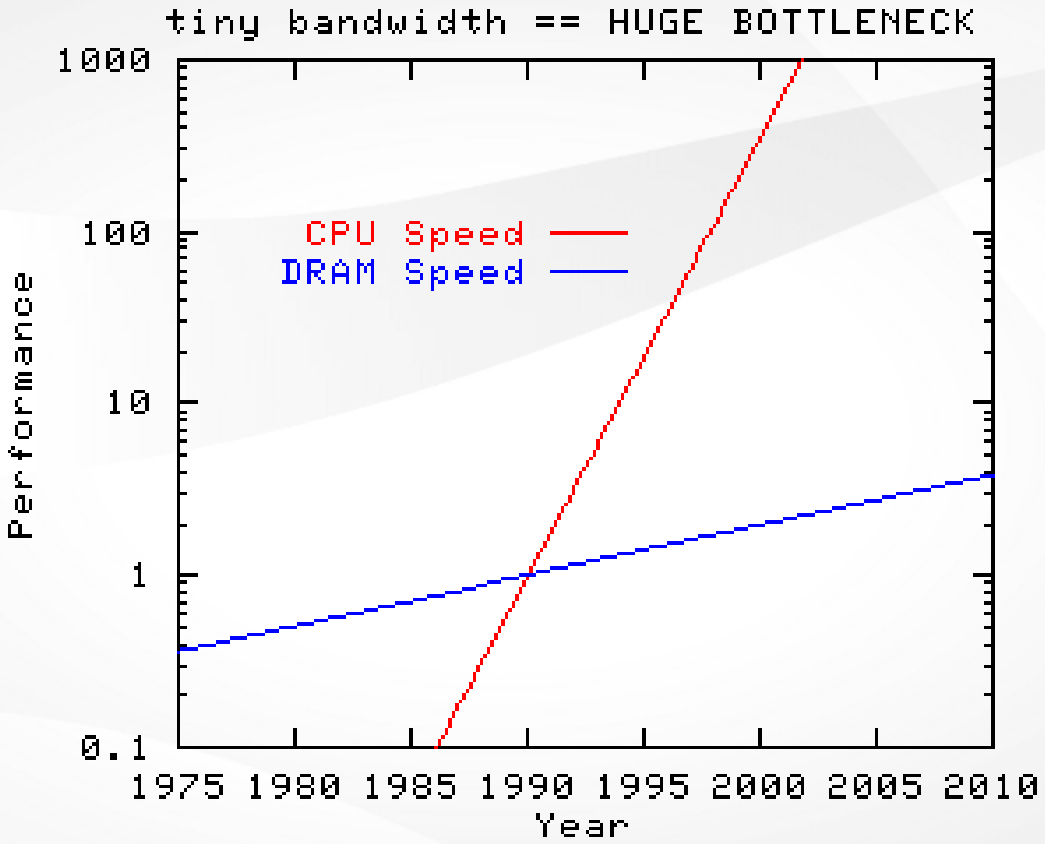
Chimie quantique et parallélisme massif

Anthony Scemama[1] <scemama@irsamc.ups-tlse.fr>
Emmanuel Giner[2], Michel Caffarel[1]

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

Barrière de la mémoire (Memory Wall)



Stream benchmark : http://www.cs.virginia.edu/stream

Latence

Temps de transfert d'une seule donnée entre deux points

Bande passante

Quantité de données qui passent par un point par unité de temps.

Latence

Améliorations de la latence et de la bande passante sur 20 ans:

Type	Latence	Bande passante
Disque	8x	143x
RAM	4x	120x
Ethernet	16x	1000x
CPU	21x	2250x

- Mémoires hierarchiques pour masquer les latences (caches)
- Les accès aléatoires sont de plus en plus coûteux

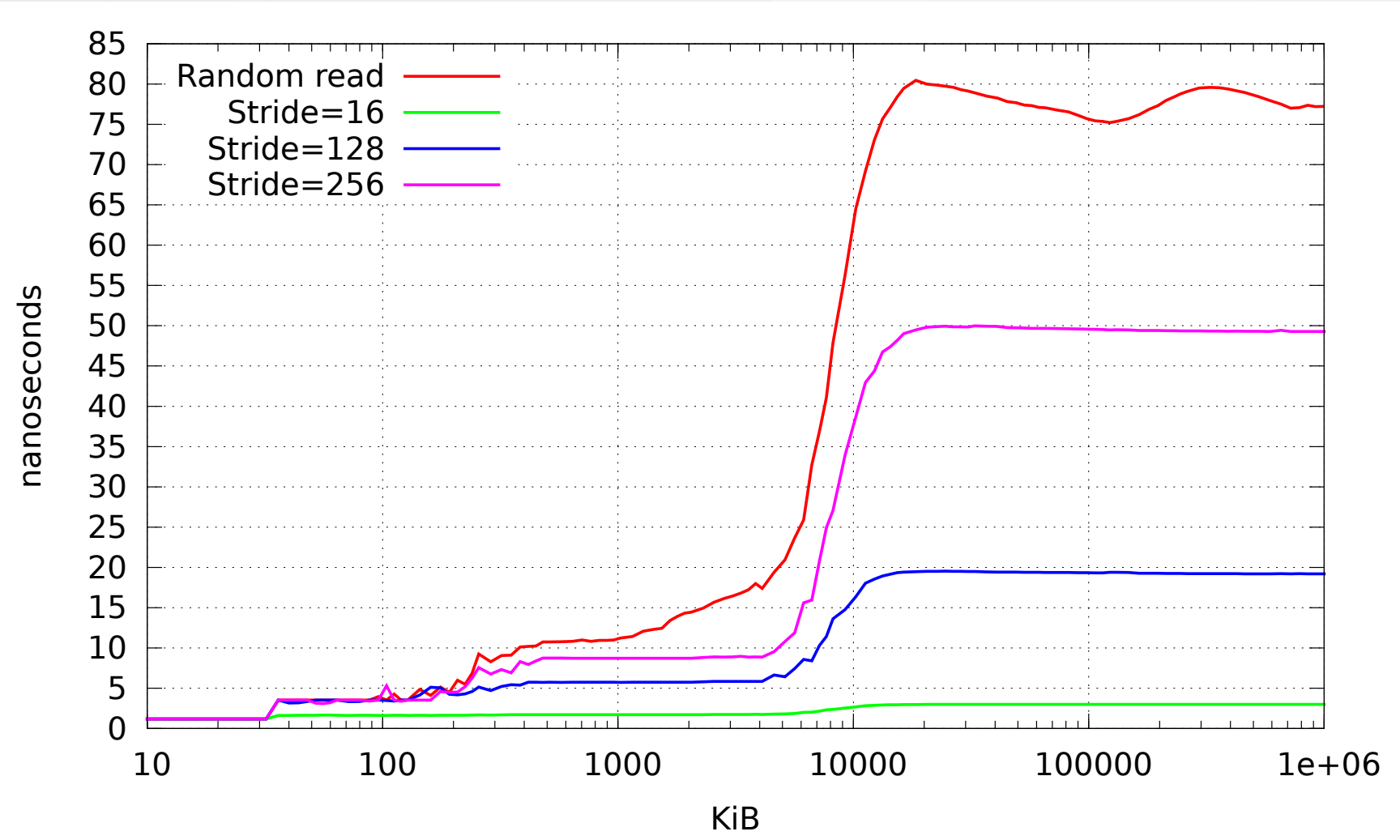
Situation en 2011 (Sandy Bridge CPU)

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.


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

- Accès aléatoire à la RAM très coûteux : ~ 300 cycles CPU : ~ 10 000 flops (Peak SP sur Haswell)
- Accès régulier avec un stride < 4096 KiB (1 page mémoire) déclenche les prefetchers hardware pour réduire la latence

D'autres nombres :

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

Bande passante

Intel® Xeon® Processor E5-2690 v3 (30M Cache, 2.60 GHz):

Deux FMA vectoriels par cycle

a = a + b*c, vecteurs de 4 flottants en double precision

Nécessite pour `a(i) = a(i) + b(i)*c(i)`

3 loads et 1 store par FMA, soit 256 octets par cycle (16 octets par flop)

Puissance crête

2.6 GHz x (2x2x4 flops) x 12 coeurs = **499.2 GFlops/s Peak** (double precision)

Bande passante nécessaire pour `a(i) = a(i) + b(i)*c(i)`

499.2 Gflops/s x 16 o/flops ~ **8 To/s**

Bande passante mémoire max

4 canaux x 2133 MHz x 8 octets = **68.2 Go/s**

La bande passante mémoire est 117x trop faible !

Pour atteindre la crête, il faut faire *au moins* 117 flops par élément chargé *en streaming* de la RAM.

Conclusions

1. La performance crête d'une machine n'est pas du tout représentative du temps de restitution des applications scientifiques (DGEMM : "seulement" ~85% d'efficacité)
2. Réduire le nombre de flops ne conduit pas forcément à l'accélération d'un programme. Ex: Algorithmes "linear scaling" sont linéaires en stockage et en calcul, donc probablement memory-bound (117x...).

Pour avoir un programme efficace il faut:

- Réutiliser au maximum ce qui peut être dans les caches
- Minimiser les accès aléatoires à la RAM : avoir des accès réguliers
- Minimiser les accès au réseau (communications bloquantes)
- Oublier les I/O sur disque
- Utiliser des bibliothèques optimisées dès que possible (MKL, PETSc, MUMPS, etc)

Chimie quantique moléculaire

1. Optimisation des orbitales moléculaires (SCF) : HF, DFT, SCC-DFTB
   - Optimisation non-linéaire
   - Diagonalisations successives de matrices construites à partir d'intégrales atomiques
   - Orbitales atomiques sont locales -> Intégrales atomiques très creuses
   - Réplication des intégrales : parallélisation distribuée facile (ScaLapack)
   - Schémas itératifs (SCF), donc barrières de synchronisation
   - Si les intégrales ne tiennent pas en mémoire : direct SCF
   - Schémas creux : remplacement des diagonalisations par des produits de matrices (gradient conjugué)

2. Approches perturbatives (MP2, MP4, RS2, ...)
   - Approches en base de MOs : transformation à 4 indices (partielle : $oo \rightarrow vv$)
   - Approches en base d'AOs : beaucoup de comm. inter-noeuds
   - Schémas approchés (RI, Laplace transform, Local MP2)
   - Pas de barrière de synchronisation
   - $E = E_0 + \sum_{iikl} c_{ijkl} \frac{\langle ij|kl\rangle^2}{\Delta E}$

3. Approches variationnelles post-HF (CI,CC,CAS,MR-CI,MR-CC,FCI)
   - Transformation à 4 indices coûteuse et difficilement parallélisable (communication inter-noeuds importante)
   - Nécessitent (sauf CAS) la transformation à 4 indices complète ($oo \rightarrow vv$ et $vv \rightarrow vv$)
   - Schémas itératifs (Davidson), donc beaucoup de barrières de synchronisation

Transformation à 4 indices

1. $(pq|rs)$: Toujours très creux $\mathcal{O}(N^2)$
2. $(iq|rs) = \sum_p c_p^i (pq|rs)$: un peu creux $\mathcal{O}(N^3)$
3. $(ij|rs) = \sum_q c_q^j (iq|rs)$: plein $\mathcal{O}(N^4)$
4. $(ij|ks) = \sum_r c_r^k (ij|rs)$: plein, ou un peu creux si MO localisées
5. $(ij|kl) = \sum_s c_s^l (ij|ks)$: plein, ou assez creux si MO localisées

- $\mathcal{O}(N^4)$ data et $\mathcal{O}(N^5)$ flops.
- Chaque CPU a besoin de tout à un moment (soit toutes les AOs, soit toutes les MOs)
- OK en SMP, mais beaucoup de communications en distribué

Adressage des intégrales

$\mathcal{O}(N^4)$ intégrales : on ne veut stocker que les non-nulles.

Adressage $i \leftrightarrow (pqrs)$ difficile.

1. Table de hachage : insertion en $\mathcal{O}(1)$, recherche en $\mathcal{O}(1)$.
2. Arbre binaire de recherche : insertion $\mathcal{O}(\log(N))$, recherche $\mathcal{O}(\log(N))$.

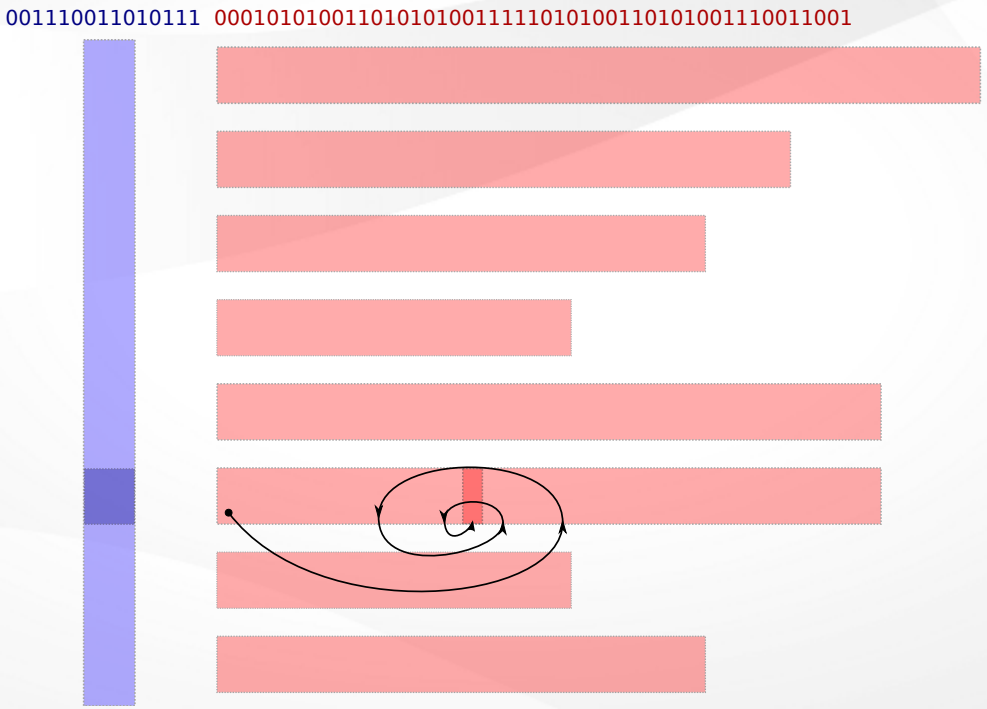
1. Temps pas constant et long en pratique : uniquement des accès aléatoires à la RAM, donc des latences gigantesques. Rehash nécessaires.

1,2. Plusieurs insertions ne peuvent pas être effectuées en parallèle en SMP.

Implémentation dans *Quantum Package*: Tableau de listes

- $(pqrs) \rightarrow i$ donne un entier unique sur 64 bits
- 15 premiers bits : accès direct au premier élément d'une liste (tableau de pointeurs)
- Recherche de l'entier sur les bits restants par dichotomie ($\mathcal{O}(\log(n))$)

- Recherche par dichotomie très rapide (prefetch dans les caches)
- Les 2^{15} listes sont triées par indices croissants : Radix-sort en $\mathcal{O}(n)$
- Tris des listes en parallèle
- Insertions simultanées dans plusieurs listes possibles



Interactions de configurations

Règles de Slater

- Transforment des intégrales à *3N* dimensions en sommes d'intégrales en 3 ou 6 dimensions.
- Condition nécessaire : Orbitales orthonormales

Si 2 déterminants diffèrent par une spin-orbitale:

- $\langle D|\mathcal{O}_1|D_i^j\rangle = \langle i|\mathcal{O}_1|j\rangle$
- $\langle D|\mathcal{O}_2|D_i^j\rangle = \sum_{k\in D}\langle ik|\mathcal{O}_2|jk\rangle - \langle ik|\mathcal{O}_2|kj\rangle$

Si 2 déterminants diffèrent par 2 spin-orbitales:

- $\langle D|\mathcal{O}_1|D_{ik}^{jl}\rangle = 0$
- $\langle D|\mathcal{O}_2|D_{ik}^{jl}\rangle = \langle ik|\mathcal{O}_2|jl\rangle - \langle ik|\mathcal{O}_2|lj\rangle$

- Dans la base des déterminants, $\mathcal{H}$ est très creux et symétrique
- Chaque élément extra-diagonal de $\mathcal{H}$ est soit 0, soit une somme d'intégrales mono- et bi-électroniques

Calcul de l'énergie CI

- Algorithme naïf : construction de la matrice $\mathcal{H}$ et diagonalisation
- Algorithme moins naïf : construction de la matrice $\mathcal{H}$ en format creux et diagonalisation
- CISD : nb de déterminants ~ nb d'intégrales. On ne pourra pas stocker $\mathcal{H}$ en RAM, même en format creux.
- $\mathcal{H}$ est diagonal-dominant $\rightarrow$ diagonalisation itérative de Davidson.
- Problème : au-delà de CISD, le nb de déterminants est plus grand que le nb d'intégrales. On ne pourra pas non plus stocker les vecteurs.
- Donc : Ecriture des vecteurs sur disque.

Algorithmes *determinant-driven*

```
do j=1,N_determinants
  do i=1,j
    H(i,j) = SlaterRules( D(i), 'H', D(j) )
  end do
end do
```

- Algorithme naturel
- Lecture en continu des déterminants
- Écriture en continu de `H`
- Lectures aléatoires des intégrales
- Très creux, donc `SlaterRules( D(i), 'H', D(j) ) = 0` presque toujours
- Inefficace tel quel (sauf Giner *et al.*)

Algorithmes *integral-driven*

- Chaque élément extra-diagonal non-nul de $\mathcal{H}$ est une somme d'intégrales
- On parcourt la liste des intégrales
- Pour chaque intégrale on regarde à quels `H(p,q)` elle contribue

```
for n in range(1,N_integrals):
    i, j, k, l = indices[n]
    for p,q in get_pq(i,j,k,l):
        H[p][q] += c[p][q] * integral[n]
```

- Lecture en continu des intégrales
- Ecritures aléatoires dans `H`
- Nécessite une fonction d'adressage des déterminants (`get_pq`)
- Conséquence de la fonction d'adressage : espaces de déterminants sélectionnés *a priori* : CISD, CISDT, CISDTQ, CAS, Full-CI
- Difficulté : scaling de ces espaces avec la taille du problème

Conclusion sur les algorthmes déterministes

- Communications all-to-all à cause de la transformation à 4 indices
- Gros besoins en RAM
- Éventuel besoin en disque
- Patterns d'accès irréguliers à la RAM
- Efficacité faible (loin de la performance crête)
- Méthodes itératives donc beaucoup de barrières de synchronisation
- Problème difficile sur les machines actuelles

Algorithmes stochastiques

QMC

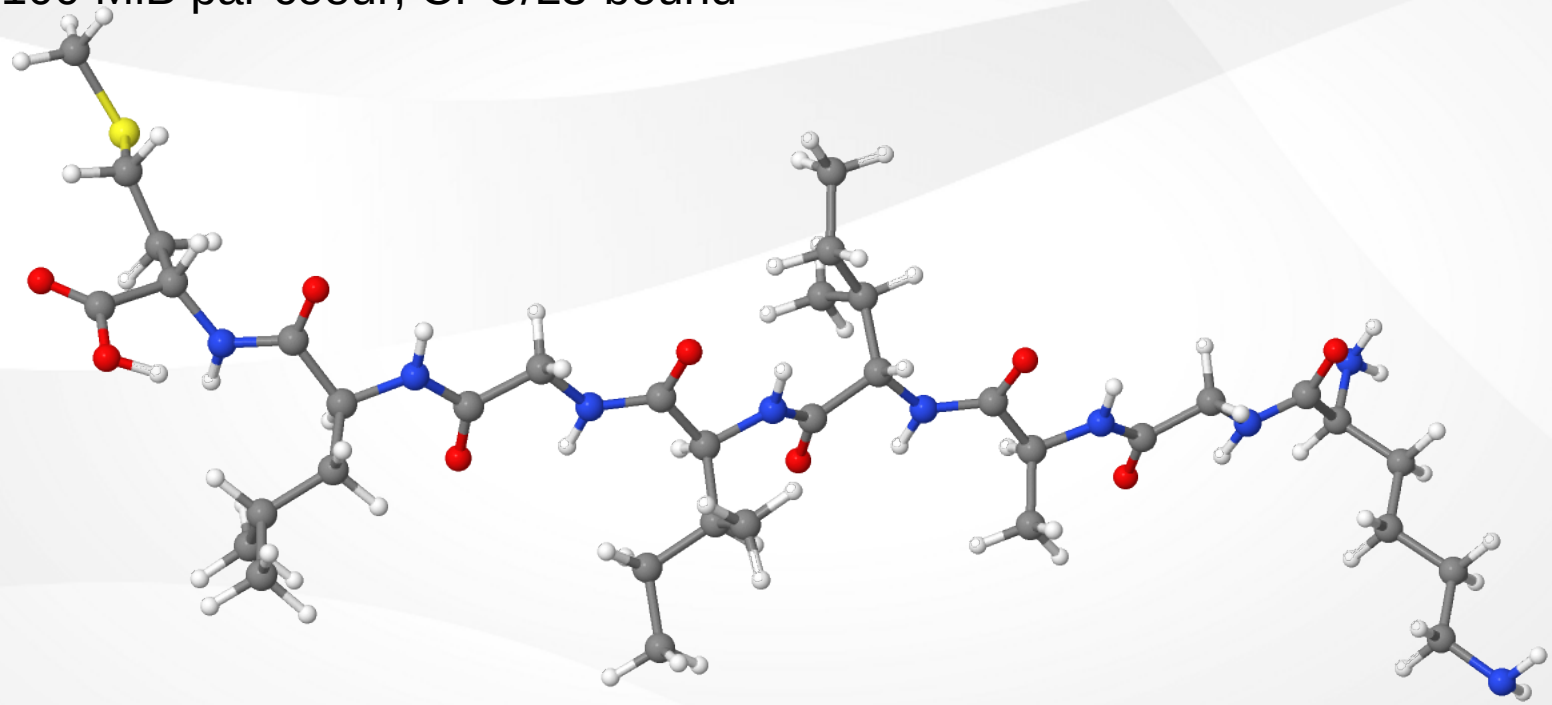
$$E = \frac{\int \mathrm{d}\mathbf{r}_1 \ldots \mathrm{d}\mathbf{r}_N \Phi(\mathbf{r}_1, \ldots, \mathbf{r}_N) \mathcal{H} \Phi(\mathbf{r}_1, \ldots, \mathbf{r}_N)}{\int \mathrm{d}\mathbf{r}_1 \ldots \mathrm{d}\mathbf{r}_N \Phi(\mathbf{r}_1, \ldots, \mathbf{r}_N) \Phi(\mathbf{r}_1, \ldots, \mathbf{r}_N)}$$

$$\sim \sum \frac{\mathcal{H}\Psi(\mathbf{r}_1, \ldots, \mathbf{r}_N)}{\Psi(\mathbf{r}_1, \ldots, \mathbf{r}_N)}, \text{ sampled with } (\Psi \times \Phi)$$

- Pas de calcul d'intégrale, donc pas de transformation à 4 indices.
- Nécessite d'évaluer Ψ, $\nabla_i \Psi$ et $\Delta_i \Psi$ à chaque position $\mathbf{r}_1, \ldots, \mathbf{r}_N$
- Toutes les positions $\mathbf{r}_1, \ldots, \mathbf{r}_N$ sont indépendantes $\rightarrow$: parallélisme massif

Code QMC=Chem développé au LCPQ (Toulouse)

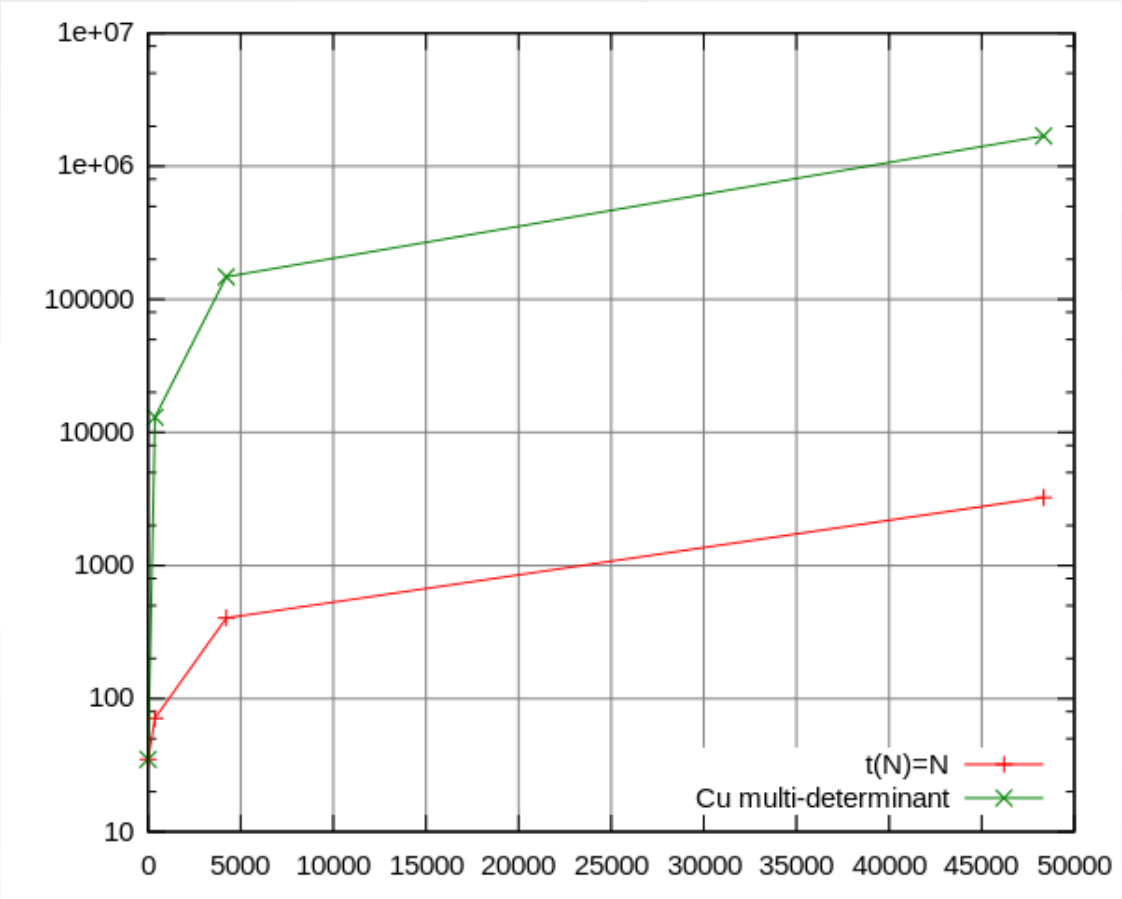
- En 2011: 0.96 PFlops/s soutenus sur Curie pendant 24h (76 800 coeurs), 434 électrons, 2960 AOs
- ~100 MiB par coeur, CPU/L3-bound



Optimisation mono-coeur

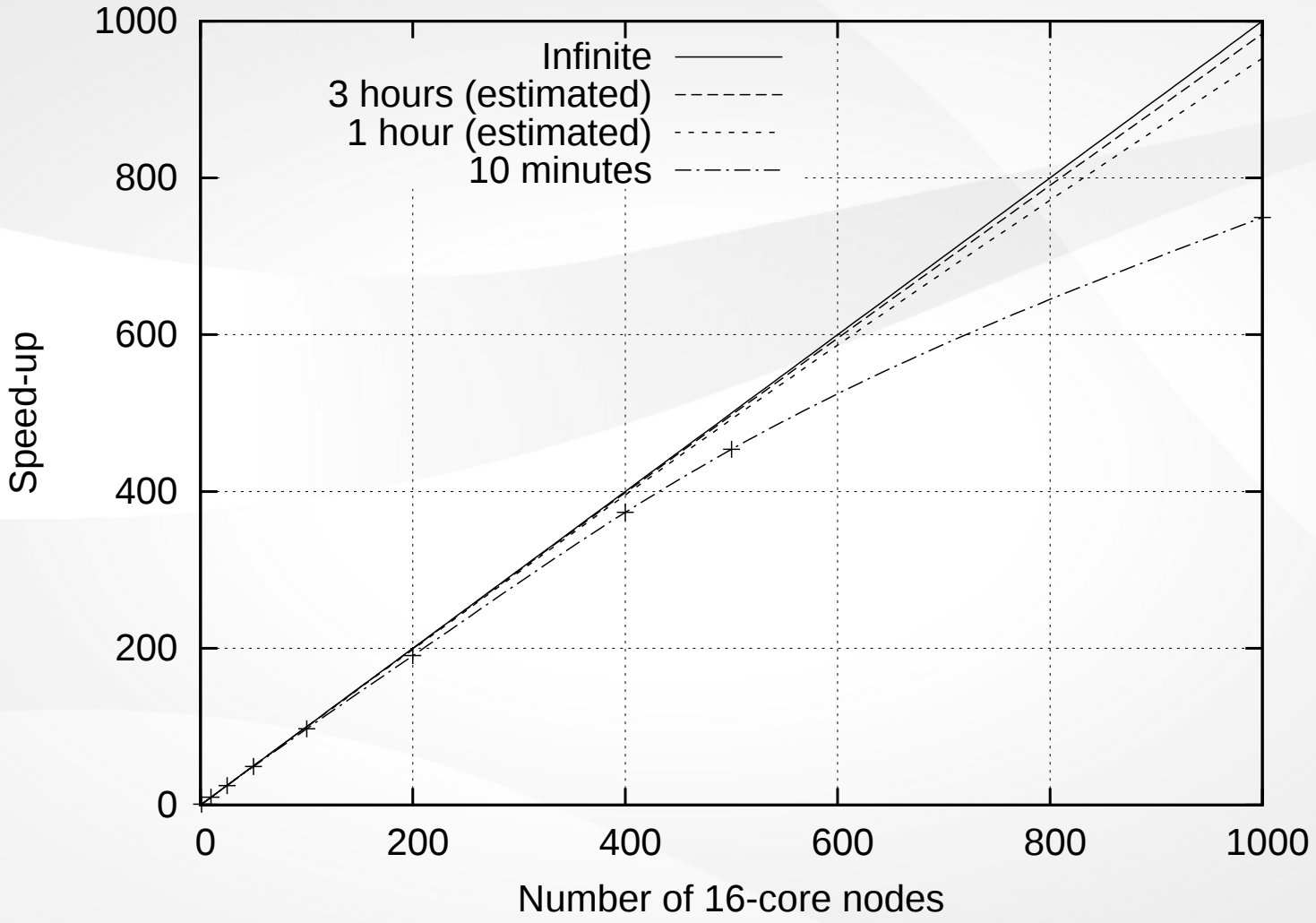
- Difficulté : manipulation de petites matrices (~1000 x 1000)
- Utilisation de la simple precision quand c'est pertinent
- Produit matrice dense $\times$vecteur creux pour calculer les MOs : atteint 100% du peak quand tout est en L1. 63.6% mesuré sur Sandy Bridge.
- Uniquement FMA vectoriel dans les boucles internes, donc accélère systématiquement sur les nouvelles architecture
- Gain de 10% avec HyperThreading
- Collaboration avec W. Jalby et E. Oseret (Exascale computing Ressearch, Versailles). Utilisation de MAQAO (analyse statique) pour aider l'opimisation.
- Plus efficace que des splines 3D (et beaucoup plus de flops).

Très efficace pour les fonctions d'ondes multi-determinantales (méthodes "post-Full-CI"):


1e+07
1e+06
100000
10000
1000
100
10
0
5000
10000
15000
20000
25000
30000
35000
40000
45000
50000
t(N)=N
Cu multi-determinant

Parallélisme

- Modèle client/serveur
- Client : Fortran, Serveur : Ocaml
- Algorithme mixte poids/marcheurs (PDMC/DMC) : évite de distribuer la population. Donc 1 trajectoire par coeur et pas de communication entre trajectoires.
- Communications et I/O toujours non-bloquantes
- Bibliothèque de communication ZeroMQ : pas de MPI donc
  - Tolérance aux pannes
  - Gestion elastique des ressources (ajout/suppression de noeuds)
  - Fonctionne sur grille / cloud
- Efficacité parallèle de 98.4 % sur 16 000 coeurs pour un run de 3h
- Calcul des plus basses energies totales variationnelles de la littérature pour les metaux de transition en un week-end (12 000 coeurs, 40 000 determinants).


Infinite
3 hours (estimated)
1 hour (estimated)
10 minutes
Speed-up
Number of 16-core nodes
0
200
400
600
800
1000

FCI-QMC

Cleland, Booth, Alavi et al : Full-CI QMC (2009)

- Résolution des équations du Full-CI dans la base des déterminants avec un algorithme stochastique
- Nécessite d'avoir les intégrales moléculaires (transformation à 4 indices)
- Contrairement à Davidson, pas besoin de stocker le vecteur propre dans la base complète
- Énergies Full-CI/cc-pCVQZ des diatomiques de la 1ère ligne
- Échantillonnage de la matrice densité à 2 corps, donc accès à toutes les propriétés
- Ils sont en train de développer le CAS-SCF stochastique pour traiter des CAS de plusieurs dizaines d'électrons

Remarque

Toutes les méthodes post-HF approximent le FCI. Maintenant, on peut avoir accès au Full-CI pour des systèmes non-triviaux.

MP2

Willow, Kim et Hirata (2012)

- But: Suppression de la transformation à 4 indices et du stockage des intégrales en RAM. Donc accès au parallélisme massif:
- Transformée de Laplace, et intégration Monte Carlo dans un espace à 12 dimensions

$$E = 2\sum_{i,j}^{\text{occ.}}\sum_{a,b}^{\text{vir.}} \frac{\langle ij|ab\rangle\langle ab|ij\rangle}{\varepsilon_i+\varepsilon_j-\varepsilon_a-\varepsilon_b} - \sum_{i,j}^{\text{occ.}}\sum_{a,b}^{\text{vir.}} \frac{\langle ij|ab\rangle\langle ab|ji\rangle}{\varepsilon_i+\varepsilon_j-\varepsilon_a-\varepsilon_b}$$

$$\frac{1}{\varepsilon_i+\varepsilon_j-\varepsilon_a-\varepsilon_b} = -\int_0^\infty d\tau \exp\left\{(\varepsilon_i+\varepsilon_j-\varepsilon_a-\varepsilon_b)\tau\right\}$$

$$E_B^{(2)} = \int d\boldsymbol{r}_1 \int d\boldsymbol{r}_2 \int d\boldsymbol{r}_3 \int d\boldsymbol{r}_4 \int_0^\infty d\tau \times \frac{o(\boldsymbol{r}_1,\boldsymbol{r}_3,\tau)\, o(\boldsymbol{r}_2,\boldsymbol{r}_4,\tau)\, v(\boldsymbol{r}_1,\boldsymbol{r}_4,\tau)\, v(\boldsymbol{r}_2,\boldsymbol{r}_3,\tau)}{r_{12} r_{34}}$$

Conclusion

- Le QMC est la preuve qu'on pourra faire de la chimie quantique sur les machines exascale car:
  - Massivement parallélisable
  - Utilise très peu de RAM (la RAM par coeur va encore diminuer)
  - Tolérant aux pannes
  - Pas de synchronisation nécessaire
  - Fonctionne bien sur grille, donc avec des réseau à grande latence
  - Fonctionnera naturellement sur des machines Exascale
- On voit apparaître des versions stochastiques de toutes les méthodes (Full-CI, MP2, CAS-SCF, Coupled Cluster)
- Quand elles auront la maturité des méthodes QMC, elles seront probablement les mêmes bénéfices que le QMC
- Le post-HF a un bel avenir, mais avec de nouveaux codes

Publicité

ZeroMQ : Bibliothèque d'échange de messages haute performance

http://zeromq.org

- Echange de message asynchrone entre threads, processus ou inter-noeuds.
- Facile à utiliser
- Rapide : 8M messages/seconde, 30 microsecondes de latence
- Open source
- Tolérant aux pannes
- Disponible en C, C++, Java, Ocaml, Python, PHP, Ruby, Perl, **Fortran**
- Utilisé par AT&T, Cisco, EA, Los Alamos Labs, NASA, Weta Digital, Zynga, Spotify, Samsung Electronics, Microsoft, CERN, et *LCPQ*.

Fortran : https://github.com/scemama/f77_zmq