) is rewritten under the following equivalent integral form describing the evolution of the density during a time interval τ

$$\pi(\mathbf{R}, t+\tau) = \int d\mathbf{R}' K(\mathbf{R},\mathbf{R}',\tau)\pi(\mathbf{R}',t) \tag{19}$$

where K is the following integral kernel (or imaginary-time propagator)

$$K(\mathbf{R},\mathbf{R}',\tau) = (\mathbf{R}, e^{\tau L - \tau(E_L - E_T)}\mathbf{R}'). \tag{20}$$

For an arbitrary value of τ, the kernel is not known. However, for small enough time-step accurate approximations of K can be obtained and sampled. To see this, let us first split the exponential operator into a product of exponentials by using the Baker-Campbell-Hausdorff formulas [3]

$$e^{\tau L - \tau(E_L - E_T)} = e^{-\frac{\tau}{2}(E_L - E_T)} e^{\tau L} e^{-\frac{\tau}{2}(E_L - E_T)} + O(\tau^3) \tag{21}$$

and then introduce a short-time gaussian approximation of the Fokker-Planck kernel [2],

$$(\mathbf{R}, e^{\tau L}\mathbf{R}') \simeq \left(\frac{1}{\sqrt{2\pi\tau}}\right)^{3N} e^{-\frac{(\mathbf{R}' - \mathbf{R} - \tau\mathbf{b}(\mathbf{R}))^2}{2\tau}} \tag{22}$$

Finally, a working short-time approximation of the DMC kernel can be written as

$$K_{DMC}(\mathbf{R},\mathbf{R}',\tau) \simeq \left(\frac{1}{\sqrt{2\pi\tau}}\right)^{3N} e^{-\frac{(\mathbf{R}' - \mathbf{R} - \tau\mathbf{b}(\mathbf{R}))^2}{2\tau}} e^{-\frac{\tau}{2}[(E_L(\mathbf{R}') - E_T) + (E_L(\mathbf{R}) - E_T)]} \tag{23}$$

By considering small enough τ, the residual error (called the short-time error in the context of QMC) can be made arbitrarily small. In practice, the DMC simulation is performed as follows. A population of walkers [or configuration $\mathbf{R}^{(k)}$] propagated stochastically from generation to generation according to the DMC kernel is introduced. At each

step, the walkers are moved according to the gaussian transition probability, (22). Next, each walker is killed, kept unchanged, or duplicated a certain number of times proportionally to the remaining part of the K_{DMC} kernel, namely, $w=e^{-\frac{\tau}{2}[(E_L(\mathbf{R}')-E_T)+(E_L(\mathbf{R})-E_T)]}$. In practice, an unbiased integer estimator M defining the number of copies (M = 0, 1, …) is used, M = E[w + u], where E is the integer part and u is a uniform random number in (0, 1) (unbiased $\Rightarrow \int_0^1 duM = w$). In contrast with the Fokker-Planck part, this branching (or birth-death) process causes fluctuations in the number of walkers. Because of that, some sort of population control step is needed [1]. The stationary distribution resulting from these stochastic rules can be obtained as the time-independent solution of (16). After some simple algebra we get

$$\pi(\mathbf{R}) = \frac{\Psi_T(\mathbf{R})\Psi_0(\mathbf{R})}{\int d\mathbf{R}\Psi_T(\mathbf{R})\Psi_0(\mathbf{R})} \tag{24}$$

provided the reference energy E_T is adjusted to the exact value, $E_T = E_0$. From this mixed DMC distribution density, a simple and unbiased estimator of the total energy is obtained

$$E_0=\langle E_L(\mathbf{R})\rangle_\pi. \tag{25}$$

For properties other than the energy, the exact distribution density, Ψ_0^2, must be sampled. This can be realized in different ways, for example, by using a forward walking scheme Ref.[4] or a reptation Monte Carlo algorithm, Ref.[5].

The Fixed-Node Approximation

In the preceding section, the DMC approach has been presented without taking care of the specific mathematical constraints resulting from the Pauli principle, (5b). As it is, this algorithm can be directly employed for quantum systems not subject to such constraints (bosonic systems, quantum oscillators, ensemble of distinguishable particles, etc.). An important remark is that the algorithm converges to the stationary density, (24), associated with the lowest eigenfunction $\psi_0(\mathbf{R})$ which, in the case of a Hamiltonian of the form $H=-\frac{1}{2}\boldsymbol{\nabla}^2+V$, is known to have a constant sign (say, positive). This property is the generalization to continuous operators of the Perron-Frobenius theorem valid for matrices with off-diagonal elements of the same sign.

For electronic systems, the additional fermionic constraints are to be taken into account and we must now force the DMC algorithm to converge to the lowest eigenfunction obeying the Pauli principle (the “physical” or fermionic ground-state) and not to the “mathematical” (or bosonic) ground-state having a constant sign. Unfortunately, up to now it has not been possible to define a computationally tractable (polynomial) algorithm implementing exactly such a property for a general fermionic system (known as the “sign problem”). However, at the price of introducing a fixed-node approximation, a stable method can be defined. This approach called fixed-node DMC (FN-DMC) just consists in choosing a trial wavefunction fulfilling the fermionic constraints, (5b). In contrast with the bosonic-type simulations where the trial wavefunction does not vanish at finite distances, the walkers are now no longer free to move within the entire configurational space. This property results directly from the fact that the nodes of the trial wavefunction [defined as the (3N − 1)-dimensional hypersurface where $\Psi_T(\mathbf{R})=0$] act as infinitely repulsive barriers for the walkers [divergence of the drift vector, (18)]. Each walker is thus trapped forever within the nodal pocket cut by the nodes of Ψ_T where it starts from and the Schrödinger equation is now solved with the additional fixed-node boundary conditions defined as

$$\psi(\mathbf{R})=0 \quad \text{whenever} \quad \Psi_T(\mathbf{R})=0. \tag{26}$$

When the nodes of Ψ_T coincide with the exact nodes, the algorithm is exact. If not, a fixed-node error is introduced. Hopefully, all the nodal pockets do not need to be sampled – which would be an unrealistic task for large systems – due

to the existence of a “tilling” theorem stating that all the nodal pockets of the fermionic ground-state are essentially equivalent and related by permutational invariance [6]. For a mathematical presentation of the fixed-node approximation, see Ref.[7]. Finally, remark that in principle defining an exact fermionic DMC scheme avoiding the fixed-node approximation is not difficult. For example, by letting the walkers go through the nodes and by keeping track of the various changes of signs of the trial wavefunction. However, in practice all the schemes proposed up to now are faced with the existence of an exponentially vanishing signal-to-noise problem related to the uncontrolled fluctuations of the trial wavefunction sign. For details, the reader is referred to the work by Ceperley and Alder [8].

The Trial Wavefunction

A standard form for the trial wavefunction is

$$\Psi_T(\mathbf{R}) = e^{J(\mathbf{R})} \sum_k c_k Det_k^{\uparrow}(\mathbf{r}_1, \ldots, \mathbf{r}_{N_\uparrow}) \, Det_k^{\downarrow}(\mathbf{r}_{N_\uparrow+1}, \ldots, \mathbf{r}_N). \tag{27}$$

where the term $e^{J(\mathbf{R})}$ is usually referred to as the Jastrow factor describing explicitly the electron-electron interactions at different level of approximations. A quite general form employed for $J(\mathbf{R})$ is

$$J(\mathbf{R}) = \sum_\alpha U^{(e-n)}(r_{i\alpha}) + \sum_{i<j} U^{(e-e)}(r_{ij}) + \sum_{\alpha i<j} U^{(e-e-n)}(r_{ij}, r_{i\alpha}, r_{j\alpha}) + \ldots \tag{28}$$

where U’s are simple functions (Many different expressions have been employed). The second part of the wavefunction is quite standard in chemistry and describes the shell-structure of molecules via a linear combination of a product of two Slater determinants built from one-electron molecular orbitals. Note that several other forms for the trial wavefunction have been introduced in the literature but so far they have remained of marginal use. Finally, let us emphasize that the magnitude of the statistical error and the importance of the fixed-node bias being directly related to the quality of the trial wavefunction (both errors vanish in the limit of an exact wavefunction), it is in general quite profitable to optimize the parameters of the trial wavefunction. Several approaches have been proposed, we just mention here the recently proposed method of Umrigar and collaborators [9].

Applications

In computational chemistry, the vast majority of the VMC and FN-DMC applications have been concerned with the calculation of total energies and differences of total energies: atomization energies, electronic affinities, ionization potentials, reaction barriers, excited-state energies, etc. To get a brief view of what can be achieved with QMC, let us mention the existence of several benchmark studies comparing FN-DMC with the standard DFT and post-HF methods [10–12]. In such studies, FN-DMC appears to be as accurate as the most accurate post-HF methods and advanced DFT approaches. In addition, like DFT – but in sharp contrast with the post-HF methods – the scaling of the computational cost as a function of the system size is favorable, typically in $O(N^3)$. However, QMC simulations are much more CPU-intensive than DFT ones. To date the largest systems studied involve about 2,000 active electrons, see, e.g., [13]. Finally, note that in principle, all chemical properties can be evaluated using QMC. Unfortunately, to reach the desired accuracy is often difficult in practice. More progress is needed to improve the QMC estimators of such properties.

QMC and High-Performance Computing (HPC)

Let us end by emphasizing on one of the most important practical aspect of QMC methods, namely, their remarkable adaptation to high performance computing (HPC) and, particularly, to massive parallel computations. As most Monte Carlo algorithms, the computational effort is almost exclusively concentrated on pure CPU (“number crunching method”).

In addition, – and this is the key aspect for massive parallelism – calculations of averages can be decomposed at will: n Monte Carlo steps over a single processor being equivalent to n／p steps over p processors with no communication between the processors (apart from the initial/final data transfers). Very recently, it has been demonstrated that an almost perfect parallel efficiency up to about 100,000 compute cores is achievable in practice [14, 15]. In view of the formidable development of computational platforms: Presently up to a few hundreds of thousands compute cores (petascale platforms) and many more soon (exascale in the near future) this property could be critical in assuring the success of QMC in the years to come.

References

1. Foulkes, W.M.C., Mitas, L., Needs, R.J., Rajagopal, G.: Quantum Monte Carlo simulations of Solids. Rev. Mod. Phys. 73, 33–83 (2001)
2. Risken, H.: The Fokker-Planck Equation: Methods of Solutions and Applications. Springer Series in Synergetics, 3rd edn. Springer, Berlin (1996)
3. Gilmore, R.: Baker-Campbell-Hausdorff formulas. J. Math. Phys. 15, 2090–2092 (1974)
4. Caffarel, M., Claverie, P.: Development of a pure diffusion quantum Monte Carlo method using a full generalized Feynman-Kac formula. I. Formalism. J. Chem. Phys. 88, 1088–1099 (1988)
5. Baroni, S., Moroni, S.: Reptation quantum Monte Carlo: a method for unbiased ground-state averages and imaginary-time correlations. Phys. Rev. Lett. 82, 4745–4748 (1999)
6. Ceperley, D.M.: Fermion nodes. J. Stat. Phys. 63, 1237–1267 (1991)
7. Cancès, E., Jourdain, B., Lelièvre, T.: Quantum Monte Carlo simulation of fermions. A mathematical analysis of the fixed node approximation. Math. Model Method App. Sci. 16, 1403–1440 (2006)
8. Ceperley, D.M., Alder, B.J.: Quantum Monte Carlo for molecules: Green's function and nodal release. J. Chem. Phys. 81, 5833–5844 (1984)
9. Umrigar, C.J., Toulouse, J., Filippi, C., Sorella, S., Hennig, R.G.: Alleviation of the Fermion-sign problem by optimization of many-body wave functions. Phys. Rev. Lett. 98, 110201 (2007)
10. Manten, S., Lüchow, A.: On the accuracy of the fixed-node diffusion quantum Monte Carlo methods. J. Chem. Phys. 115, 5362–5366 (2001)
11. Grossman, J.C.: Benchmark QMCarlo calculations. J. Chem. Phys. 117, 1434–1440 (2002)
12. Nemec, N., Towler, M.D., Needs, R.J.: Benchmark all-electron ab initio quantum Monte Carlo calculations for small molecules. J. Chem. Phys. 132, 034111-7 (2010)
13. Sola, E., Brodholt, J.P., Alfè, D.: Equation of state of hexagonal closed packed iron under Earth's core conditions from quantum Monte Carlo calculations. Phys. Rev. B 79: 024107-6 (2009)
14. Esler, K.P., Kim, J., Ceperley, D.M., Purwanto, W., Walter, E.J., Krakauer, H., Zhang, S.: Quantum Monte Carlo algorithms for electronic structure at the petascale; the endstation project. J. Phys. Conf. Ser. 125 012057 (2008)
15. Gillan, M.J., Towler, M.D., Alfè, D.: Petascale computing opens new vistas for quantum Monte Carlo Psi-k Highlight of the Month (February, 2011) (2011)

Quantum Monte Carlo Methods in Chemistry

Michel Caffarel — Laboratoire de Chimie et Physique Quantiques, IRSAMC, Université de Toulouse, Toulouse, France

DOI: 10.1007/SpringerReference_333776

URL: http://www.springerreference.com/index/chapterdbid/333776

Part of: Encyclopedia of Applied and Computational Mathematics

Editors: -

PDF created on: July, 16, 2012 13:27

Quantum Monte Carlo for Large Chemical Systems: Implementing Efficient Strategies for Petascale Platforms and Beyond

Anthony Scemama,*[a] Michel Caffarel,[a] Emmanuel Oseret,[b] and William Jalby[b]

Various strategies to implement efficiently quantum Monte Carlo (QMC) simulations for large chemical systems are presented. These include: (i) the introduction of an efficient algorithm to calculate the computationally expensive Slater matrices. This novel scheme is based on the use of the highly localized character of *atomic* Gaussian basis functions (not the *molecular* orbitals as usually done), (ii) the possibility of keeping the memory footprint minimal, (iii) the important enhancement of single-core performance when efficient optimization tools are used, and (iv) the definition of a universal, dynamic, fault-tolerant, and load-balanced framework adapted to all kinds of computational platforms (massively parallel machines, clusters, or distributed grids). These strategies have been implemented in the QMC=Chem code developed at Toulouse and illustrated with numerical applications on small peptides of increasing sizes (158, 434, 1056, and 1731 electrons). Using 10–80 k computing cores of the Curie machine (GENCI-TGCC-CEA, France), QMC=Chem has been shown to be capable of running at the petascale level, thus demonstrating that for this machine a large part of the peak performance can be achieved. Implementation of large-scale QMC simulations for future exascale platforms with a comparable level of efficiency is expected to be feasible. © 2013 Wiley Periodicals, Inc.

DOI: 10.1002/jcc.23216

Introduction

Quantum Monte Carlo (QMC) is a generic name for a large class of stochastic approaches solving the Schrödinger equation by using random walks. In the last 40 years, they have been extensively used in several fields of physics including nuclear physics,[1] condensed-matter physics,[2] spin systems,[3] quantum liquids,[4] infrared spectroscopy,[5,6] and so on. In these domains, QMC methods are usually considered as routine methods and even in most cases as state-of-the-art approaches. In sharp contrast, this is not yet the case for the electronic structure problem of quantum chemistry, where QMC[7,8] is still of confidential use when compared to the two well-established methods of the domain [Density Functional Theory (DFT) and post-Hartree–Fock methods]. Without entering into the details of the forces and weaknesses of each approach, a major limiting aspect of QMC hindering its diffusion is the high computational cost of the simulations for realistic systems.

However—and this is the major concern of this work—a unique and fundamental property of QMC methods is their remarkable adaptation to high-performance computing (HPC) and, particularly, to massively parallel computations. In short, the algorithms are simple and repetitive, central memory requirements may be kept limited whatever the system size, and I/O flows are negligible. As most Monte Carlo algorithms, the computational effort is almost exclusively concentrated on pure CPU ("number crunching method") and the execution time is directly proportional to the number of Monte Carlo steps performed. In addition, and this is a central point for massive parallelism, calculations of averages can be decomposed at will: n Monte Carlo steps over a single processor being equivalent to n/p Monte Carlo steps over p processors with no communication between the processors (apart from the initial/final data transfers). Once the QMC algorithm is suitably implemented the maximum gain of parallelism (ideal scalability) should be expected.

A most important point is that mainstream high-level quantum chemistry methods do not enjoy such a remarkable property. They are essentially based on iterative schemes defined within the framework of linear algebra and involve the manipulation and storage of extremely large matrices. Their adaptation to extreme parallelism is intrinsically problematic.

Now, in view of the formidable development of computational platforms, particularly in terms of the number of computing cores (presently up to a few hundreds of thousands and many more to come) the practical bottleneck associated with the high-computational cost of QMC is expected to become much less critical. Thus, QMC may become in the coming years a method of practical use for treating chemical problems out of the reach of present-day approaches. Following this line of thought, a number of QMC groups are presently working on implementing strategies allowing their QMC codes to run efficiently on very large-scale parallel

[a] A. Scemama, M. Caffarel
Laboratoire de Chimie et Physique Quantiques, CNRS-IRSAMC, Université de Toulouse, France
[b] E. Oseret, W. Jalby
Exascale Computing Research Laboratory, GENCI-CEA-INTEL-UVSQ, Université de Versailles Saint-Quentin, France
E-mail: scemama@irsamc.ups-tlse.fr

Contract/grant sponsor: ANR (to AS and MC); Contract/grant number: ANR 2011 BS08 004 01.

computers.[9–11] Essentially, most strategies rely on massive parallelism and on some efficient treatment ("linear-scaling"-type algorithms) for dealing with the matrix computations and manipulations that represent the most CPU-expensive part of the algorithm.

Here, we present several strategies implemented in the QMC=Chem code developed in our group at the University of Toulouse.[12] A number of actual simulations realized on the Curie machine at the French GENCI-TGCC-CEA computing center with almost ideal parallel efficiency in the range 10,000–80,000 cores and reaching the petascale level have been realized.

The contents of this article are as follows. In the first section, a brief account of the QMC method used is presented. Only those aspects essential to the understanding of the computational aspects discussed in this article are given. In second section, the problem of computing efficiently the Slater matrices at the heart of the QMC algorithm (computational hot spot) is addressed. A novel scheme taking advantage of the highly-localized character of the *atomic* Gaussian basis functions [not the molecular orbitals (MOs) as usually done] is proposed. A crucial point is that the approach is valid for an arbitrary molecular shape (e.g., compact molecules), there is no need of considering extended or quasi-one-dimensional molecular systems as in linear-scaling approaches. The third section discusses the overall performance of the code and illustrates how much optimizing the single-core performance of the specific processor at hand can be advantageous. The fourth section is devoted to the way our massively parallel simulations are deployed on a general computational platform and, particularly, how fault-tolerance is implemented, a crucial property for any large-scale simulation. Finally, a summary of the various strategies proposed in this article is presented in the last section.

The QMC Method

In this article, we shall consider a variant of the fixed-node diffusion Monte Carlo (FN-DMC) approach, the standard QMC method used in computational chemistry. Here, we shall insist only on the aspects needed for understanding the rest of the work. For a complete presentation of the FN-DMC method, the reader is referred, for example, to Refs. [2], [7], or [8] and references therein.

Fixed-node diffusion Monte Carlo (FN-DMC)

Diffusion Monte Carlo. In a diffusion Monte Carlo scheme, a finite population of "configurations" or "walkers" moving in the 3*N*-dimensional space (*N*, number of electrons) is introduced. A walker is described by a 3*N*-dimensional vector $\mathbf{R} \equiv (\mathbf{r}_1, \mathbf{r}_2, \ldots, \mathbf{r}_N)$ giving the positions of the *N* electrons. At each Monte Carlo step, each walker of the population is diffused and drifted according to

$$\mathbf{R}' = \mathbf{R} + \tau \mathbf{b}(\mathbf{R}) + \sqrt{\tau}\boldsymbol{\eta} \tag{1}$$

where τ is a small time-step, $\boldsymbol{\eta}$ is a Gaussian vector (3*N* independent normally distributed components simulating a free Brownian diffusion), and $\mathbf{b}(\mathbf{R})$ the drift vector given by

$$\mathbf{b}(\mathbf{R}) \equiv \frac{\nabla \psi_T(\mathbf{R})}{\psi_T(\mathbf{R})}, \tag{2}$$

where ψ_T, the trial wave function, is a known computable approximation of the exact wavefunction. At the end of this drift/diffusion step, each walker is killed, kept unchanged, or duplicated a certain number of times proportionally to the branching weight *w* given by

$$w = e^{-\frac{\tau}{2}[(E_L(\mathbf{R}')-E_T)+(E_L(\mathbf{R})-E_T)]} \tag{3}$$

where E_T is some reference energy and E_L the local energy defined as

$$E_L(\mathbf{R}) \equiv \frac{H\psi_T(\mathbf{R})}{\psi_T(\mathbf{R})}. \tag{4}$$

The population is propagated and after some equilibrium time it enters a stationary regime, where averages are evaluated. As an important example, the exact energy may be obtained as the average of the local energy.

The fixed-node approximation. Apart from the statistical and the short-time (finite time step) errors which can be made arbitrary small, the only systematic error left in a DMC simulation is the so-called fixed-node (FN) error. This error results from the fact that the nodes of the trial wavefunction [defined as the $(3N-1)$-dimensional hypersurface where $\Psi_T(\mathbf{R}) = 0$] act as infinitely repulsive barriers for the walkers [divergence of the drift vector, eq. (2)]. Each walker is thus trapped forever within the nodal pocket delimited by the nodes of Ψ_T where it starts from. When the nodes of ψ_T coincide with the exact nodes, the algorithm is exact. If not, a variational FN error is introduced. However, with the standard trial wavefunctions used, this error is in general small,* a few percent of the correlation energy for total energies.

Parallelizing FN-DMC

Each Monte Carlo step is carried out *independently* for each walker of the population. The algorithm can thus be easily parallelized over an arbitrary number of processors by distributing the walkers among the processors, but doing this implies synchronizations of the CPUs since the branching step requires that all the walkers have first finished their drifted-diffusion step.

To avoid this aspect, we have chosen to let each CPU core manage its own population of walkers without any communication between the populations. On each computing unit a population of walkers is propagated and the various averages of interest are evaluated. At the end of the simulation, the

*A word of caution is necessary here. Although the FN error on total energies is indeed usually very small compared with typical errors of standard computational chemistry methods, this error can still be large enough to have a non-negligible impact on *small energy differences* of interest in chemistry (binding energies, energy barriers, electronic affinities, etc.). Accordingly, to have to our disposal nodal hypersurfaces of sufficient quality for a general molecular system remains an important issue of QMC approaches.

averages obtained on each processor are collected and summed up to give the final answers. Regarding parallelism the situation is thus ideal since, apart from the negligible initial/final data transfers, there are no communications among processors.

The only practical problem left with FN-DMC is that the branching process causes fluctuations in the population size and thus may lead to load-balancing problem among processors. More precisely, nothing prevents the population size from decreasing or increasing indefinitely during the Monte Carlo iterations. To escape from this, a common solution consists in forcing the number of walkers not to deviate too much from some target value for the population size by introducing a population control step. It is usually realized by monitoring in time the value of the reference energy E_T via a feedback mechanism, see, for example, Ref. [13]. The price to pay is the introduction of some transient load imbalances and inter-processor communications/synchronization to redistribute walkers among computing cores, inevitably degrading the parallel speedup. This solution has been adapted by several groups and some tricks have been proposed to keep this problem under control.[9–11,14]

Here, we propose to avoid this problem directly from the beginning by using a variant of the FN-DMC working with a constant number of walkers. Several proposals can be found in the literature, for example, Refs. [15,16]. Here, we shall use the method described in Ref. [16]. In this approach, the branching step of standard DMC is replaced by a so-called reconfiguration step. Defining the *normalized* branching weights as follows:

$$p_k = \frac{w_k}{\sum_{i=1}^{M} w_i} \tag{5}$$

the population of walkers is "reconfigured" by drawing at each step M walkers among the M walkers according to the probabilities p_k. At infinite population, the normalization factor $\sum_{i=1}^{M} w_i$ is a constant and this step reduces to the standard branching step, where walkers are deleted or duplicated proportionally to the weight w. At finite M, the normalization factor now fluctuates and a finite-population bias is introduced. A simple way to remove this error and to recover the exact averages consists in adding to the averages a global weight given by the product of the normalization factors of all preceding generations, thus compensating exactly the same product introduced into the dynamics by successive reconfiguration steps. The price to pay is some increase of statistical fluctuations due to the presence of an additional fluctuating weight. However, this increase is found to be rapidly very moderate when M is increased. In practice, thanks to this algorithm free of a finite-population bias, rather small walker populations on each core can be used (typically, we use 10–100 walkers per core). For all details, the reader is referred to Ref. [16].

Critical CPU part

At each Monte Carlo step, the CPU effort is almost completely dominated by the evaluation of the wavefunction Ψ_T and its first and second derivatives (computational hot spot). More precisely, for each walker the values of the trial wavefunction, Ψ_T, its first derivatives with respect to all $3N$-coordinates [drift vector, eq. (2)], and its Laplacian $\nabla^2\Psi_T$ [kinetic part of the local energy, eq. (4)] are to be calculated. It is essential that such calculations be as efficient as possible since in realistic applications their number may be very large (typically of the order of 10^9–10^{12}).

A common form for the trial wavefunction is

$$\Psi_T(\mathbf{R}) = e^{J(\mathbf{R})} \sum_{K=(K_\uparrow,K_\downarrow)} c_K \mathrm{Det}_{K_\uparrow}(\mathbf{r}_1,\ldots,\mathbf{r}_{N_\uparrow})\mathrm{Det}_{K_\downarrow}(\mathbf{r}_{N_\downarrow},\ldots,\mathbf{r}_N). \tag{6}$$

where the electron coordinates of the $N_\uparrow$ (respectively, $N_\downarrow$) electrons of spin $\uparrow$ (respectively, $\downarrow$) have been distinguished, $N = N_\uparrow + N_\downarrow$. In this formula, $e^{J(\mathbf{R})}$ is the Jastrow factor describing explicitly the electron–electron interactions at different levels of approximations. A quite general form may be written as

$$J(\mathbf{R}) = \sum_{\alpha} U^{(e-n)}(r_{i\alpha}) + \sum_{i,j} U^{(e-e)}(r_{ij}) + \sum_{\alpha i,j} U^{(e-e-n)}(r_{ij}, r_{i\alpha}, r_{j\alpha}) + \ldots \tag{7}$$

where $r_{ij} = |\mathbf{r}_i - \mathbf{r}_j|$ is the inter-electronic distance and $r_{i\alpha} = |\mathbf{r}_i - \mathbf{Q}^\alpha|$ is the distance between electron i and nucleus α located at $\mathbf{Q}^\alpha$. Here, U's are simple functions and various expressions have been used in the literature. The Jastrow factor being essentially local, short-ranged expressions can be used and the calculation of this term is usually a small contribution to the total computational cost. As a consequence, we shall not discuss further the computational aspect of this term here.

The second part of the wavefunction describes the shell-structure in terms of single-electron MOs and is written as a linear combination of products of two Slater determinants, one for the $\uparrow$ electrons and the other for the $\downarrow$ electrons. Each Slater matrix is built from a set of MOs $\phi_i(\mathbf{r})$ usually obtained from a preliminary DFT or self consistent field (SCF) calculations. The N_{orb} molecular orbitals (MOs) are expressed as a sum over a finite set of N_{basis} basis functions [atomic orbitals (AOs)]

$$\phi_i(\mathbf{r}) = \sum_{j=1}^{N_{\mathrm{basis}}} a_{ij}\chi_j(\mathbf{r}) \tag{8}$$

where the basis functions $\chi_j(\mathbf{r})$ are usually expressed as a product of a polynomial and a linear combination of Gaussian functions. In the present article, the following standard form is used

$$\chi(\mathbf{r}) = (x - Q_x)^{n_x}(y - Q_y)^{n_y}(z - Q_z)^{n_z} g(\mathbf{r}) \tag{9}$$

with

$$g(\mathbf{r}) = \sum_k c_k e^{-\gamma_k(\mathbf{r}-\mathbf{Q})^2}. \tag{10}$$

Here, $\mathbf{Q} = (Q_x, Q_y, Q_z)$ is the vector position of the nucleus-center of the basis function, $\mathbf{n} = (n_x, n_y, n_z)$ a triplet of positive

integers, $g(\mathbf{r})$ is the spherical Gaussian component of the AO, and γ_k its exponents. The determinants corresponding to spin ↑-electrons are expressed as

$$\mathrm{Det}_{K_\uparrow}(\mathbf{r}_1,\ldots,\mathbf{r}_{N_\uparrow}) = \mathrm{Det}\begin{pmatrix} \phi_{i_1}(\mathbf{r}_1) & \ldots & \phi_{i_1}(\mathbf{r}_{N_\uparrow}) \\ \vdots & \vdots & \vdots \\ \phi_{i_{N_\uparrow}}(\mathbf{r}_1) & \ldots & \phi_{i_{N\uparrow}}(\mathbf{r}_{N_\uparrow}) \end{pmatrix} \quad (11)$$

where $K_\uparrow$ is a compact notation for denoting the set of indices $\{i_1, \ldots, i_{N_\uparrow}\}$ specifying the subset of the MOs used for this particular Slater matrix. A similar expression is written for spin ↓-electrons.

In contrast to the calculation of the Jastrow factor, the evaluation of the determinantal part of the wavefunction and its derivatives is critical. To perform such calculations, we use a standard approach[7] consisting in calculating the matrices of the first and second (diagonal) derivatives of each MO ϕ_i with respect to the three space variables $I = x, y, z$ evaluated for each electron position $\mathbf{r}_j$, namely,

$$D^{(1)}_{I,i\,j} \equiv \frac{\partial\phi_i(\mathbf{r}_j)}{\partial x^j_I} \quad (12)$$

$$D^{(2)}_{I,i\,j} \equiv \frac{\partial^2\phi_i(\mathbf{r}_j)}{\partial x^{j\,2}_I} \quad (13)$$

and then computing the inverse D^{-1} of the Slater matrix defined as $D_{ij} = \phi_i(\mathbf{r}_j)$. The drift components and the Laplacian corresponding to the determinantal part of the trial wavefunction are thus evaluated as simple vector-products

$$\frac{1}{\mathrm{Det}(\mathbf{R})}\frac{\partial\mathrm{Det}(\mathbf{R})}{\partial x^i_I} = \sum_{j=1,\,N} D^{(1)}_{I,i\,j}D^{-1}_{ji} \quad (14)$$

$$\frac{1}{\mathrm{Det}(\mathbf{R})}\frac{\partial^2\mathrm{Det}(\mathbf{R})}{\partial x^{i\,2}_I} = \sum_{j=1,\,N} D^{(2)}_{I,i\,j}D^{-1}_{ji} \quad (15)$$

From a numerical point of view, the computational time T needed to evaluate such quantities as a function of the number of electrons N scales as $\mathcal{O}(N^3)$

$$T = \alpha N^3 + \beta N^3. \quad (16)$$

The first N^3-term results from the fact that the N^2 matrix elements of the Slater matrices are to be computed, each element being expressed in terms of the $N_{\mathrm{basis}} \sim N$ basis functions needed to reproduce an arbitrary delocalized MO. The second N^3-term is associated with the generic cubic scaling of any linear algebra method for inverting a general matrix.

Exploiting the Highly Localized Character of Atomic Basis Functions

As seen in the previous section, one of the two computational hot spots of QMC is the calculation of the derivatives of the determinantal part of the trial wave function for each electronic configuration $(\mathbf{r}_1,\ldots,\mathbf{r}_N)$ at each Monte Carlo step. To be more precise, the N_{orb} MO used in the determinantal expansion (6) are to be computed (here, their values will be denoted as $\mathbf{C}_1$) together with their first derivatives with respect to x, y, and z (denoted $\mathbf{C}_2$,$\mathbf{C}_3$,$\mathbf{C}_4$) and their Laplacians (denoted $\mathbf{C}_5$). Calculations are made in single precision using an efficient matrix product routine we describe now. The matrix products involve the matrix of the MO coefficients a_{ij}, eq. (8) (here denoted as $\mathbf{A}$) the matrix of the atomic Gaussian basis functions evaluated at all electronic positions, $\chi_j(\mathbf{r}_i)$ (denoted $\mathbf{B}_1$), their first derivatives (denoted $\mathbf{B}_2$,$\mathbf{B}_3$,$\mathbf{B}_4$), and Laplacians (denoted $\mathbf{B}_5$). The five matrix products are written under the convenient form

$$\mathbf{C}_i = \mathbf{A}\mathbf{B}_i \quad i = 1,5 \quad (17)$$

Note that matrix $\mathbf{A}$ remains constant during the simulation, whereas matrices $\mathbf{B}_i$ and $\mathbf{C}_i$ depend on electronic configurations. The matrix sizes are as follows: $N_{\mathrm{orb}} \times N$ for the $\mathbf{C}_i$'s, $N_{\mathrm{orb}} \times N_{\mathrm{basis}}$ for $\mathbf{A}$, and $N_{\mathrm{basis}} \times N$ for $\mathbf{B}$. In practical applications, N_{orb} is of the order of N, whereas N_{basis} is greater than N by a factor 2 or 3 for standard calculations and much more when using high-quality larger basis sets. The expensive part is essentially dominated by the N_{basis} multiplications. The total computational effort is thus of order $N_{\mathrm{orb}} \times N \times N_{\mathrm{basis}}$, that is, $\sim \mathcal{O}(N^3)$.

The standard approach proposed in the literature for reducing the N^3-price is to resort to the so-called linear-scaling or $\mathcal{O}(N)$-techniques.[17–22] The basic idea consists in introducing *spatially localized* MOs instead of the standard delocalized (canonical) ones obtained from diagonalization of reference Hamiltonians (usually, Hartree–Fock or Kohn–Sham). Since localized orbitals take their value in a finite region of space—usually in the vicinity of a fragment of the molecule—the number of basis set functions N_{basis} needed to represent them with sufficient accuracy becomes essentially independent of the system size (not scaling with N as in the case of canonical ones). In addition to this, each electron contributes only to a small subset of the localized orbitals (those nonvanishing in the region where the electron is located). As a consequence, the number of nonvanishing matrix elements of the $\mathbf{C}_i$ matrices no longer scales as $N_{\mathrm{orb}} \times N \sim N^2$ but linearly with N. Furthermore, each matrix element whose computation was proportional to the number of basis set functions used, $N_{\mathrm{basis}} \sim N$, is now calculated in a finite time independent of the system size. Putting together these two results, we are led to a linear dependence of the computation of the $\mathbf{C}_i$ matrices upon the number of electrons.

Here, we choose to follow a different path. Instead of localizing the canonical MOs, we propose to take advantage of the localized character of the underlying *atomic* Gaussian basis set functions. The advantages are essentially three-fold:

1. The atomic basis set functions are naturally localized *independently of the shape of the molecule*. This is the most important point since the localization procedures are known to be effective for chemical systems having a molecular shape made of well-separated subunits (e.g., linear systems) but much less for general compact molecular systems that are ubiquitous in chemistry.

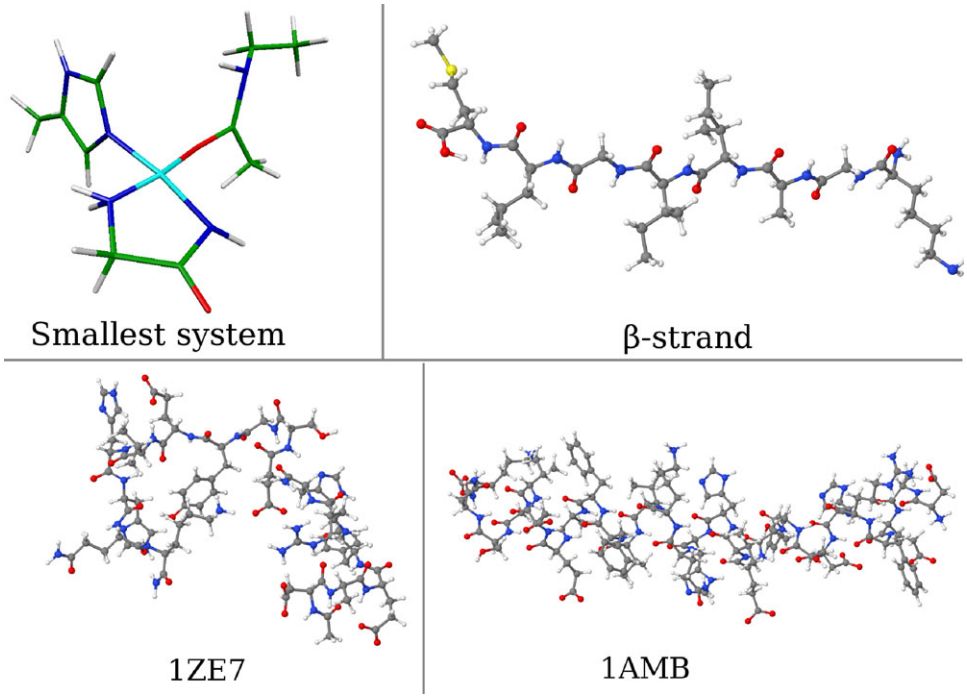


Figure 1. Molecular systems used as benchmarks.

2. The degree of localization of the standard atomic Gaussian functions is much larger than that obtained for MOs after localization (see results below)

3. By using the product form, eq. (17), the localized nature of the atomic Gaussian functions can be exploited very efficiently (see next section).

In practice, when the value of the spherical Gaussian part $g(\boldsymbol{r})$ of an AO function $\chi(\boldsymbol{r})$ is smaller than a given threshold $\varepsilon = 10^{-8}$, the value of the AO, its gradients and Laplacian are considered null. This property is used to consider the matrices $\mathbf{B}_1,\ldots,\mathbf{B}_5$ as sparse. However, in contrast with linear-scaling approaches, the MO matrix **A** is not considered here as sparse. We shall come back to this point later. To accelerate the calculations, an atomic radius is computed as the distance beyond which all the Gaussian components $g(\boldsymbol{r})$ of the AOs $\chi(\boldsymbol{r})$ centered on the nucleus are less than ε. If an electron is farther than the atomic radius, all the AO values, gradients and Laplacians centered on the nucleus are set to zero.

The practical implementation to perform the matrix products is as follows. For each electron, the list of indices (array "indices" in what follows) where $g(\mathbf{r}) > 0$ is calculated. Then, the practical algorithm can be written as

```
C1 = 0.
C2 = 0.
C3 = 0.
C4 = 0.
C5 = 0.
do i=1, Number of electrons
do k=1, Number of non-zero AOs for electron i
 do j=1, Number of molecular orbitals
  C1 (j, i) += A (j, indices (k, i)) *B1 (k, i)
  C2 (j, i) += A (j, indices(k, i)) *B2 (k, i)
  C3 (j, i) += A (j, indices(k, i)) *B3 (k, i)
  C4 (j, i) += A (j, indices(k, i)) *B4 (k, i)
  C5 (j, i) += A (j, indices(k, i)) *B5 (k, i)
end do
end do
end do
```

(where x += y denotes x = x + y).

This implementation allows to take account of the sparsity of the **B** matrices, while keeping the efficiency due to a possible vectorization of the inner loop. The load/store ratio is 6/5 (6 load-from-memory instructions, 5 store-to-memory instructions) in the inner loop: the elements of $\mathbf{B}_n$ are constant in the inner loop (in registers), and the same element of **A** is used at each line of the inner loop (loaded once per loop cycle). As store operations are more expensive than load operations, increasing the load/store ratio improves performance as will be shown in the next section. Using this algorithm, the scaling of the matrix products is expected to drop from $\mathcal{O}(N^3)$ to a scaling roughly equal to $\mathcal{O}(N^2)$ (in a regime where N is large enough, see discussion in the next section). Let us now illustrate such a property in the applications to follow.

The different systems used here as benchmarks are represented in Figure 1. The trial wavefunctions used for describing each system are standard Hartree–Fock wavefunctions (no Jastrow factor) with MOs expressed using various Gaussian basis sets. System 1 is a copper complex with four ligands having 158 electrons and described with a cc-pVDZ basis set. System 2 is a polypeptide taken from Ref. [23] (434 electrons and 6-31G* basis set). System 3 (not shown in Figure 1) is identical

Table 1. System sizes, percentage of nonzero molecular orbital coefficients, and average percentage of nonzero atomic orbital values.

	Smallest system	β-strand	β-strand TZ	1ZE7	1AMB
Numb. of electrons, N	158	434	434	1056	1731
Numb. of basis functions, N_{basis}	404	963	2934	2370	3892
% of non-zero[a] *canonical* MO coefficients $a_{ij}(A_{ij} \neq 0)$	(99.4%)	(76.0%	(81.9%)	(72.0%)	(66.1%)
% of non-zero[a] *localized* MO coefficients $a_{ij}(A_{ij} \neq 0)$	81.3%	48.4%	73.4%	49.4%	37.1%
Average % of non-zero[b] basis functions $\chi_i(\mathbf{r}_j)$ $(B_{1ij} \neq 0)$	40.3%	19.1%	9.0%	6.5%	4.5%
Average number of non-zero elements per column of B_{1ij}	163	184	266	155	175
Maximum number of non-zero elements per column of B_{1ij}	251	298	394	246	305

[a] Zero MO coefficients are those below 10^{-5}. [b] Zero AO matrix elements are those for which the radial component of the basis function has a value below 10^{-8} for given electron positions.

to System 2 but using a larger basis set, namely, the cc-pVTZ basis set. System 4 is the 1ZE7 molecule from the Protein Data Bank (1056 electrons, 6-31G*), and System 5 is the 1AMB molecule from the Protein Data Bank (1731 electrons, 6-31G*).

Table 1 shows the level of sparsity of the matrices **A** ($A_{ij} \equiv a_{ij}$) and $\mathbf{B}_1$ ($B_{1_{ij}} \equiv \chi_i(\mathbf{r}_j)$) for the five systems (matrices $\mathbf{B}_n$ with $n > 1$ behave as $\mathbf{B}_1$ with respect to sparsity). As seen the number of basis set functions used is proportional to the number of electrons with a factor ranging from about 2.2 to 6.8.

Regarding the matrix **A** of MO coefficients, the results are given both for standard canonical (delocalized) MOs and for localized orbitals. To get the latter ones, different localization schemes have been applied.[24–26] However, they essentially lead to similar results. Here, the results presented are those obtained by using the Cholesky decomposition of the density matrix expressed in the AO basis set.[26] As seen the level of sparsity of the matrix **A** is low. Although it increases here with the system size it remains modest for the largest size (there are still about one third of nonzero elements). Of course, such a result strongly depends on the type of molecular system considered (compact or not compact) and on the diffuse character of the atomic basis set. Here, we have considered typical systems of biochemistry.

Next, the level of sparsity of the **B** matrices is illustrated. The percentage of nonzero values of $\chi_i(\mathbf{r}_j)$ has been obtained as an average over a variational Monte Carlo (VMC) run. In sharp contrast with MOs the AOs are much more localized, thus leading to a high level of sparsity. For the largest system, only 3.9% of the basis function values are nonnegligible.

In the last line of the table the maximum number of nonzero elements obtained for all the columns of the matrix during the entire Monte Carlo simulation is given. A first remark is that this number is roughly constant for all system sizes. A second remark is that the maximum number of non-zero values is only slightly greater than the average, thus showing that the **B** matrices can be considered sparse during the whole simulation, not only in average. As an important consequence, the loop over the number of non-zero AOs for each electron in the practical algorithm presented above (loop over k index) is expected to be roughly constant as a function of the size at each Monte Carlo step. This latter remark implies for this part an expected behavior of order $\mathcal{O}(N^2)$ for large N. Let us now have a closer look at the actual performance of the code.

Overall Performance of QMC=CHEM

When discussing performance several aspects must be considered. A first one, which is traditionally discussed, is the formal scaling of the code as a function of the system size N ($N \sim$ number of electrons). As already noted, due to the innermost calculation, products, and inversion of matrices, such a scaling is expected to be cubic, $\mathcal{O}(N^3)$. However, there is a second important aspect, generally not discussed, which is related to the way the expensive innermost floating-point operations are implemented and on how far and how efficiently the potential performance of the processor at hand is exploited. In what follows, we shall refer to this aspect as "single-core optimization." It is important to emphasize that such an aspect is by no way minor and independent on the previous "mathematical" one.

To explicit this point, let us first recall that the computational time T results essentially from two independent parts, the first one resulting from the computation of the matrix elements, $T_1 \sim \alpha N^3$ and the second one from the inversion of the Slater matrix, $T_2 \sim \beta N^3$. Now, let us imagine that we have been capable of devising a highly efficient linear-scaling algorithm for the first contribution such that $T \sim \varepsilon N \ll T_2$ within the whole range of system sizes N considered. We would naturally conclude that the overall computational cost $T \sim T_2$ is cubic. In the opposite case where a very inefficient linear-scaling algorithm is used for the first part, $T \sim T_1 \gg T_2$, we would conclude to a linear-scaling type behavior. Of course, mathematically speaking such a way of reasoning is not correct since scaling laws are only meaningful in the *asymptotic* regime where N goes to infinity. However, in practice only a *finite range of sizes is considered* (here, between 2 and about 2000 active electrons) and it is important to be very cautious with the notion of scaling laws. A more correct point of view consists in looking at the global performance of the code in terms of total CPU time for a given range of system sizes, a given compiler, and a given type of CPU core.

Finally, a last aspect concerns the memory footprint of the code whose minimization turns out to be very advantageous. Indeed, the current trend in supercomputer design is to increase the number of cores more rapidly than the available total memory. As the amount of memory per core will continue to decrease, it is very likely that programs will need to have a low memory footprint to take advantage of exascale

Table 2. Single core performance (GFlops/s) of the matrix products (single precision), inversion (double precision), and overall performance of QMC = Chem (mixed single/double precision).

	Core2			Sandy Bridge		
	Products	Inversion	Overall	Products	Inversion	Overall
Linpack (DP)		7.9 (84.9%)			24.3 (92.0%)	
Peak	18.6	9.3		52.8	26.4	
Smallest system	9.8 (52.7%)	2.6 (28.0%)	3.3	26.6 (50.3%)	8.8 (33.3%)	6.3
β-Strand	9.7 (52.2%)	4.3 (46.2%)	3.7	33.1 (62.7%)	13.7 (51.2%)	13.0
β-Strand TZ	9.9 (53.2%)	4.3 (46.2%	4.5	33.6 (63.6%)	13.7 (51.2%)	14.0
1ZE7	9.3 (50.0%)	5.2 (55.9%	4.6	30.6 (57.9%)	15.2 (57.6%)	17.9
1AMB	9.2 (49.5%)	5.6 (60.2%	5.0	28.2 (53.4%)	16.2 (61.4%)	17.8

The percentage of the peak performance is given in parentheses.
Core2: Intel Xeon 5140, Core2 2.33 GHz, Dual core, 4 MiB shared L2 cache.
Sandy Bridge: Intel Xeon E3–1240, Sandy Bridge 3.30 GHz, Quad core, 256 KiB L2 cache/core, 8 MiB shared L3 cache (3.4 GHz with turbo).

computers. Another point is that when less memory is used less electrical power is needed to perform the calculation: data movement from the memory modules to the cores needs more electrical power than performing floating point operations. Although at present time the power consumption is not yet a concern to software developers, it is a key aspect in present design of the exascale machines to come.

In this section, the results discussed will be systematically presented by using two different generations of Intel Xeon processors. The first processor, referred to as *Core2*, is an Intel Xeon 5140, Core2 2.33 GHz, Dual core, 4 MiB shared L2 cache. The second one, referred to as *Sandy Bridge*, is an Intel Xeon E3-1240 at 3.30 GHz, Quad core, 256 KiB L2 cache/core, 8 MiB shared L3 cache (3.4 GHz with turbo). Note also that the parallel scaling of QMC being close to ideal (see next section), single-core optimization is very interesting: the gain in execution time obtained on the single-core executable is directly transferred to the whole parallel simulation.

Improving the innermost expensive floating-point operations

For the Core2 architecture, the practical algorithm presented above may be further improved by first using the *unroll and jam* technique,[27] which consists in unrolling the outer loop and merging multiple outer-loop iterations in the inner loop:

```
do i=1, Number of electrons
 do k=1, Number of non-zero AOs for electron i, 2
  do j=1, Number of molecular orbitals
   C1 (j, i) += A (j, indices (k, i)) *B1 (k, i) + &
                A (j, indices (k+1, i)) * B1 (k+1, i)
   C2 (j, i) += A (j, indices (k, i)) * B2 (k ,i) + &
                A (j, indices (k+1, i)) * B2 (k+1, i)
   ...
  end do
 end do
end do
```

To avoid register spilling, the inner loop is split in two loops: one loop computing $\mathbf{C}_1$, $\mathbf{C}_2$, $\mathbf{C}_3$ and a second loop computing $\mathbf{C}_4$, $\mathbf{C}_5$. The load/store ratio is improved from 6/5 to 5/3 and 4/2.

For the Sandy Bridge architecture, the external body is unrolled four times instead of two, and the most internal loop is split in three loops: one loop computing $\mathbf{C}_1$, $\mathbf{C}_2$, a second loop computing $\mathbf{C}_3$, $\mathbf{C}_4$, and a third loop computing $\mathbf{C}_5$. The load/store ratio is improved from 6/5 to 6/2 and 5/1.

Then, all arrays were 256-bit aligned using compiler directives and the first dimensions of all arrays were set to a multiple of eight elements (if necessary, padded with zeros at the end of each column) to force a 256-bit alignment of every column of the matrices. These modifications allowed the compiler to use *only* vector instructions to perform the matrix products, both with the Streaming SIMD Extension (SSE) or the Advanced Vector Extension (AVX) instruction sets. The x86_64 version of the MAQAO framework[28] indicates that, as the compiler unrolled twice the third loop ($\mathbf{C}_5$), these three loops perform 16 floating point operations per cycle, which is the peak performance on this architecture.

Finally, to improve the cache hit probability, blocking was used on the first dimension of $\mathbf{B}_n$ (loop over k). In each block, the electrons (columns of **B**) are sorted by ascending first element of the indices array in the block. This increases the probability that columns of **A** will be in the cache for the computation of the values associated with the next electron.

The results obtained using the Intel Fortran Compiler XE 2011 are presented in Table 2 for both the Core2 and the Sandy Bridge architectures. The single-core double-precision Linpack benchmark is also mentioned for comparison. The results show that the full performance of the matrix products is already reached for the smallest system. However, as opposed to dense matrix product routines, we could not approach further the peak performance of the processor since the number of memory accesses scales as the number of floating point operations (both $\mathcal{O}(N^2)$): the limiting factor is inevitably the data access. Nevertheless, the DECAN tool[29] revealed that data access only adds a 30% penalty on the pure arithmetic time, indicating an excellent use of the hierarchical memory and the prefetchers.

Single-core performance

Computational cost as a function of the system size. In Table 3, the memory required together with the CPU time obtained for

Table 3. Single-core memory consumption and elapsed time for one VMC step.

	Smallest system	β-strand	β-strand TZ	1ZE7	1AMB
RAM (MiB)	9.8	31	65	133	313
Core2					
QMC step(s)	0.0062	0.0391	0.0524	0.2723	0.9703
Inversion	15%	31%	21%	47%	58%
Products	25%	23%	35%	21%	18%
Sandy Bridge					
QMC step(s)	0.0026	0.0119	0.0187	0.0860	0.3042
Inversion	12%	26%	17%	42%	52%
Products	24%	22%	32%	21%	20%

Values in % represent the percentage of the total CPU time.
Core2: Intel Xeon 5140, Core2 2.33 GHz, Dual core, 4 MiB shared L2 cache.
Sandy Bridge: Intel Xeon E3–1240, Sandy Bridge 3.30 GHz, Quad core, 256 KiB L2 cache/core, 8 MiB shared L3 cache (3.4 GHz with turbo).

one VMC step for the five systems are presented using both processors. The two expensive computational parts (matrix products and inversion) are distinguished. A first remark is that the trends for both processors are very similar so we do not need to make a distinction at this moment. A second remark is that the memory footprint of QMC=Chem is particularly low. For the biggest size considered (1731 electrons), the amount of RAM needed is only 313 MiB. Finally, another important remark is that at small number of electrons the multiplicative part is dominant while this is not the case at larger sizes. Here, the change of regime is observed somewhere between 400 and 1000 electrons but its precise location depends strongly on the number of basis functions used. For example, for systems 3 and 4 corresponding to the same molecule with a different number of basis functions, the multiplicative part is still dominant for the larger basis set (β-strand with cc-pVTZ) while it is no longer true for the smaller basis set (β-strand with 6-31G*). In Figure 2, a plot of the total computational time for the Sandy Bridge core as a function of the number of electrons is presented. A standard fit of the curve with a polynomial form N^γ leads to a γ-value of about 2.5. However, as discussed above such a power is not really meaningful. From the data of Table 3, it is easy to extract the pure contribution related to the inversion and a factor very close to 3 is obtained, thus illustrating that for this linear algebra part we are in the asymptotic regime. For the multiplicative part, the pure N^2 behavior is not yet recovered and we are in an intermediate regime. Putting together these two situations leads to some intermediate scaling around 2.5.

Sparsity. In our practical algorithm, for the matrix products we have chosen to consider the **B** matrices as sparse as opposed to the **A** matrix which is considered dense. The reason for that is that considering the matrix **A** sparse would not allow us to write a stride-one inner loop. In single precision, SSE instructions executed on Intel processors can perform up to eight instructions per CPU cycle (one four-element vector ADD instruction and one four-element vector MUL instruction in parallel). Using the latest AVX instruction set available on the Sandy Bridge architecture, the width of the SIMD vector registers have been doubled and the CPU can now perform up to 16 floating point operations per cycle. A necessary condition for enabling vectorization is a stride-one access to the data. This implies that using a sparse representation of **A** would disable vectorization, and reduce the maximum number of floating operations per cycle by a factor of four using SSE (respectively, eight using AVX). If matrix **A** has more than 25% (respectively, 12.5%) nonzero elements, using a sparse representation is clearly not the best choice. This last result is a nice illustration of the idea that the efficiency of the formal mathematical algorithm depends on the core architecture.

Inversion step. Now, let us consider the inversion step which is the dominant CPU-part for the big enough systems (here, for about a thousand electrons and more). In Table 2, the performance in GFlops/s of the inversion step is presented for both processors. For comparisons, the theoretical single-core peak and single-core Linpack performance are given. For each processor the third column gives the overall performance of the code while the second column is specific to the inversion part. As seen the performance of both parts increases with the number of electrons. For largest systems, the performance represents more than 50% of the peak performance of each processor. For the largest system, the whole code has a performance of about 54% of the peak performance for the Core2 and about 61% for the Sandy Bridge. The performance is still

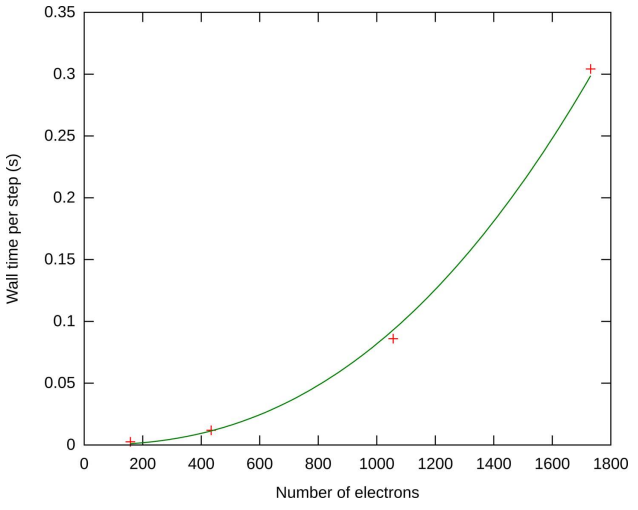


Figure 2. Single-core scaling with system size. [Color figure can be viewed in the online issue, which is available at wileyonlinelibrary.com.]

Table 4. Number of million CPU cycles needed for the computation of the values, gradients and Laplacians of the molecular orbitals using the Einspline package and using our implementation for the Core2 and the Sandy Bridge micro-architectures. The ratio QMC = Chem/Einspline is also given.

	Core2			Sandy Bridge		
	QMC = Chem	Einspline	Ratio	QMC = Chem	Einspline	Ratio
Smallest system	16.7	15.0	1.11	9.2	10.6	0.87
β-strand TZ	177.3	113.0	1.57	81.7	80.1	1.02
1ZE7	783.5	669.1	1.17	352.0	473.9	0.74
1AMB	2603.0	1797.8	1.45	1183.9	1273.5	0.93

better for the inversion part: 60.2% for the Core2 and 61.4% for the Sandy Bridge.

Determinant calculation compared to spline interpolation. Most authors use three-dimensional spline representations of the MOs to compute in constant time the values, first derivatives and Laplacians of one electron in one MO, independently of the size of the atomic basis set. This approach seems efficient at first sight, but the major drawback is that the memory required for a single processor can become rapidly prohibitive since each MO has to be precomputed on a three-dimensional grid. To overcome the large-memory problem, these authors use shared memory approaches on the computing nodes, which implies coupling between the different CPU cores. In this paragraph, we compare the wall time needed for spline interpolation or computation of the values, first derivatives and Laplacians of the wave function at all electron positions.

Version 0.9.2 of the Einspline package[30] was used as a reference to compute the interpolated values, gradients and Laplacians of 128 MOs represented on $23 \times 21 \times 29$ single precision arrays. The "multiple uniform splines" set of routines were used. To evaluate the value, gradient and Laplacian of one MO at one electron coordinate, an average of 1200 CPU cycles was measured using LIKWID[31] on the Core2 processor versus 850 CPU cycles on the Sandy Bridge processor. Even if the interpolation is done using a very small amount of data and of floating point operations, it is bound by the memory latency. Indeed, the needed data is very unlikely to be in the CPU cache and this explains why the number of cycles per matrix element is quite large. As our code uses a very small amount of memory, and as the computationally intensive routines are very well vectorized by the compiler, the computation of the matrix elements is bound by the floating point throughput of the processor.

The number of cycles needed to build the $\mathbf{C}_1 \ldots \mathbf{C}_5$ matrices is the number of cycles needed for one matrix element scaled by the number of matrix elements $N_\alpha^2 + N_\beta^2$. Table 4 shows the number of CPU cycles needed to build the full $\mathbf{C}_1 \ldots \mathbf{C}_5$ matrices for a new set of electron positions using spline interpolation or using computation. The computation includes the computation of the values, gradients and Laplacians of the AOs (matrices $\mathbf{B}_1 \ldots \mathbf{B}_5$) followed by the matrix products.

Using a rather small basis set (6-31G*), the computation of the matrices in the 158-electron system is only 10% slower than the interpolation on the Core2 architecture. Using a larger basis set (cc-pVTZ), the computation is only 57% slower.

As the frequency is higher in our Sandy Bridge processor than in our Core2 processor, we would have expected the number of cycles of one memory latency to increase, and therefore we would have expected the Einspline package to be less efficient on that specific processor. One can remark that the memory latencies have been dramatically improved from the Core2 to the Sandy Bridge architectures and the number of cycles for the interpolation decreases.

The full computation of the matrix elements benefits from the improvement in the memory accesses, but also from the enlargement of the vector registers from 128 to 256 bits. This higher vectorization considerably reduces the number of cycles needed to perform the calculation such that in the worst case (the largest basis set), the full computation of the matrix elements takes as much time as the interpolation. In all other cases, the computation is faster than the spline interpolation. Finally, let us mention that as the memory controller is directly attached to the CPU, on multisocket computing nodes the memory latencies are higher when accessing a memory module attached to another CPU (Non-uniform memory access (NUMA) architecture).

Parallelism: Implementing a Universal, Dynamic, and Fault-Tolerant Scheme

Our objective was to design a program that could take maximum advantage of heterogeneous clusters, grid environments, the petaflops platforms available now and those to come soon (exascale).

To achieve the best possible parallel speed-up on any hardware, all the parallel tasks have to be completely decoupled. Feldman et al. have shown that a naive implementation of parallelism does not scale well on commodity hardware.[32] Such bad scalings are also expected to be observed on very large-scale simulations. Therefore, we chose an implementation where each CPU core realizes a QMC run with its own population of walkers independently of all the other CPU cores. The run is divided in *blocks* over which the averages of the quantities of interest are computed. The only mandatory communications are the *one-to-all* communication of the input data and the *all-to-one* communications of the results, each result being the Monte Carlo average computed with a single-core executable. If a single-core executable is able to start as soon as the input data is available and stop at any time sending an average over all the computed Monte Carlo steps, the best possible parallel speed-up on the machine can always be obtained. This aspect is detailed in this section.

Fault-tolerance

Fault-tolerance is a critical aspect since the mean time before failure increases with the number of hardware components: using N identical computing nodes for a singe run multiplies by

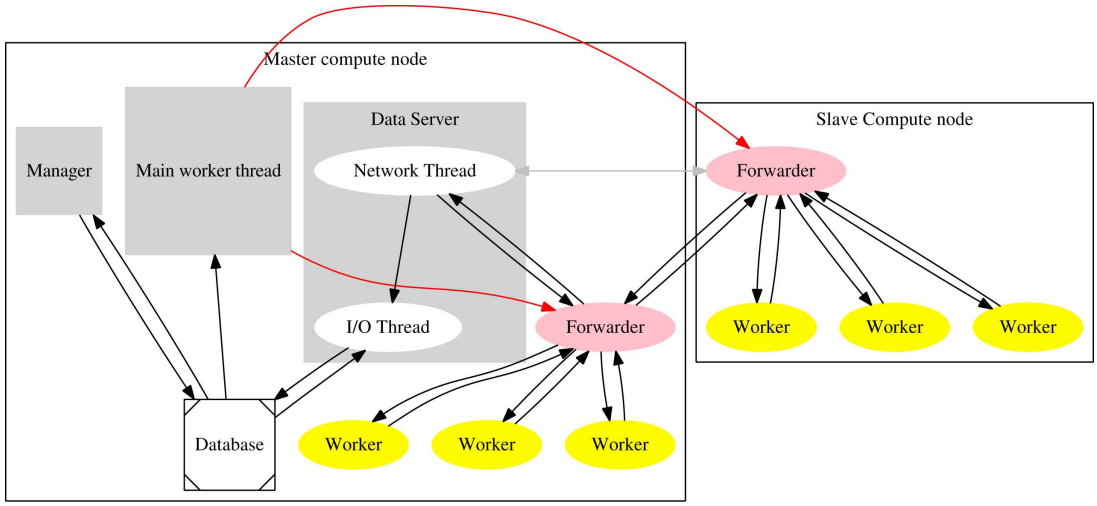


Figure 3. Overview of the QMC=Chem architecture. [Color figure can be viewed in the online issue, which is available at wileyonlinelibrary.com.]

N the probability of failure of the run. If one computing node is expected to fail once a year, a run using 365 computing nodes is not expected to last more than a day. As our goal is the use both of massive resources and commodity clusters found in laboratories, hardware failure is at the center of our software design.

The traditional choice for the implementation of parallelism is the use of the message passing interface (MPI).[33] Efficient libraries are proposed on every parallel machine, and it is probably the best choice in most situations. However, all the complex features of MPI are not needed for our QMC program, and it does not really fit our needs: in the usual MPI implementations, the whole run is killed when one parallel task is known not be able to reach the MPI_Finalize statement. This situation occurs when a parallel task is killed, often due to a system failure (I/O error, frozen computing node, hardware failure, etc). For deterministic calculations where the result of every parallel task is required, this mechanism prevents from unexpected dead locks by immediately stopping a calculation that will never end. In our implementation, as the result of the calculation of a block is a Gaussian distributed random variable, removing the result of a block from the simulation is not a problem since doing that does not introduce any bias in the final result. Therefore, if one computing node fails, the rest of the simulation should survive.

We wrote a simple Python TCP client/server application to handle parallelism. To artificially improve the bandwidth, all network transfers are compressed using the Zlib library,[34] and the results are transferred asynchronously in large packets containing a collection of small messages. Similarly, the storage of the results is executed using a nonblocking mechanism. The computationally intensive parts were written using the IRPF90 code generator,[35] to produce efficient Fortran code that is also easy to maintain. The architecture of the whole program is displayed in Figure 3.

Program interface

Our choice concerning the interaction of the user with the program was not to use the usual "input file and output file" structure. Instead, we chose to use a database containing all the input data and control parameters of the simulation, and also the results computed by different runs. A few simple scripts allow the interaction of the user with the database. This choice has several advantages:

- The input and output data are tightly linked together. It is always possible to find to which input corresponds output data.
- If an output file is needed, it can be generated on demand using different levels of verbosity.
- Graphical and web interfaces can be trivially connected to the program.
- Simple scripts can be written by the users to manipulate the computed data in a way suiting their needs.

Instead of storing the running average as the output of a run, we store all the independent block-averages in the database, and the running averages are post-processed on demand by database queries. There are multiple benefits from this choice:

- Checkpoint/restart is always available
- It is possible to compute correlations, combine different random variables, and so on, even when the QMC run is finished.
- Combining results computed on different clusters consists in simply merging the two databases, which allows automatically the use of the program on computing grids.[36]
- Multiple independent jobs running on the same cluster can read/write in the same database to communicate via the file system. This allows to gather more and more resources as they become available on a cluster or to run a massive number of tasks in a *best effort* mode.[†]

Error checking

We define the critical data of a simulation as the input data that characterizes uniquely a given simulation. For instance, the molecular coordinates, the MOs, the Jastrow factor parameters are critical data since they are fixed parameters of the wave function during a QMC run. In contrast, the number of walkers of a simulation is not critical data for a VMC run since

[†]When cluster resources are unused, a QMC job starts. When another user requests the resources, the QMC job is killed.

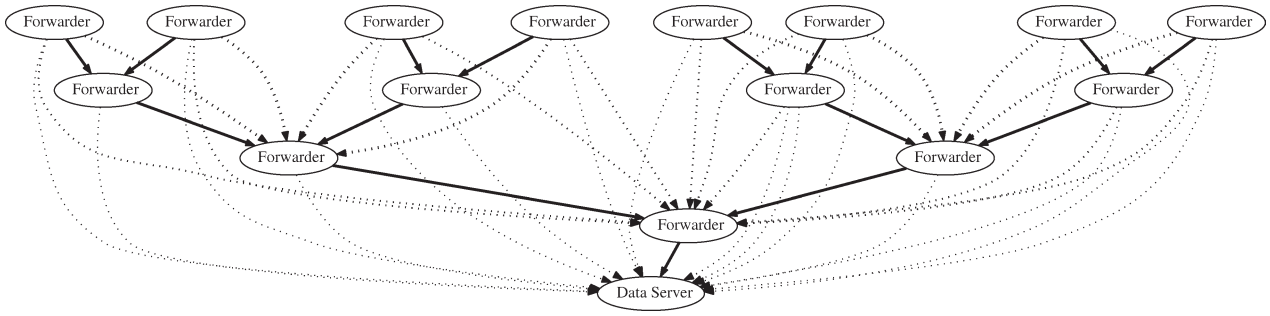


Figure 4. Connections of the forwarders with the data server.

the results of two VMC simulations with a different number of walkers can be combined together. A 32-bit cyclic redundancy code (CRC-32 key) is associated with the critical data to characterize a simulation. This key will be used to guarantee that the results obtained in one simulation will never be mixed with the results coming from another simulation and corrupt the database. It will also be used to check that the input data have been well transferred on every computing node.

Program execution

When the program starts its execution, the *manager* process runs on the master node and spawns two other processes: a *data server* and a *main worker process*.

At any time, new clients can connect to the data server to add dynamically more computational resources to a running calculation, and some running clients can be terminated without stopping the whole calculation. The manager periodically queries the database and computes the running averages using all the blocks stored in the database. It controls the running/stopping state of the workers by checking if the stopping condition is reached (based, e.g., on the wall-clock time, on the error bar of the average energy, a Unix signal, etc).

When running on super-computers, the main worker process spawns one single instance of a *forwarder* on each computing node given by the batch scheduler system using an MPI launcher. As soon as the forwarders are started the MPI launcher terminates, and each forwarder connects to the data server to retrieve the needed input data. The forwarder then starts multiple *workers* on the node with different initial walker positions.

Each worker is an instance of the single-core Fortran executable, connected to the forwarder by Unix pipes. Its behavior is the following:while (.True.)

```
{
  compute_a_block_of_data();
  send_the_results_to_the_forwarder();
}
```

Unix signals SIGTERM and SIGUSR2 are trapped to trigger the send_the_results_to_the_forwarder procedure followed by the termination of the process. Using this mechanism, any single-core executable can be stopped *immediately* without losing a single Monte Carlo step. This aspect is essential to obtain the best possible speed-up on massively parallel machines. Indeed, using the matrix product presented in the previous section makes the CPU time of a block nonconstant. Without this mechanism, the run would finish when the last CPU finishes, and the parallel efficiency would be reduced when using a very large number of CPU cores.

While the workers are computing the next block, the forwarder sends the current results to the data-server using a path going through other forwarders. The forwarders are organized in a binary tree as displayed in Figure 4: every node of the tree can send data to all its ancestors, to deal with possible failures of computing nodes. This tree-organization reduces the number of connections to the data server, and also enlarges the size of the messages by combining in a single message the results of many forwarders.

At the end of each block, the last walker positions are sent from the worker to the forwarder. The forwarder keeps a fixed-sized list of N_{kept} walkers enforcing the distribution of local energies: when a forwarder receives a set of N walkers, it appends the list of new walkers to its N_{kept} list, and sorts the $N_{\text{kept}} + N$ list by increasing local energies. A random number η is drawn to keep all list entries at indices $\lfloor \eta + i\ (N_{\text{kept}} + N)/N_{\text{kept}} \rfloor$, $i = \{1,\ldots N_{\text{kept}}\}$. After a random timeout, if the forwarder is idle, it sends its list of walkers to its parent in the binary tree which repeats the list merging process. Finally, the data server receives a list of walkers, merges it with its own list and writes it to disk when idle. This mechanism ensures that the walkers saved to disk will represent homogeneously the whole run and avoids sending all the walkers to the data server. These walkers will be used as new starting points for the next QMC run.

Using such a design the program is robust to system failures. Any computing node can fail with a minimal impact on the simulation:

- If a worker process fails, only the block being computed by this worker is lost. It does not affect the forwarder to which it is linked.
- If a forwarder fails, then only one computing node is lost thanks to the redundancy introduced in the binary tree of forwarders.
- The program execution survives short network disruption (a fixed timeout parameter). The data will arrive to the data server when the network becomes operational again.
- The disks can crash on the computing nodes: the temporary directory used on the computing nodes is a RAM-disks (/dev/shm).

- The shared file system can fail as the single-core static executable, the python scripts and input files are broadcast to the RAM-disks of the compute nodes with the MPI launcher when the run starts.
- Redundancy can be introduced on the data server by running multiple jobs using the same database. Upon a failure of a data server, only the forwarders connected to it will be lost.
- In the case of a general power failure, all the calculations can be restarted without losing what has already been stored in the database.

Finally, we have left the possibility of using different executables connected to the same forwarder. This will allow a combined use of pure CPU executables with hybrid CPU/GPU and CPU/MIC executables, to use efficiently all the available hardware. The extension to hybrid architectures will be the object of a future work.

Parallel speed-up

The benchmarks presented in this section were performed on the Curie machine (GENCI-TGCC-CEA, France). Each computing node is a dual socket Intel Xeon E5-2680: 2× (8 cores, 20 MiB shared L3-cache, 2.7 GHz) with 64 GiB of RAM. The benchmark is a DMC calculation of the β-strand system with the cc-PVTZ basis set (Table 1) using 100 walkers per core performing 300 steps in each block. Note that these blocks are very short compared to realistic simulations, where the typical number of steps would be larger than 1000 to avoid the correlation between the block averages.

Intra node. The CPU consumption of the forwarder is negligible (typically 1% of the CPU time spent in the single-core executables). The speed-up with respect to the number of sockets is ideal. Indeed, the single-core binaries do not communicate between each other, and as the memory consumption per core is very low, each socket never uses memory modules attached to another socket. When multiple cores on the same socket are used, we observed a slow-down for each core due to the sharing of the L3-cache and memory modules. Running simultaneously 16 instances of the single-core binaries on our benchmark machine yields an increase of 10.7% of the wall-clock time compared to running only one instance. For a 16-core run, we obtained a 14.4 × speed-up (the Turbo feature of the processors was deactivated for this benchmark).

Inter node. In this section, the wall-clock time is measured from the very beginning to the very end of the program execution using the standard *GNU time* tool. Hence, the wall-clock time includes the initialization and finalization steps.

The initialization step includes

- Input file consistency checking
- Creating a gzipped tar file containing the input files (wave function parameters, simulation parameters, a pool of initial walkers), the Python scripts and static single-core executable needed for the program execution on the slave nodes
- MPI initialization
- Broadcasting the gzipped tar file via MPI to all the slave nodes

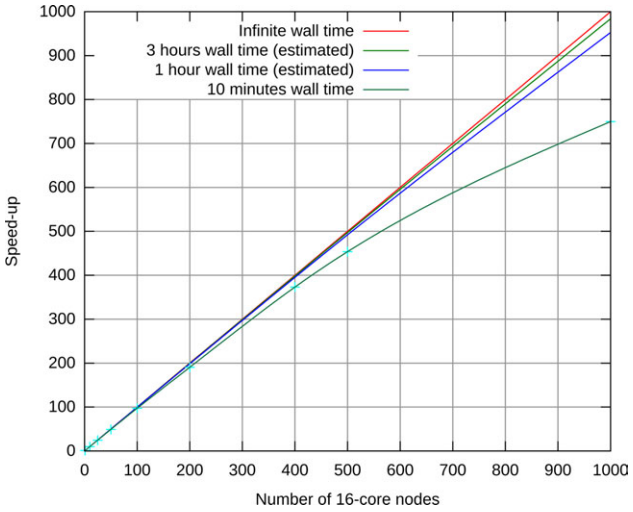


Figure 5. Parallel speed-up of QMC=Chem with respect to 16-core compute nodes (reference is one 16-core node).

- Extracting the tar file to the RAM-disk of the slave nodes
- Starting the forwarders
- Starting the single-core instances.

Note that as no synchronization is needed between the nodes, the computation starts as soon as possible on each node.

The finalization step occurs as follows. When the data server receives a termination signal, it sends a termination signal to all the forwarders that are leaves in the tree of forwarders. When a forwarder receives such a signal, it sends a SIGTERM signal to all the single-core binary instances of the computing node which terminate after sending to the forwarder the averages computed over the truncated block. Then, the forwarder sends this data to its parent in the binary tree with a termination signal and sends a message to the data server to inform it that it is terminated. This termination step walks recursively through the tree. When all forwarders are done, the data server exits. Note that if a failure happened on a node during the run, the data server never receives the message corresponding to a termination of the corresponding forwarder. Therefore, when the data server receives the termination signal coming from the forwarders tree, if the data server is still running after a given timeout it exits.

We prepared a 10-min run for this section to compute the parallel speed-up curve as a function of the number of 16-core nodes given in Figure 5. The data corresponding to this curve are given in Table 5. The reference for the speed-up is the one-node run. The speed-up for N nodes is computed as:

$$\frac{t_{\mathrm{CPU}}(N)/t_{\mathrm{Wall}}(N)}{t_{\mathrm{CPU}}(1)/t_{\mathrm{Wall}}(1)}. \tag{18}$$

The initialization time was 9 s for the single node run and 22 s for the 1000 nodes run. The finalization time was 13 s for the single node run and 100 s for the 1000 nodes run.

Apart from the initialization and finalization steps (which obviously do not depend on the total execution time), the

Table 5. Data relative to the scaling curve (Fig. 5).

Number of 16-core nodes	10 min			60 min (estimated)			180 min (estimated) speed-up
	CPU (s)	Wall (s)	Speed-up	CPU (s)	Wall (s)	Speed-up	
1	9627	625	1.0	57,147	3625	1.00	1.00
				57,332	*3629*	*1.00*	
10	95,721	627	9.9	570,921	3627	9.98	9.99
25	239,628	629	24.7	1427,628	3629	24.95	24.98
50	477,295	631	49.1	2,853,295	3631	49.85	49.95
100	952,388	636	97.2	5704,388	3636	99.52	99.84
				5,708,422	*3638*	*99.32*	
200	1,869,182	637	190.5	11,373,182	3637	198.36	199.45
400	3,725,538	648	373.3	22,733,538	3648	395.30	398.42
500	4,479,367	641	453.7	28,239,367	3641	491.98	497.31
1000	8,233,981	713	749.7	55,753,981	3713	952.50	983.86

CPU time is the cumulated CPU time spent only in the Fortran executables, Wall time is the measured wall-clock time, including initialization and finalization steps (serial).
The 10-min run were measured, and the longer runs are estimated from the 10-min run data.
Two checks were measured for the 60-min runs with 1 and 100 nodes (in italics).

parallel speed-up is ideal. This allowed us to estimate the speed-ups, we would have obtained for a 1-h run and for a 3-h run. For instance, to estimate the 1-h run we added 50 min to the wall-clock time and 50 min $\times$ 16 $\times$ number of nodes $\times$ 0.99 to the CPU time. The 99% factor takes account of the CPU consumption of the forwarder for communications. Our simple model was checked by performing a 1-h run on one node and a 1-h run on 100 nodes. An excellent agreement with the prediction was found: a 99.5 $\times$ speed-up was predicted for 100 nodes and a 99.3 $\times$ speed-up was measured.

Finally, a production run was made using 76,800 cores of Curie (4800 nodes) on the β-strand molecule with a cc-pVTZ basis set via 12 runs of 40 nodes, and a sustained performance of 960 TFlops/s was measured. All the details and scientific results of this application will be presented elsewhere (Caffarel and Scemama, Unpublished).

Summary

Let us summarize the main results of this work. First, to enhance the computational efficiency of the expensive innermost floating-point operations (calculation and multiplication of matrices), we propose to take advantage of the highly localized character of the *atomic* Gaussian basis functions, in contrast with the standard approaches using localized MOs. The advantages of relying on atomic localization have been illustrated on a series of molecules of increasing sizes (number of electrons ranging from 158 to 1731). In this article, it is emphasized that the notion of scaling of the computational cost as a function of the system size has to be considered with caution. Here, although the algorithm proposed is formally quadratic it displays a small enough prefactor to become very efficient in the range of number of electrons considered. Furthermore, our implementation of the linear-algebra computational part has allowed to enlighten a fundamental issue rarely discussed, namely the importance of taking into consideration the close links between algorithmic structure and CPU core architecture. Using efficient techniques and optimization tools for enhancing single-core performance, this point has been illustrated in various situations. Remark that this aspect is particularly important: as the parallel speed-up is very good, the gain in execution time obtained for the single-core executable will also be effective in the total parallel simulation.

In our implementation, we have chosen to minimize the memory footprint. This choice is justified first by the fact that today the amount of memory per CPU core tends to decrease and second by the fact that small memory footprints allow in general a more efficient usage of caches. In this spirit, we propose not to use 3D-spline representation of the MOs as usually done. We have shown that this can be realized without increasing the CPU cost. For our largest system with 1731 electrons, only 313 MiB of memory per core was required. As a consequence, the key limiting factor of our code is only the available CPU time and neither the memory nor disk space requirements, nor the network performance. Let us reemphasize that this feature is well aligned with the current trends in computer architecture for large HPC systems.

Finally, let us conclude by the fact that there is no fundamental reason why the implementation of such a QMC simulation environment which has been validated at petaflops level could not be extended to exascale.

Acknowledgments

This work was possible thanks to the generous computational support from CALMIP (Université de Toulouse), under the allocation 2011-0510, GENCI under the allocation GEN1738, CCRT (CEA), and PRACE under the allocation RA0824. The authors would also like to thank Bull, GENCI and CEA for their help in this project.

Keywords: quantum Monte Carlo · petascale · parallel speedup · single-core optimization · large systems

How to cite this article: A. Scemama, M. Caffarel, E. Oseret, W. Jalby, *J. Comput. Chem.* **2013,** *34,* 938–951. DOI: 10.1002/jcc.23216

[1] S. Gandolfi, F. Pederiva, S. Fantoni, K. Schmidt, *Phys. Rev. Lett.* **2007**, *98,* 102503, available at: http://link.aps.org/doi/10.1103/PhysRevLett.98.102503

[2] W. Foulkes, L. Mitas, R. Needs, G. Rajagopal, *Rev. Mod. Phys.* **2001**, *73,* 33; available at: http://link.aps.org/doi/10.1103/RevModPhys.73.33
[3] M. Suzuki, Quantum Monte Carlo Methods in Condensed Matter Physics; World Scientific: Singapore, **1994**.
[4] D. Ceperley, *Rev. Mod. Phys.* **1995**, *67,* 279.
[5] D. Coker, R. Watts, *J. Phys. Chem.* **1987**, *91,* 2513.
[6] M. Caffarel, P. Claverie, C. Mijoule, J. Andzelm, D. Salahub, *J. Chem. Phys.* **1989**, *90,* 990.
[7] B. Hammond, W. Lester, Jr., P. Reynolds, Monte Carlo Methods in Ab Initio Quantum Chemistry, Vol. *1 of Lecture and Course Notes in Chemistry; World Scientific: Singapore,* **1994**; ISBN 978-981-02-0322-1, World Scientific Lecture and Course Notes in Chemistry, Vol. *1.*
[8] M. Caffarel, In Encyclopedia of Applied and Computational Mathematics; B. Engquist, Ed.; Springer, **2012**; available at: http://qmcchem.ups-tlse.fr/files/caffarel/qmc.pdf. Accessed December 21, 2012.
[9] K. P. Esler, J. Kim, D. M. Ceperley, W. Purwanto, E. J. Walter, H. Krakauer, S. Zhang, P. R. C. Kent, R. G. Hennig, C. Umrigar, M. Bajdich, J. Kolorenč, L. Mitas, A. Srinivasan, *J. Phys.: Conf. Series* **2008**, *125,* 1.
[10] J. Kim, K. Esler, J. McMinis, D. M. Ceperley, In Proceedings of the 2010 Scientific Discovery through Advanced Computing (SciDac) Conference, Tennessee, 11–15 July, **2010**, Oak Ridge National Laboratory.
[11] M. Gillan, M. Towler, D. Alf, In Psi-k Highlight of the Month, February, **2011**.
[12] Quantum Monte Carlo for Chemistry@Toulouse, available at: http://qmcchem.ups-tlse.fr. Accessed December 21, 2012.
[13] C. Umrigar, M. Nightingale, K. Runge, *J. Chem. Phys.* **1993**, *99,* 2865.
[14] J. Krogel, D. Ceperley, In Advances in Quantum Monte Carlo; W. L. J. S. Tanaka, S. Rothstein, Eds.; Vol. *1094 of ACS Symposium Series,* **2012i**; pp. 13–26.
[15] M. C. Buonaura, S. Sorella, *Phys. Rev. B* **1998**, *57,* 11446.
[16] R. Assaraf, M. Caffarel, A. Khelif, *Phys. Rev. E* **2000**, *61,* 4566.
[17] A. Williamson, R. Hood, J. Grossman, *Phys. Rev. Lett.* **2001**, *87,* 246406.
[18] D. Alfè, M. J. Gillan, *J. Phys.: Condens. Matter* **2004**, *16,* 305.
[19] A. Aspuru-Guzik, R. S. N-Ferrer, B. Austin, W. Lester, Jr., *J. Comput. Chem.* **2005**, *26,* 708.
[20] F. A. Reboredo, A. J. Williamson, *Phys. Rev. B* **2005**, *71,* 121105(R).
[21] J. Kussmann, H. Riede, C. Ochsenfeld, *Phys. Rev. B* **2007**, *75,* 165107.
[22] J. Kussmann, C. Ochsenfeld, *J. Chem. Phys.* **2008**, *128,* 134104.
[23] S. Nagarajan, J. Rajadas, E. P. Malar, *J. Struct. Biol.* **2010**, *170,* 439, ISSN 1047-8477, available at: http://www.sciencedirect.com/science/article/pii/S104784771000064X
[24] S. Boys, *Rev. Mod. Phys.* **1960**, *32,* 296.
[25] J. Pipek, P. Mezey, *J. Chem. Phys.* **1989**, *90,* 4916.
[26] F. Aquilante, T. B. Pedersen, A. Sanchez de Mers, H. Koch, *J. Chem, Phys.* **2006**, *125,* 174101, available at: http://link.aip.org/link/?JCP/125/174101/1
[27] N. Zingirian, M. Maresca, In High-Performance Computing and Networking; P. Sloot, M. Bubak, A. Hoekstra, B. Hertzberger, Eds.; Springer: Berlin/Heidelberg, **1999**; Vol. *1593 of Lecture Notes in Computer Science, pp.* 633–642, ISBN 978-3-540-65821-4, 10.1007/BFb0100624.
[28] L. Djoudi, D. Barthou, P. Carribault, C. Lemuet, J.- T. Acquaviva, W. Jalby, In Workshop on EPIC Architectures and Compiler Technology, San Jose, CA, **2005**.
[29] S. Koliai, S. Zuckerman, E. Oseret, M. Ivascot, T. Moseley, D. Quang, W. Jalby, In LCPC, **2009**, pp. 111–125.
[30] http://einspline.sourceforge.net. Accessed December 21, 2012.
[31] J. Treibig, G. Hager, G. Wellein, In 39th International Conference on Parallel Processing Workshops (ICPPW), San Diego, CA, **2010**; pp. 207–216, ISSN 1530–2016.
[32] M. T. Feldman, C. Julian, D. R. Cummings, R. Kent IV, R. P. Muller, W. A. Goddard III, *J. Comput. Chem.* **2008**, *29,* 8.
[33] W. Gropp, E. Lusk, N. Doss, A. Skjellum, *Parallel Comput.* **1996**, *22,* 789.
[34] http://zlib.net
[35] A. Scemama, ArXiv e-prints [cs.SE], 0909.5012v1, **2009**, arXiv:0909.5012v1, available at: http://arxiv.org/abs/0909.5012. Accessed December 21, 2012.
[36] A. Monari, A. Scemama, M. Caffarel, In Remote Instrumentation for eScience and Related Aspects; F. Davoli, M. Lawenda, N. Meyer, R. Pugliese, J. Wglarz, S. Zappatore, Eds.; Springer: New York, **2012**; pp. 195–207, ISBN 978-1-4614-0508-5, available at: http://dx.doi.org/10.1007/978-1-4614-0508-5_13. Accessed December 21, 2012.

Received: 28 September 2012
Revised: 27 November 2012
Accepted: 1 December 2012
Published online on 3 January 2013

IRPF90: a programming environment for high performance computing

Anthony Scemama
Laboratoire de Chimie et Physique Quantiques,
CNRS-UMR 5626,
IRSAMC Université Paul Sabatier,
118 route de Narbonne
31062 Toulouse Cedex, France
(Dated: October 22, 2009)

Abstract

IRPF90 is a Fortran programming environment which helps the development of large Fortran codes. In Fortran programs, the programmer has to focus on the order of the instructions: before using a variable, the programmer has to be sure that it has already been computed in all possible situations. For large codes, it is common source of error. In IRPF90 most of the order of instructions is handled by the pre-processor, and an automatic mechanism guarantees that every entity is built before being used. This mechanism relies on the {needs/needed by} relations between the entities, which are built automatically. Codes written with IRPF90 execute often faster than Fortran programs, are faster to write and easier to maintain.

I. INTRODUCTION

The most popular programming languages in high performance computing (HPC) are those which produce fast executables (Fortran and C for instance). Large programs written in these languages are difficult to maintain and these languages are in constant evolution to facilitate the development of large codes. For example, the C++ language[1] was proposed as an improvement of the C language by introducing classes and other features of object-oriented programming. In this paper, we propose a Fortran pre-processor with a very limited number of keywords, which facilitates the development of large programs and the re-usability of the code without affecting the efficiency.

In the imperative programming paradigm, a computation is a ordered list of commands that change the state of the program. At the lowest level, the machine code is imperative: the commands are the machine code instructions and the state of the program is represented by to the content of the memory. At a higher level, the Fortran language is an imperative language. Each statement of a Fortran program modifies the state of the memory.

In the functional programming paradigm, a computation is the evaluation of a function. This function, to be evaluated, may need to evaluate other functions. The state of the program is not known by the programmer, and the memory management is handled by the compiler.

Imperative languages are easy to understand by machines, while functional languages are easy to understand by human beings. Hence, code written in an imperative language can be made extremely efficient, and this is the main reason why Fortran and C are so popular in the field of High Performance Computing (HPC).

However, codes written in imperative languages usually become excessively complicated to maintain and to debug. In a large code, it is often very difficult for the programmer to have a clear image of the state of the program at a given position of the code, especially when side-effects in a procedure modifiy memory locations which are used in other procedures.

In this paper, we present a tool called "Implicit Reference to Parameters with Fortran 90" (IRPF90). It is a Fortran pre-processor which facilitates the development of large simulation codes, by allowing the programmer to focus on *what* is being computed, instead of *how* it is computed. This last sentence often describes the difference between the functional and the imperative paradigms[2]. From a practical point of view, IRPF90 is a program written in the Python[3] language. It produces Fortran source files from IRPF90 source files. IRPF90 source files are Fortran source files with a limited number of additional statements. To explain how to use the IRPF90 tool, we will write a simple molecular dynamics program as a tutorial.

II. TUTORIAL: A MOLECULAR DYNAMICS PROGRAM

A. Imperative and functional implementation of the potential

We first choose to implement the Lennard-Jones potential[4] to compute the interaction of pairs of atoms:

$$V(r) = 4\epsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right] \tag{1}$$

```
 1  program potential_with_imperative_style
 2   implicit none
 3   double precision :: sigma_lj, epsilon_lj
 4   double precision :: interatomic_distance
 5   double precision :: sigma_over_r
 6   double precision :: V_lj
 7   print *, 'Sigma?'
 8   read(*,*) sigma_lj
 9   print *, 'Epsilon?'
10   read(*,*) epsilon_lj
11   print *, 'Interatomic Distance?'
12   read(*,*) interatomic_distance
13   sigma_over_r = sigma_lj/interatomic_distance
14   V_lj = 4.d0 * epsilon_lj * ( sigma_over_r**12 &
15    -  sigma_over_r**6 )
16   print *, 'Lennard-Jones potential:'
17   print *, V_lj
18  end program
```

FIG. 1: Imperative implementation of the Lennard-Jones potential.

where r is the atom-atom distance, ϵ is the depth of the potential well and σ is the value of r for which the potential energy is zero. ϵ and σ are the parameters of the force field.

Using an imperative style, one would obtain the program given in figure 1. One can clearly see the sequence of statements in this program: first read the data, then compute the value of the potential.

This program can be re-written using a functional style, as shown in figure 2. In the functional form of the program, the sequence of operations does not appear as clearly as in the imperative example. Moreover, the order of execution of the commands now depends on the choice of the compiler: the function `sigma_over_r` and the function `epsilon_lj` are both called on line 12-13, and the order of execution may differ from one compiler to the other.

The program was written in such a way that the functions have no arguments. The reason for this choice is that the references to the entities which are needed to calculate a function appear inside the function, and not outside of the function. Therefore, the code is simpler to understand for a programmer who never read this particular code, and it can be easily represented as a production tree (figure 3, above). This tree exhibits the relation {needs/needed by} between the entities of interest: the entity `V_lj` needs the entities `sigma_over_r` and `epsilon_lj` to be produced, and `sigma_over_r` needs `sigma_lj` and `interatomic_distance`.

In the imperative version of the code (figure 1), the production tree has to be known by the programmer so he can place the instructions in the proper order. For simple programs it is not a problem, but for large codes the production tree can be so large that the programmer is likely to make wrong assumptions in the dependencies between the entities. This complexifies the structure of the code by the introduction of many different methods to compute the same quantity, and the performance of the code can be reduced due to the computation of entities which are not needed.

In the functional version (figure 2), the production tree does not need to be known by the programmer. It exists implicitely through the function calls, and the evaluation of the main function is realized by exploring the tree with a depth-first algorithm. A large advantage of the functional style is that there can only be one way to calculate the value of an entity:

```
 1  program potential_with_functional_style
 2   double precision :: V_lj
 3   print *, V_lj()
 4  end program
 5
 6  double precision function V_lj()
 7   double precision :: sigma_lj
 8   double precision :: epsilon_lj
 9   double precision :: interatomic_distance
10   double precision :: sigma_over_r
11   V_lj = 4.d0 * epsilon_lj() * &
12     ( sigma_over_r()**12 - sigma_over_r()**6 )
13  end function
14
15  double precision function epsilon_lj()
16   print *, 'Epsilon?'
17   read(*,*) epsilon_lj
18  end function
19
20  double precision function sigma_lj ()
21   print *, 'Sigma?'
22   read(*,*) sigma_lj
23  end function
24
25  double precision function sigma_over_r()
26   double precision :: sigma_lj
27   double precision :: interatomic_distance
28   sigma_over_r = sigma_lj()/interatomic_distance()
29  end function
30
31  double precision function interatomic_distance()
32   print *, 'Interatomic Distance?'
33   read(*,*) interatomic_distance
34  end function
```

FIG. 2: Functional implementation of the Lennard-Jones potential.

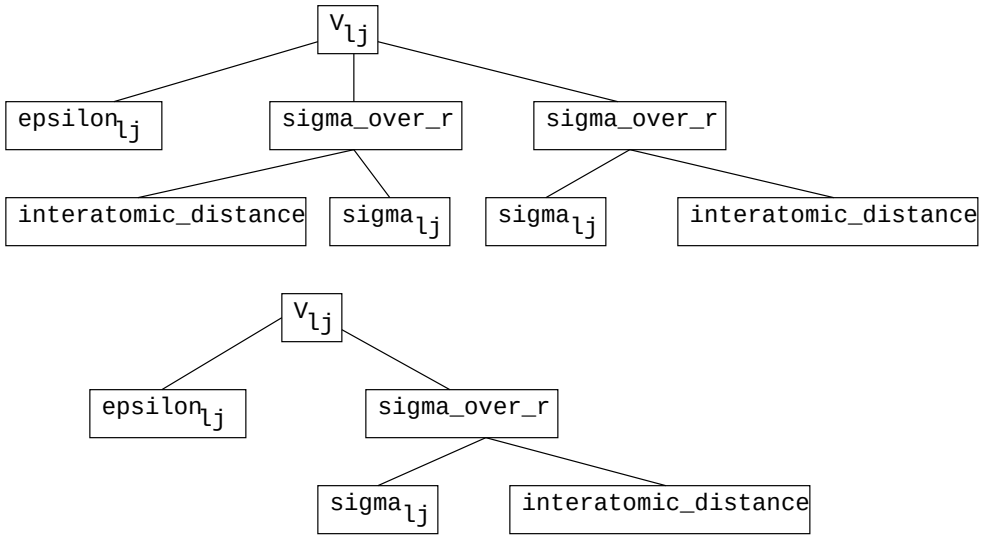


FIG. 3: The production tree of `V_lj`. Above, the tree produced by the program of figure 2. Below, the tree obtained if only one call to `sigma_over_r` is made.

```
 1  double precision function sigma_over_r()
 2   double precision       :: sigma_lj
 3   double precision       :: interatomic_distance
 4   double precision, save :: last_result
 5   integer, save          :: first_time_here
 6   if (first_time_here.eq.0) then
 7     last_result = sigma_lj()/interatomic_distance()
 8     first_time_here = 1
 9   endif
10   sigma_over_r = last_result
11  end function
```

FIG. 4: Memoized `sigma_over_r function`

calling the corresponding function. Therefore, the readability of the code is improved for a programmer who is not familiar with the program. Moreover, as soon as an entity is needed, it is calculated and valid. Writing programs in this way reduces considerably the risk to use un-initialized variables, or variables that are supposed to have a given value but which have been modified by a side-effect.

With the functional example, every time a quantity is needed it is computed, even if it has already been built before. If the functions are pure (with no side-effects), one can implement memoization[5, 6] to reduce the computational cost: the last value of the function is saved, and if the function is called again with the same arguments the last result is returned instead of computing it again. In the present example we chose to write functions with no arguments, so memoization is trivial to implement (figure 4). If we consider that the leaves of the production tree are constant, memoization can be applied to all the functions. The production tree of `V_lj` can now be simplified, as shown in figure 3, below.

B. Presentation of the IRPF90 statements

IRPF90 is a Fortran pre-processor: it generates Fortran code from source files which contain keywords specific to the IRPF90 program. The keywords understood by IRPF90 pre-processor are briefly presented. They will be examplified in the next subsections for the molecular dynamics example.

`BEGIN_PROVIDER ... END_PROVIDER`
Delimitates the definition of a provider (sections II C and II D).

`BEGIN_DOC ... END_DOC`
Delimitates the documentation of the current provider (section II C).

`BEGIN_SHELL ... END_SHELL`
Delimitates an embedded script (section II E).

`ASSERT`
Expresses an assertion (section II C).

`TOUCH`
Expresses the modification of the value of an entity by a side-effect (section II F).

`FREE`
Invalidates an entity and free the associated memory. (section **??**).

`IRP_READ / IRP_WRITE`
Reads/Writes the content of the production tree to/from disk (section II G).

```
 1  program lennard_jones_dynamics
 2   print *, V_lj
 3  end program
 4
 5  BEGIN_PROVIDER [ double precision, V_lj ]
 6    implicit none
 7    BEGIN_DOC
 8  ! Lennard Jones potential energy.
 9    END_DOC
10    double precision :: sigma_over_r
11    sigma_over_r = sigma_lj / interatomic_distance
12    V_lj = 4.d0 * epsilon_lj * ( sigma_over_r**12 &
13       - sigma_over_r**6 )
14  END_PROVIDER
15
16   BEGIN_PROVIDER [ double precision, epsilon_lj ]
17  &BEGIN_PROVIDER [ double precision, sigma_lj ]
18    BEGIN_DOC
19  ! Parameters of the Lennard-Jones potential
20    END_DOC
21    print *, 'Epsilon?'
22    read(*,*) epsilon_lj
23    ASSERT (epsilon_lj > 0.)
24    print *, 'Sigma?'
25    read(*,*) sigma_lj
26    ASSERT (sigma_lj > 0.)
27  END_PROVIDER
28
29  BEGIN_PROVIDER[double precision,interatomic_distance]
30    BEGIN_DOC
31  ! Distance between the atoms
32    END_DOC
33    print *, 'Inter-atomic distance?'
34    read (*,*) interatomic_distance
35    ASSERT (interatomic_distance >= 0.)
36  END_PROVIDER
```

FIG. 5: IRPF90 implementation of the Lennard-Jones potential.

`IRP_IF ... IRP_ELSE ... IRP_ENDIF`
Delimitates blocks for conditional compilation (section II G).
`PROVIDE`
Explicit call to the provider of an entity (section II G).

C. Implementation of the potential using IRPF90

In the IRPF90 environment, the entities of interest are the result of memoized functions with no arguments. This representation of the data allows its organization in a production tree, which is built and handled by the IRPF90 pre-processor. The previous program may be written again using the IRPF90 environment, as shown in figure 5.

The program shown in figure 5 is very similar to the functional program of figure 2. The difference is that the entities of interest are not functions anymore, but variables. The variable corresponding to an entity is provided by calling a providing procedure (or provider), defined between the keywords `BEGIN_PROVIDER ... END_PROVIDER`. In the IRPF90 environment, a provider can provide several entities (as shown with the parameters of the potential), although it is preferable to have providers that provide only one entity.

When an entity has been built, it is tagged as built. Hence, the next call to the provider will return the last computed value, and will not build the value again. This explains why in the IRPF90 environment the parameters of the force field are asked only once to the user.

The `ASSERT` keyword was introduced to allow the user to place assertions[9] in the code. An assertion specifies certain general properties of a value. It is expressed as a logical expression which is supposed to be always true. If it is not, the program is wrong. Assertions in the code provide run-time checks which can dramatically reduce the time spent finding bugs: if an assertion is not verified, the program stops with a message telling the user which assertion caused the program to fail.

The `BEGIN_DOC ... END_DOC` blocks contain the documentation of the provided entities. The descriptions are encapsulated inside these blocks in order to facilitate the generation of technical documentation. For each entity a "man page" is created, which contains the {needs/needed by} dependencies of the entity and the description given in the `BEGIN_DOC ... END_DOC` block. This documentation can be accessed by using the `irpman` command followed by the name of the entity.

The IRPF90 environment was created to simplify the work of the scientific programmer. A lot of time is spent creating Makefiles, which describe the dependencies between the source files for the compilation. As the IRPF90 tool "knows" the production tree, it can build automatically the Makefiles of programs, without any interaction with the user. When the user starts a project, he runs the command `irpf90 -init` in an empty directory. A standard Makefile is created, with the gfortran compiler[10] as a default. Then, the user starts to write IRPF90 files which contain providers, subroutines, functions and main programs in files characterized by the `.irp.f` suffix. Running `make` calls `irpf90`, and a correct Makefile is automatically produced and used to compile the code.

D. Providing arrays

Now the basics of IRPF90 are known to the reader, we can show how simple it is to write a molecular dynamics program. As we will compute the interaction of several atoms, we will change the previous program such that we produce an array of potential energies per atom. We first need to introduce the quantity `Natoms` which contains the number of atoms. Figure 6 shows the code which defines the geometrical parameters of the system. Figure 7 shows the providers corresponding to the potential energy V per atom i, where it is chosen equal to the Lennard-Jones potential energy:

$$V_i = V_i^{LJ} = \sum_{j \neq i}^{N_{\text{atoms}}} 4\epsilon \left[\left(\frac{\sigma}{||\mathbf{r}_{ij}||} \right)^{12} - \left(\frac{\sigma}{||\mathbf{r}_{ij}||} \right)^{6} \right] \tag{2}$$

Figure 8 shows the providers corresponding to the kinetic energy T per atom i:

$$T_i = \frac{1}{2} m_i ||\mathbf{v}_i||^2 \tag{3}$$

where m_i is the mass and $\mathbf{v}_i$ is the velocity vector of atom i. The velocity vector is chosen to be initialized zero.

The dimensions of arrays are given in the definition of the provider. If an entity, defines the dimension of an array, the provider of the dimensioning entity will be called before allocating the array. This guarantees that the array will always be allocated with the proper

```
 1  BEGIN_PROVIDER [ integer, Natoms ]
 2   BEGIN_DOC
 3  ! Number of atoms
 4   END_DOC
 5   print *, 'Number of atoms?'
 6   read(*,*) Natoms
 7   ASSERT (Natoms > 0)
 8  END_PROVIDER
 9
10   BEGIN_PROVIDER [ double precision, coord, (3,Natoms) ]
11  &BEGIN_PROVIDER [ double precision, mass , (Natoms)   ]
12   implicit none
13   BEGIN_DOC
14  ! Atomic data, input in atomic units.
15   END_DOC
16   integer :: i,j
17   print *, 'For each atom: x, y, z, mass?'
18   do i=1,Natoms
19    read(*,*) (coord(j,i), j=1,3), mass(i)
20    ASSERT (mass(i) > 0.)
21   enddo
22  END_PROVIDER
23
24  BEGIN_PROVIDER[double precision,distance,(Natoms,Natoms)]
25    implicit none
26    BEGIN_DOC
27  ! distance  : Distance matrix of the atoms
28    END_DOC
29    integer :: i,j,k
30    do i=1,Natoms
31     do j=1,Natoms
32      distance(j,i) = 0.
33      do k=1,3
34       distance(j,i) = distance(j,i) + &
35         (coord(k,i)-coord(k,j))**2
36      enddo
37      distance(j,i) = sqrt(distance(j,i))
38     enddo
39    enddo
40  END_PROVIDER
```

FIG. 6: Code defining the geometrical parameters of the system

size. In IRPF90, the memory allocation of an array entity is not written by the user, but by the pre-processor.

Memory can be explicitely freed using the keyword `FREE`. For example, de-allocating the array `velocity` would be done using `FREE velocity`. If the memory of an entity is freed, the entity is tagged as "not built", and it will be allocated and built again the next time it is needed.

E. Embedding scripts

The IRPF90 environment allows the programmer to write scripts inside his code. The scripting language that will interpret the script is given in brackets. The result of the shell script will be inserted in the file, and then will be interpreted by the Fortran pre-processor. Such scripts can be used to write templates, or to write in the code some information that has to be retrieved at compilation. For example, the date when the code was compiled can

```
BEGIN_PROVIDER [ double precision, V, (Natoms) ]
  BEGIN_DOC
! Potential energy.
  END_DOC
  integer :: i
  do i=1,Natoms
   V(i) = V_lj(i)
  enddo
END_PROVIDER

BEGIN_PROVIDER [ double precision, V_lj, (Natoms) ]
  implicit none
  BEGIN_DOC
! Lennard Jones potential energy.
  END_DOC
  integer :: i,j
  double precision :: sigma_over_r
  do i=1,Natoms
   V_lj(i) = 0.
   do j=1,Natoms
    if ( i /= j ) then
      ASSERT (distance(j,i) > 0.)
      sigma_over_r = sigma_lj / distance(j,i)
      V_lj(i) = V_lj(i) + sigma_over_r**12 &
       - sigma_over_r**6
    endif
   enddo
   V_lj(i) = 4.d0 * epsilon_lj * V_lj(i)
  enddo
END_PROVIDER

 BEGIN_PROVIDER [ double precision, epsilon_lj ]
&BEGIN_PROVIDER [ double precision, sigma_lj ]
  BEGIN_DOC
! Parameters of the Lennard-Jones potential
  END_DOC
  print *, 'Epsilon?'
  read(*,*) epsilon_lj
  ASSERT (epsilon_lj > 0.)
  print *, 'Sigma?'
  read(*,*) sigma_lj
  ASSERT (sigma_lj > 0.)
END_PROVIDER
```

FIG. 7: Definition of the potential.

be inserted in the source code using the example given in figure 9.

In our molecular dynamics program, the total kinetic energy `E_kin` is the sum over all the elements of the kinetic energy vector `T`:

$$E_{\text{kin}} = \sum_{i=1}^{N_{\text{atoms}}} T_i \tag{4}$$

Similarly, the potential energy `E_pot` is the sum of all the potential energies per atom.

$$E_{\text{pot}} = \sum_{i=1}^{N_{\text{atoms}}} V_i \tag{5}$$

The code to build `E_kin` and `E_pot` is very close: only the names of the variables change, and it is convenient to write the code using a unique template for both quantities, as shown in

```
BEGIN_PROVIDER [ double precision, T, (Natoms) ]
 BEGIN_DOC
! Kinetic energy per atom
 END_DOC
 integer :: i
 do i=1,Natoms
  T(i) = 0.5d0 * mass(i) * velocity2(i)
 enddo
END_PROVIDER

BEGIN_PROVIDER[double precision,velocity2,(Natoms)]
 BEGIN_DOC
! Square of the norm of the velocity per atom
 END_DOC
 integer :: i, k
 do i=1,Natoms
  velocity2(i) = 0.d0
  do k=1,3
   velocity2(i) = velocity2(i) + velocity(k,i)**2
  enddo
 enddo
END_PROVIDER

BEGIN_PROVIDER[double precision,velocity,(3,Natoms)]
 BEGIN_DOC
! Velocity vector per atom
 END_DOC
 integer :: i, k
 do i=1,Natoms
  do k=1,3
   velocity(k,i) = 0.d0
  enddo
 enddo
END_PROVIDER
```

FIG. 8: Definition of the kinetic energy.

```
program print_the_date
 BEGIN_SHELL [ /bin/sh ]
  echo print *, \'Compiled by $USER on `date`\'
 END_SHELL
end program
```

FIG. 9: Embedded shell script which gets the date of compilation.

figure 10. In this way, adding a new property which is the sum over all the atomic properties can done be done in only one line of code: adding the triplet (Property, Documentation, Atomic Property) to the list of entities at line 15.

F. Changing the value of an entity by a controlled side-effect

Many computer simulation programs contain iterative processes. In an iterative process, the same function has to be calculated at each step, but with different arguments. In our IRPF90 environment, at every iteration the production tree is the same, but the values of some entities change. To keep the program correct, if the value of one entity is changed it has to be tagged as "built" with its new value, and all the entities which depend on this

```
BEGIN_SHELL [ /usr/bin/python ]
template = """
BEGIN_PROVIDER [ double precision, %(entity)s ]
 BEGIN_DOC
! %(doc)s
 END_DOC
 integer :: i
 %(entity)s = 0.
 do i=1,Natoms
  %(entity)s = %(entity)s+%(e_array)s(i)
 enddo
END_PROVIDER
"""
entities = [ ("E_pot", "Potential Energy", "V"),
             ("E_kin", "Kinetic Energy", "T") ]
for e in entities:
  dictionary = { "entity": e[0],
    "doc": e[1],
    "e_array": e[2]}
  print template%dictionary
END_SHELL
```

FIG. 10: Providers of the Lennard-Jones potential energy and the kinetic energy using a template.

entity (directly or indirectly) need to be tagged as "not built". These last entities will need to be re-computed during the new iteration. This mechanism is achieved automatically by the IRPF90 pre-processor using the keyword `TOUCH`. The side-effect modifying the value of the entity is controlled, and the program will stay consistent with the change everywhere in the rest of the code.

In our program, we are now able to compute the kinetic and potential energy of the system. The next step is now to implement the dynamics. We choose to use the velocity Verlet algorithm[11]:

$$\mathbf{r}^{n+1} = \mathbf{r}^n + \mathbf{v}^n \Delta t + \mathbf{a}^n \frac{\Delta t^2}{2} \tag{6}$$

$$\mathbf{v}^{n+1} = \mathbf{v}^n + \frac{1}{2}(\mathbf{a}^n + \mathbf{a}^{n+1})\Delta t \tag{7}$$

where $\mathbf{r}^n$ and $\mathbf{v}^n$ are respectively the position and velocity vectors at step n, Δt is the time step and the acceleration vector $\mathbf{a}$ is defined as

$$\mathbf{a} = \sum_{i=1}^{N_{\text{atoms}}} -\frac{1}{m_i} \nabla_i E_{\text{pot}} \tag{8}$$

The velocity Verlet algorithm is written in a subroutine `verlet`, and the gradient of the potential energy ∇E_{pot} can be computed by finite difference (figure 11).

Computing a component i of the numerical gradient of E_{pot} can be decomposed in six steps:

1. Change the component i of the coordinate $\mathbf{r}_i \longrightarrow (\mathbf{r}_i + \delta)$
2. Compute the value of E_{pot}
3. Change the coordinate $(\mathbf{r}_i + \delta) \longrightarrow (\mathbf{r}_i - \delta)$

```
 1  BEGIN_PROVIDER  [ double precision, dstep ]
 2   BEGIN_DOC
 3  ! Finite difference step
 4   END_DOC
 5   dstep = 1.d-4
 6  END_PROVIDER
 7
 8  BEGIN_PROVIDER[double precision,V_grad_numeric,(3,Natoms)]
 9   implicit none
10   BEGIN_DOC
11  ! Numerical gradient of the potential
12   END_DOC
13   integer :: i, k
14   do i=1,Natoms
15    do k=1,3
16     coord(k,i) = coord(k,i) + dstep
17     TOUCH coord
18     V_grad_numeric(k,i) = E_pot
19     coord(k,i) = coord(k,i) - 2.d0*dstep
20     TOUCH coord
21     V_grad_numeric(k,i) = &
22       ( V_grad_numeric(k,i)-E_pot )/(2.d0*dstep)
23     coord(k,i) = coord(k,i) + dstep
24    enddo
25   enddo
26   TOUCH coord
27  END_PROVIDER
28
29  BEGIN_PROVIDER [ double precision, V_grad, (3,Natoms) ]
30   BEGIN_DOC
31  ! Gradient of the potential
32   END_DOC
33   integer :: i,k
34   do i=1,Natoms
35    do k=1,3
36     V_grad(k,i) = V_grad_numeric(k,i)
37    enddo
38   enddo
39  END_PROVIDER
```

FIG. 11: Provider of the gradient of the potential.

4. Compute the value of $E_{\rm pot}$

5. Compute the component of the gradient using the two last values of $E_{\rm pot}$

6. Re-set $(\mathbf{r}_i - \delta) \longrightarrow \mathbf{r}_i$

The provider of `V_grad_numeric` follows these steps: in the internal loop, the array `coord` is changed (line 16). Touching it (line 17) invalidates automatically `E_pot`, since it depends indirectly on `coord`. As the value of `E_pot` is needed in line 18 and not valid, it is re-computed between line 17 and line 18. The value of `E_pot` which is affected to `V_grad_numeric(k,i)` is the value of the potential energy, consistent with the current set of atomic coordinates. Then, the coordinates are changed again (line 19), and the program is informed of this change at line 20. When the value of `E_pot` is used again at line 22, it is consistent with the last change of coordinates. At line 23 the coordinates are changed again, but no touch statement follows. The reason for this choice is efficiency, since two cases are possible for the next instruction: if we are at the last iteration of the loop, we exit the main loop and

line 26 is executed. Otherwise, the next instruction will be line 16. Touching `coord` is not necessary between line 23 and line 16 since no other entity is used.

The important point is that the programmer doesn't have to know *how* `E_pot` depends on `coord`. He only has to apply a simple rule which states that when the value of an entity A is modified, it has to be touched before any other entity B is used. If B depends on A, it will be re-computed, otherwise it will not, and the code will always be correct. Using this method to compute a numerical gradient allows a programmer who is not familiar with the code to compute the gradient of any entity A with respect to any other quantity B, without even knowing if A depends on B. If A does not depend on B, the gradient will automatically be zero. In the programs dealing with optimization problems, it is a real advantage: a short script can be written to build automatically all the possible numerical derivatives, involving all the entities of the program, as given in figure 12.

The velocity Verlet algorithm can be implemented (figure 13) as follows:

1. Compute the new value of the coordinates
2. Compute the component of the velocities which depends on the old set of coordinates
3. Touch the coordinates and the velocities
4. Increment the velocities by their component which depends on the new set of coordinates
5. Touch the velocities

G. Other Features

As IRPF90 is designed for HPC, conditional compilation is an essential requirement. Indeed, it is often used for activating and deactivating blocks of code defining the behavior of the program under a parallel environment. This is achieved by the `IRP_IF...IRP_ELSE...IRP_ENDIF` constructs. In figure 14, the checkpointing block is activated by running `irpf90 -DCHECKPOINT`. If the `-D` option is not present, the other block is activated.

The current state of the production tree can written to disk using the command `IRP_WRITE` as in figure 14. For each entity in the subtrees of `E_pot` and `E_kin`, a file is created with the name of the entity which contains the value of the entity. The subtree can be loaded again later using the `IRP_READ` statement. This functionality is particularly useful for adding quickly a checkpointing feature to an existing program.

The `PROVIDE` keyword was added to assign imperatively a needs/needed by relation between two entities. This keyword can be used to associate the value of an entity to an iteration number in an iterative process, or to help the preprocessor to produce more efficient code.

A last convenient feature was added: the declarations of the local variables do not need anymore to be located before the first executable statement. The local variables can now be declared anywhere inside the providers, subroutines and functions. The IRPF90 preprocessor will put them at the beginning of the subroutines or functions for the programmer. It allows the user to declare the variables where the reader needs to know to what they correspond.

```
BEGIN_SHELL [ /usr/bin/python ]
# Read the names of the entities and their dimensions
dims = {}
import os
for filename in os.listdir('.'):
  if filename.endswith('.irp.f'):
    file = open(filename,'r')
    for line in file:
     if "%" not in line:
      if line.strip().lower().startswith('begin_provider'):
        buffer = line.split(',',2)
        name = buffer[1].split(']')[0].strip()
        if len(buffer) == 2:
          dims[name] = []
        else:
          dims[name] = buffer[2]
          for c in "()] \n":
            dims[name] = dims[name].replace(c,"")
          dims[name] = dims[name].split(",")
    file.close()
# The template to use for the code generation
template = """
BEGIN_PROVIDER[double precision, grad_%(var1)s_%(var2)s %(dims2)s]
 BEGIN_DOC
! Gradient of %(var1)s with respect to %(var2)s
 END_DOC
 integer :: %(all_i)s
 double precision :: two_dstep
 two_dstep = dstep + dstep
%(do)s
   %(var2)s %(indice)s =  %(var2)s %(indice)s + dstep
   TOUCH %(var2)s
   grad_%(var1)s_%(var2)s %(indice)s = %(var1)s
   %(var2)s %(indice)s =  %(var2)s %(indice)s - two_dstep
   TOUCH %(var2)s
   grad_%(var1)s_%(var2)s %(indice)s =  &
      (grad_%(var1)s_%(var2)s %(indice)s - %(var1)s)/two_dstep
   %(var2)s %(indice)s =  %(var2)s %(indice)s + dstep
%(enddo)s
 TOUCH %(var2)s
END_PROVIDER
"""
# Generate all possibilities of d(v1)/d(v2), with v1 scalar
for v1 in dims.keys():
 if dims[v1] == []:
  for v2 in dims.keys():
   if v2 != v1:
    do = ""
    enddo = ""
    if dims[v2] == []:
      dims2 = ""
      all_i = "i"
      indice = ""
    else:
      dims2 = ', ('+','.join(dims[v2])+')'
      all_i = ','.join([ "i"+str(k) for k in range(len(dims[v2]))])
      indice = "("
      for k,d in enumerate(dims[v2]):
        i = "i"+str(k)
        do = " do "+i+" = 1,"+d+"\n"+do
        enddo += " enddo\n"
        indice += i+","
      indice = indice[:-1]+")"
    dictionary = {"var1"  : v1,
     "var2"  : v2, "dims2" : dims2,
     "all_i" : all_i, "do"    : do,
     "indice": indice, "enddo" : enddo}
    print template%dictionary
END_SHELL
```


FIG. 12: Automatic generation of all possible gradients of scalar entities with respect to all other entities.

```
BEGIN_PROVIDER [ integer, Nsteps ]
 BEGIN_DOC
! Number of steps for the dynamics
 END_DOC
 print *, 'Nsteps?'
 read(*,*) Nsteps
 ASSERT (Nsteps > 0)
END_PROVIDER

 BEGIN_PROVIDER [ double precision, tstep ]
&BEGIN_PROVIDER [ double precision, tstep2 ]
 BEGIN_DOC
! Time step for the dynamics
 END_DOC
 print *, 'Time step?'
 read(*,*) tstep
 ASSERT (tstep > 0.)
 tstep2 = tstep*tstep
END_PROVIDER

BEGIN_PROVIDER[double precision,acceleration,(3,Natoms)]
 implicit none
 BEGIN_DOC
! Acceleration = - grad(V)/m
 END_DOC
 integer :: i, k
 do i=1,Natoms
  do k=1,3
   acceleration(k,i) = - V_grad(k,i)/mass(i)
  enddo
 enddo
END_PROVIDER

subroutine verlet
 implicit none
 integer :: is, i, k
 do is=1,Nsteps
  do i=1,Natoms
   do k=1,3
    coord(k,i) = coord(k,i) + tstep*velocity(k,i) + &
     0.5*tstep2*acceleration(k,i)
    velocity(k,i) = velocity(k,i) + 0.5*tstep* &
     acceleration(k,i)
   enddo
  enddo
  TOUCH coord velocity
  do i=1,Natoms
   do k=1,3
    velocity(k,i) = velocity(k,i) + 0.5*tstep* &
     acceleration(k,i)
   enddo
  enddo
  TOUCH velocity
  call print_data(is)
 enddo
end subroutine
```

FIG. 13: The velocity Verlet algorithm.

```
1  program dynamics
2
3    call verlet
4
5  IRP_IF CHECKPOINT
6
7    print *, 'Checkpoint'
8    IRP_WRITE E_pot
9    IRP_WRITE E_kin
10
11 IRP_ELSE
12
13   print *, 'No checkpoint'
14
15 IRP_ENDIF
16
17 end
```

FIG. 14: The main program.

III. EFFICIENCY OF THE GENERATED CODE

In the laboratory, we are currently re-writing a quantum Monte Carlo (QMC) program, named QMC=Chem, with the IRPF90 tool. The same computation was realized with the old code (usual Fortran code), and the new code (IRPF90 code). Both codes were compiled with the Intel Fortran compiler version 11.1 using the same options. A benchmark was realized on an Intel Xeon 5140 processor.

The IRPF90 code is faster than the old code by a factor of 1.60: the CPU time of the IRPF90 executable is 62% of the CPU time of the old code. This time reduction is mainly due to the avoidance of computing quantities that are already computed. The total number of processor instructions is therefore reduced.

The average number of instructions per processor cycle is 1.47 for the old code, and 1.81 for the IRPF90 code. This application shows that even if the un-necessary computations were removed from the old code, the code produced by IRPF90 would still be more efficient. The reason is that in IRPF90, the programmer is guided to write efficient code: the providers are small subroutines that manipulate a very limited number of memory locations. This coding style improves the temporal locality of the code[12] and thus minimizes the number of cache misses.

The conclusion of this real-size application is that the overhead due to the management of the production tree is negligible compared to the efficiency gained by avoiding to compute many times the same quantity, and by helping the Fortran compiler to produce optimized code.

IV. SUMMARY

The IRPF90 environment is proposed for writing programs with reduced complexity. This technique for writing programs, called "Implicit Reference to Parameters" (IRP),[7] is conform to the recommendations of the "Open Structure Interfaceable Programming Environment" (OSIPE)[8]:

- Open: Unambiguous identification and access to any entity anywhere in the program

- Interfaceable: Easy addition of any new feature to an existing code
- Structured: The additions will have no effect on the program logic

The programming paradigm uses some ideas of functional programming and thus clarifies the correspondance between the mathematical formulas and the code. Therefore, scientists do not need to be experts in programming to write clear, reusable and efficient code, as shown with the simple molecular dynamics code presented in this paper.

The consequences of the locality of the code are multiple:

- the code is efficient since the temporal locality is increased,
- the overlap of pieces of code written simultaneously by multiple developers is reduced.
- regression testing[13] can be achieved by writing, for each entity, a program which tests that the entity is built correctly.

Finally, let us mention that the IRPF90 pre-processor generates Fortran 90 which is fully compatible with standard subroutines and functions. Therefore the produced Fortran code can be compiled on any architecture, and the usual HPC libraries (BLAS[14], LAPACK[15], MPI[16],...) can be used.

The IRPF90 program can be downloaded on http://irpf90.sourceforge.net

Acknowledgments

The author would like to acknowledge F. Colonna (CNRS, Paris) for teaching him the IRP method, and long discussions around this subject. The author also would like to thank P. Reinhardt (Université Pierre et Marie Curie, Paris) for testing and enjoying the IRPF90 tool, and F. Spiegelman (Université Paul Sabatier, Toulouse) for discussions about the molecular dynamics code.

---

[1] Stroustrup B. *The C++ Programming Language* Ed: Addison-Wesley Pub Co; 3rd edition (2000).
[2] Hudak P. *ACM Comput. Surv.* **21(3)**, 359 (1989).
[3] http://www.python.org/
[4] Lennard-Jones J. E., *Proceedings of the Physical Society* **43**, 461 (1931).
[5] Michie D. *Nature* **218** 19 (1968).
[6] Hughes R.J.M. "Lazy memo functions" in: G. Goos and J. Hartmanis, eds., Proc. Conf: on Functional Programming and Computer Architecture, Nancy, France, September 1985, Springer Lecture Note Series, Vol. 201 (Springer, Berlin, 1985).
[7] http://galileo.lct.jussieu.fr/ frames/mediawiki/index.php/IRP_Programming_Presentation
[8] Colonna F., Jolly L.-H., Poirier R. A., Ángyán J. G., and Jansen G. *Comp. Phys. Comm.* **81(3)**, 293 (1994).
[9] Hoare C.A.R., *Commun. ACM*, **12(10)**, 576 (1969).
[10] http://gcc.gnu.org/fortran/
[11] Swope W. C., Andersen H. C., Berens P. H., and Wilson K. R. *J. Chem. Phys.* **76**, 637 (1982).

[12] Denning P. J. *Commun. ACM* **48(7)**, 19 (2005).

[13] Agrawal H., Horgan J. R., Krauser, E.W., London, S., Incremental regression testing. in: Proceedings of the IEEE Conference on Software Maintenance, 348 (1993).

[14] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, R. C. Whaley, *ACM Trans. Math. Soft.* **28(2)**, 135 (2002).

[15] Anderson E., Bai Z., Bischof C., Blackford S., Demmel J., Dongarra J., Du Croz J., Greenbaum A., Hammarling S., McKenney A. and Sorensen D. *LAPACK Users' Guide*, Ed: Society for Industrial and Applied Mathematics, Philadelphia, PA, (1999).

[16] Gropp W., Lusk E., Doss N. and A. Skjellum, *Parallel Computing* **22(6)**, 789 (1996).