



# Calcul haute performance & chimie quantique

Thomas Applencourt

## ► To cite this version:

Thomas Applencourt. Calcul haute performance & chimie quantique. Chimie-Physique [physics.chem-ph]. Université Paul Sabatier - Toulouse III, 2015. Français. NNT : 2015TOU30162 . tel-01244745v2

**HAL Id: tel-01244745**

**<https://theses.hal.science/tel-01244745v2>**

Submitted on 26 Aug 2016

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Université Fédérale



Toulouse Midi-Pyrénées

# THÈSE

en vue de l'obtention du titre de

## DOCTEUR DE L'UNIVERSITÉ FÉDÉRALE TOULOUSE MIDI-PYRÉNÉES

Délivré par : l'Université Toulouse III Paul Sabatier (UT<sub>3</sub> Paul Sabatier)

---

---

Présentée et soutenue le 02/11/2015 par :  
THOMAS APPLENCOURT

### Calcul Haute Performance & Chimie Quantique

---

---

#### JURY

---

THIERRY DEUTSCH	Directeur de Recherche	Rapporteur
JEAN-PHILIP PIQUEMAL	Professeur d'Université	Rapporteur
JEAN-MARC PIERSON	Professeur d'Université	Examineur
MICHEL KERN	Chargé de Recherche	Examineur
MICHEL CAFFAREL	Directeur de Recherche	Directeur de Thèse
ANTHONY SCEMAMA	Ingénieur de Recherche	Co-directeur de Thèse

---

École doctorale et spécialité :

*SDM : Physique de la matière - CO090*

Unité de Recherche :

*Laboratoire de Chimie et Physique Quantiques - UMR5626*



« Il vaut mieux se taire et passer pour un con plutôt que de parler et de ne laisser aucun doute sur le sujet » a peut être écrit DESPROGES.  
M'enfin là, j'ai plus trop le choix ...



## REMERCIEMENTS

« Chères consœurs, chers confrères, excusez-moi d'accaparer la parole et, qui pis est, pour un exposé sans pudeur » [2]. C'est avec cette phrase magistrale que j'aurais voulu commencer cette thèse. Mais d'un commun accord unilatéral, le comité dirigeant en a décidé autrement. . . Pourtant cette découverte est due à deux personnes qui ont transformé ces longues années de travaux acharnés et aliénantes en un rêve stakhanoviste.


La première personne susnommée est mon bon vieux T110 II. Je ne saurais jamais assez le remercier pour sa patience infinie et sa capacité de concentration en tout point remarquable. Bien sûr, on a eu des hauts et des bas ; des 0 et des 1 ; des *segmentation fault* et des rm un peu trop véloce ; tu as même eu quelques bouffées de chaleur après nos longues sessions de travail. Mais quand ces souvenirs auront traversé le tamis du temps, il ne restera que des moments heureux de franche camaraderie j'en suis sûr !

Cette camaraderie, je la dois aussi à la deuxième personne ayant à la sérénité de cette magnifique phrase d'introduction : la bien nommée MALIKA. Le cynisme n'est bon que partagé, et je te remercie pour cela Championne du monde d'orthographe d'Indre et Loire 2008 catégorie poids plume. Afin de continuer dans l'orientalisme, j'aimerais aussi te remercier NADIA. Tolérante, sans être une bonne poire ; dotée d'une force née des difficultés traversées ; et faisant les meilleurs cannelés du monde : tu es mon modèle d'*über-mensch* !

En parlant de cannelés, je suis désolé LUCIE, je sais cela te vexe en tant que bordelaise, mais tu ne m'en as pas encore fait alors, malheureusement tu perds cette compétition par forfait. Pourtant, on en a traversé des choses ensemble. Alors, merci pour tous ces moments partagés, toutes ces découvertes et tous leurs plaisirs associés ; sans toi je n'aurais été qu'un petit fantoche encore plus pétri de cynisme et d'alcool bon marché que maintenant.

Si tu m'as appris à vivre comme un épicurien alors SOPHIE m'a aidé à devenir un *homme bon à marier*. Grâce à cette dame – qui est dotée d'un dévouement forçant le respect ; de goût musical tout à fait respectable ; et qui a dû naître sous un arbre à palabre – je suis fier d'annoncer que j'arrive maintenant à utiliser un tire-bouchon. Hé oui ! À homme exceptionnel, compétences exceptionnelles. Et je ne mens pas, tous les gens bons (ARJAN, PINA, NICOLAS, DANIEL, FABIENNE, . . .) de notre repas gastronomique hebdomadaire peuvent en témoigner. Ces repas furent une

bouffée d'air frais et alcoolisé tout au long de ces trois années remplies de solitude heureuse et de pâtes Rana.

Parler gastronomie m'oblige tout naturellement à parler de Nathanael, Nathan...de NATHOU. On s'est rencontré en cueillant des champignons et maintenant on passe nos soirées à ne pas mourir de faim sur internet ; il m'a appris à cuire des crevettes aussi. Si ce n'est pas une histoire de gastronomes en culotte courte, je ne sais ce qui peut l'être. Gastronomiquement, j'ai éclusé trop peu de bières avec TINMAR mais quand même : merci compagnon de galère  !

Plus intimement, je tiens à remercier MARI-CO qui fut l'une de celle qui me maintint la tête hors de l'eau quand il le fallait et qui me laissa me noyer quand les circonstances semblaient l'imposer. Un grand et sincère merci à toi, Ψ ! Sans transition aucune, je tiens aussi à remercier AUDREY, pour le dépaysement procuré par nos explorations du monde de l'*underground* et pour me prouver que sur les cendres de relations pas si romantiques que cela peut naître une amitié véritable... ou du moins des bonnes soirées.

Étant d'une normalité sans aspérité, il fut beaucoup sujet de doutes pendant cette thèse ; mais autant professionnellement qu'humainement j'ai beaucoup appris et pour cela, il est venu le temps de mon *ex-voto* au JANUS de mon panthéon ; j'ai nommé MICHEL & ANTHONY, qui avec une face tournée vers la physique-chimie et l'autre vers l'informatique furent les meilleurs directeurs de thèse que je n'aie jamais eus ? <sup>1</sup> Plus sérieusement, je souhaiterais les remercier pour leur ouverture d'esprit ainsi que la liberté qu'ils m'ont laissée. Pudiquement, j'aimerais te remercier Michel pour une certaine philosophie de vie assez rafraîchissante dans le contexte actuel assez morose. Quant à Anthony, merci à toi d'avoir partagé avec moi un certain sens esthétique vis-à-vis de l'informatique et pour ta maîtrise *amazing* de cet art ; cela va sans dire. « Aucun homme n'est jamais assez fort pour ce calcul », me disais-je quand j'avais du trac jusqu'au cou ; v'là-t'y pas que tu arrivais pour me faire mentir. Merci !

Dans un domaine plus formel, je tiens à remercier mes rapporteurs (Thierry DEUSTCH et Jean-Philip PIQUEMAL) et mes deux examinateurs (Jean-Marc PIERSON et Michel KERN) pour le temps consacré à ce manuscrit. Cela paraît convenu, mais merci beaucoup de prendre le temps de me lire ; rien ne vous y obligeait.

Rassurez vous, j'en ai fini avec les remerciements personnels, mais je vais quand faire terminer par une liste non exhaustive de toutes les personnes – au sens large – m'ayant apporté leur soutien. On ne sait jamais, avec ces histoires de karma ; alors dans le doute, je prend le

---

1. Oui, il s'agit d'un point d'ironie [1]

même pari que Pascal [3]. Malgré tout, rongé par mes problèmes de rancœur, j'interposerai dans cette liste des gens que je ne remercie pas du tout. Faut pas pousser !

Schiele	Mike
O'Saj	Arthur
Pillon	l'Utopia
Camus	Le palais des thés
Le bar de Manu	Bastien
Manu	Reddit
Amorino	Le hippy jouant sous mes fenêtres
Lian Tze Lim	Hélène
Vijay	Fernand
Burger & Co	Cyril
L'atelier du Burger	Noir désir
Alexis	Florian
Le houmous	Ducros
Valve	Les Thaï Select
Sophie	ShinShin
Schopenhauer	Fredo
Godzilla	Ma conseillère Knorr
Behemoth	Les livres
Sorbet d'amour	Les livres
L'homme de l'autre côté du miroir	Les livres

Après ces trois pages de remerciements pour ces trois années, il ne me reste plus qu'à vous souhaiter une bonne et courageuse lecture.

## BIBLIOGRAPHIE

1. BRAHM, A. d. *L'ostensoir des ironies* (1899).
2. GOOSSE, A. *Comment peut-on être grammairien ?* in (Académie royale de langue et de littérature françaises de Belgique, 1999). <<http://www.arllfb.be/ebibliotheque/communications/goosse100499.pdf>>.
3. PASCAL, B. & BRUNSCHVICG, L. *Pensées* (Hachette, 1909).



## ERRATUM

On me signale que j'ai oublié de remercier ceux sans qui je ne serais pas là : Maman CHRISTIAN(N)E et Papa BRUNO. Merci à vous pour – hum – tout ce dont vous vous sentez responsables.

# TABLE DES MATIÈRES

1	INTRODUCTION	1
2	MÉTHODES D'INTERACTION DE CONFIGURATION & MONTE-CARLO QUANTIQUE	9
2.1	Méthodes d'Interaction de Configuration . . . . .	9
2.1.1	Formulation mathématique . . . . .	9
2.1.2	Base finie : <i>Full Configuration Interaction (Full-CI)</i> . .	11
2.1.3	Troncature de l'espace des déterminants . . . . .	12
2.1.4	Notion d'espace de Hilbert sélectionné . . . . .	13
2.1.5	Sélection perturbative de l'espace des déterminants	14
2.1.6	Conclusion sur les méthodes d'Interaction de Configuration (IC) . . . . .	16
2.2	Méthodes Monte-Carlo quantique (QMC) . . . . .	17
2.2.1	Base des positions des électrons . . . . .	18
2.2.2	Diffusion Monte Carlo (DMC) sans échantillonnage selon l'importance ( <i>no importance sampling</i> ) . . . . .	18
2.2.3	Diffusion Monte Carlo avec échantillonnage selon l'importance ( <i>importance sampling</i> ) . . . . .	20
2.2.4	Approximation des nœuds fixés : <i>Fixed-Node Diffusion Monte Carlo</i> . . . . .	21
2.2.5	Aspects pratiques de l'algorithme FN-DMC . . . . .	22
2.3	Connections entre les deux mondes . . . . .	26
2.3.1	Paradigme CIPSI comme fonction d'onde d'essai .	26
2.3.2	Étude QMC-CIPSI des atomes de transition de la série $3d$ . . . . .	29
2.3.3	Calcul des différences d'énergie et problème de la compensation des erreurs des nœuds-fixés . . . . .	34
2.4	Conclusion : le meilleur des deux mondes . . . . .	40
3	ALGORITHMIQUE & IMPLÉMENTATION	43
3.1	Processeurs et Parallélisation . . . . .	43
3.1.1	Architecture des processeurs . . . . .	44
3.1.2	Cœurs de calcul . . . . .	50
3.1.3	Distribution d'un calcul entre plusieurs cœurs . . .	54
3.2	Supercalculateurs et <i>cloud computing</i> . . . . .	63
3.2.1	Supercalculateurs . . . . .	64
3.2.2	Cloud computing . . . . .	64
3.3	Application : CIPSI & Quantum Package . . . . .	68
3.3.1	Adressage des intégrales . . . . .	68

3.3.2	Construction de la matrice hamiltonienne . . . . .	70
3.3.3	Problématique de l'unicité . . . . .	73
3.3.4	Parallélisation utilisant OpenMP . . . . .	74
3.3.5	Conclusion . . . . .	74
3.4	Application : QMC et QMC=Chem . . . . .	75
3.4.1	Optimisation du processus monocœur . . . . .	75
3.4.2	Parallélisation de type client-serveur . . . . .	80
4	DE LA RÉDUCTION DE LA COMPLEXITÉ . . . . .	83
4.1	Pourquoi la complexité est un <i>casus belli</i> . . . . .	83
4.1.1	Conditions nécessaires pour un code simple . . . . .	85
4.2	Simplification pour le développeur . . . . .	87
4.2.1	Réduction du temps de Compilation : Ninja . . . . .	88
4.2.2	Réduction de la complexité du cœur calculatoire : IRP . . . . .	93
4.2.3	Système de contrôle de versions : Exemple de Git . . . . .	98
4.2.4	Outils d'aide pour la gestion de projet collaboratif . . . . .	100
4.2.5	Conclusion . . . . .	104
4.3	Simplification pour l'utilisateur . . . . .	104
4.3.1	Tous les fichiers de bases atomiques à disposition . . . . .	104
4.3.2	Du stockage des données . . . . .	109
4.3.3	Interface graphique . . . . .	110
4.4	Note sur les fichiers d'entrée et de sortie . . . . .	113
4.4.1	Le poids de l'histoire . . . . .	113
4.4.2	Du fichier d'entrée . . . . .	114
4.4.3	Fichier de sortie . . . . .	121
4.5	Conclusion : Recette pour résoudre la complexité . . . . .	121
5	CONCLUSION GÉNÉRALE . . . . .	123
A	SQGIT : GESTION DES VERSIONS DES FICHIERS SQL AVEC GIT . . . . .	127
A.1	Idée . . . . .	127
A.2	Mise en œuvre . . . . .	127
A.3	Aspect technique : Python . . . . .	128
B	DIDACTICIEL QUANTUM PACKAGE & QMC=CHEM . . . . .	129
B.1	Quantum Package . . . . .	129
B.1.1	Téléchargement, configuration et compilation . . . . .	129
B.1.2	Lancement des calculs . . . . .	132
B.2	QMC=Chem . . . . .	136
B.2.1	Commande . . . . .	136
B.2.2	Calcul . . . . .	137
C	ACCURATE NON RELATIVISTIC GROUND-STATE ENERGIES OF 3d TRANSITION METAL ATOMS . . . . .	141

D QUANTUM MONTE CARLO WITH VERY LARGE MULTI-DETERMINANT EXPANSIONS	147
---	-----

---

## TABLE DES FIGURES

FIGURE 1	Energy differences beetwen <i>Fixed-Node Diffusion Monte Carlo</i> (FN-DMC) and exact one . . . . .	36
FIGURE 2	Computing evolution overview . . . . .	45
FIGURE 3	Von Neumann architecture . . . . .	46
FIGURE 4	Memory hierarchy . . . . .	48
FIGURE 5	Pipeline d'exécution de $C_i = A_i \times B_i$ . . . . .	51
FIGURE 6	<i>Single Instruction, Multiple Data</i> (SIMD) schema . .	52
FIGURE 7	Vectorization with and without peel and tail loops	53
FIGURE 8	ANDAHl's law representation . . . . .	55
FIGURE 9	Standard representation of parallelization . . . . .	56
FIGURE 10	PGAS representation . . . . .	61
FIGURE 11	Adding some resources dynamically to QMC=Chem	67
FIGURE 12	Resarch of an integral key . . . . .	69
FIGURE 13	Bitmask xor operation : Positions of holes and particles . . . . .	72
FIGURE 14	popcount use in Quantum Package . . . . .	73
FIGURE 15	Quantum Package using OpenMP . . . . .	75
FIGURE 16	QMC=Chem parallelisation overview . . . . .	80
FIGURE 17	<i>Relevant xkcd for the compilation time</i> [128] . . . . .	88
FIGURE 18	Zen of <i>Implicit Reference to Parameters</i> (IRP) . . . . .	94
FIGURE 19	IRPF90 in action . . . . .	96
FIGURE 20	Merge workflow . . . . .	103
FIGURE 21	The API for the G2 in action . . . . .	110

## LISTE DES TABLEAUX

TABLE 2	GENCI allocation of time in 2012 . . . . .	2
TABLE 3	Comparison between the representations used in Configuration Interaction and <i>Quantum Monte Carlo</i> , Monte Carlo quantique (QMC) . . . . .	17
TABLE 4	Slater basis set energies comparaison . . . . .	32
TABLE 5	FN-DMC total energies for transition metal atoms	33
TABLE 6	MAD of the atomization energies . . . . .	35
TABLE 7	mad . . . . .	39

TABLE 8	MAD of the atomization energies with pseudo-potentials . . . . .	40
TABLE 9	Cache latency . . . . .	48
TABLE 10	Overview of the computational resources used in this thesis. . . . .	65
TABLE 11	Characteristics of processors (cloud and other) . .	65
TABLE 12	Ninja vs Make benchmark . . . . .	92

## LISTE DES CODES SOURCES

CODE 1	OpenMP example in Fortran . . . . .	57
CODE 2	Socket Client Server . . . . .	58
CODE 3	Message Passing Interface (MPI) representation . .	60
CODE 4	ØMQ REQ/REP Server and Client in Python . .	63
CODE 5	Determinant driven algorithm . . . . .	70
CODE 6	How to get determinants . . . . .	77
CODE 7	General syntax of a simple Makefile . . . . .	89
CODE 8	Syntax of a simple Makefile . . . . .	89
CODE 9	Syntax of a real Makefile . . . . .	89
CODE 10	Syntax of a simple build.ninja file. . . . .	90
CODE 11	Generated Fortran code by IRPF90 . . . . .	97
CODE 12	UnitTest implementation . . . . .	102
CODE 13	Request librarie . . . . .	106
CODE 14	EMSL local API . . . . .	108
CODE 15	Flask Hello World . . . . .	112
CODE 16	EZFIO configuration file . . . . .	116
CODE 17	Example of EZFIO usage . . . . .	116
CODE 18	Example of a high-level configfile . . . . .	118
CODE 19	Example of template . . . . .	119

# GLOSSAIRE

$\mathcal{H}$	est l'hamiltonien du système. xiv, 14–16, 71, 72
$\mathcal{P}$	est l'espace perturbatif du <i>Configuration Interaction using Perturbative Selection done Iteratively</i> (CIPSI), c'est-à-dire l'ensemble des déterminants connectés à $\mathcal{V}$ par une application de $\mathcal{H}$ mais qui ne sont pas dans $\mathcal{V}$ . 14, 15, 71, 72
$\mathcal{V}$	est l'espace variationnel du CIPSI, c'est-à-dire l'ensemble des déterminants ayant servis à diagonaliser $\mathcal{H}$ . xiv, 71
<i>Full-CI</i>	<i>Full Configuration Interaction</i> . ix, 3, 8, 10–12, 15, 16, 34, 36, 39, 117, 126, 128, 131, 132
<i>Hartree-Fock</i>	Fonction d'onde monodéterminentale minimisant l'énergie. 3, 4, 11–13, 23, 29–35, 38, 39, 70, 110, 117, 126, 128, 129
ALU	<i>Arithmetic Logic Unit</i> . 44, 49
API	<i>Application Program Interface</i> . 54, 56, 57, 60, 83, 112, 117, 132
Bash	Le Bourne Again Shell est un interpréteur de commandes ( <i>shell</i> ). 110, 113, 132
CIPSI	<i>Configuration Interaction using Perturbative Selection done Iteratively</i> . iv, xiv, 4, 13–16, 27, 28, 30–32, 34, 36, 39, 64, 66, 68, 70, 71, 126
CPU	<i>Central Processing Unit</i> . 42–44, 48, 50, 51, 58, 65, 70, 71, 78
DFT	<i>Density Functional Theory</i> . 2–4, 23, 30, 33
DRY	<i>Don't repeat yourself</i> . 84
EMSL	<i>Environment Molecular Sciences Laboratory, Pacific Northwest National Laboratory, US Department of Energy</i> . 10, 102–104, 129
EZFIO	<i>Easy Fortran I/O library generator</i> . 83, 113–118, 129, 132

FN-DMC	<i>Fixed-Node Diffusion Monte Carlo</i> . xii, 21, 26, 31–39, 73, 126, 133–135
Fortran	Le Fortran est un langage de bas niveau ; donc aussi rapide que le C. Certains auteurs considèrent que ses évolutions (Fortran 95, 2003, 2008) ne doivent pas être considérées car elles nuisent à la performance.[89]. xiii, xv, 29, 47, 55, 60, 88, 90–92, 94, 101, 109, 111, 113, 115, 117
Git	Git ([116] et sous-section 4.2.3) est un outil de gestion de versions. Ce n’est pas un acronyme. Git est l’équivalent anglais de « con ». On doit l’explication à son créateur Linux Torvalds : <i>I’m an egotistical bastard, and I name all my projects after myself. First ‘Linux’, now ‘Git’</i> . xv, 95–100, 119, 122, 124–126
Github	Github ([100] et sous-sous-section 4.2.4.1) est un service web permettant le partage de dépôts Git. Il est parfois considéré comme un réseau social pour développeurs. xvii, 97–100, 119, 122, 126
GTO	<i>Gaussian Type Orbital</i> . 10, 28
HPC	<i>High Performance Computing</i> . iv, xvi, 1, 5, 6, 57, 61–63, 94, 120, 122, 123
HPL	<i>High Performance LINPACK</i> . 41, 42, 47
I/O	Entrée/Sortie ( <i>Input/Output</i> en anglais). 83, 110, 118, 119
IC	Interaction de Configuration. iv, ix, 3, 4, 6, 8, 10–13, 15, 17, 23, 26, 27, 29–31, 34, 40, 98, 120, 126
intensité arithmétique	Le nombre d’opérations flottantes par octet chargé en mémoire. 77
IRP	<i>Implicit Reference to Parameters</i> . xii, xv, xvi, 83, 90, 91, 95, 118, 122, 131
IRPF90	Extension du Fortran90 qui implémente la méthode IRP [109]. xii, xiii, 29, 84, 88, 90–94, 101, 115, 122
LAPACK	<i>Linear Algebra PACKage</i> . 42
LRU	<i>Least Recently Used</i> . 45
MAD	<i>Mean Absolute Deviation</i> . 34, 35, 37–39



Make	Make [62] est un utilitaire de création de binaires automatique possédant de –trop– nombreuses options (Voir aussi Ninja). xiii, xvi, 6, 85–91, 121, 128
modèle Client-Serveur	Le clients envoient des requêtes ; les serveurs attendent les requêtes des clients et y répondent. Ce type d’architecture est utilisée dans de nombreux domaines (serveurs web par exemple). Une application dans le domaine du <i>High Performance Computing</i> (HPC) en chimie quantique est l’équilibrage de charge lors du calcul des intégrales bi-électroniques[77]. 60, 79
MPI	<i>Message Passing Interface</i> . xiii, 4, 57–59, 62, 79, 121
MPQC	<i>Massively Parallel Quantum Chemistry</i> . 63
MTBF	<i>Mean Time Between Failure</i> . 54
Ninja	Ninja [90] est un utilitaire de création de binaires automatique orienté vers la rapidité. Il possède –trop– peu d’options. (Voir aussi Make). xiii, xvi, 87–90, 117–119, 127, 128
NUMA	<i>Non-Uniform Memory Access</i> . 47, 55
OCaml	OCaml est un langage fonctionnel fortement typé. 79, 113–117, 119, 127, 128
OM	orbitale moléculaire. 2, 8, 9
OpenMP	<i>Open Multi-Processing</i> . 6, 54, 55
PGAS	<i>Partitioned Global Address Space</i> . 59
popcount	est le nom de la procédure permettant de compter le nombre de bits à 1 dans une <i>bit-string</i> . Elle est disponible en un cycle CPU sur les processeurs x86 [101] depuis le jeu d’instructions SSE4.2. xii, 65, 70–72
Provider	Dans le modèle IRP, un <i>provider</i> est une routine sans argument qui permet d’obtenir la valeur d’une variable, en la calculant si nécessaire.. 91, 92, 95, 115, 119, 131
Python	Python est un langage de haut niveau interprété. x, xiii, xvii, 56, 61, 64, 66, 84, 88, 89, 98, 99, 103, 104, 106, 108, 109, 113, 115, 122, 125, 128
QMC	<i>Quantum Monte Carlo</i> , Monte Carlo quantique. iv, 4, 6, 16, 17, 20, 21, 25, 27, 28, 31, 33, 36–40, 58, 63–65, 74, 120, 121

QMC=Chem	est un code de chimie quantique développé au LCPQ et implémentant des méthodes de type Monte Carlo quantique. <a href="#">iv</a> , <a href="#">xii</a> , <a href="#">6</a> , <a href="#">29</a> , <a href="#">31</a> , <a href="#">61</a> , <a href="#">65</a> , <a href="#">73</a> , <a href="#">78</a> , <a href="#">126</a> , <a href="#">133</a>
Quantum Package	est un code de chimie quantique développé au LCPQ et permettant le développement aisé de nombreuses méthodes d'interaction de configuration. <a href="#">iv</a> , <a href="#">xii</a> , <a href="#">6</a> , <a href="#">28</a> , <a href="#">29</a> , <a href="#">38</a> , <a href="#">65–67</a> , <a href="#">70–73</a> , <a href="#">82</a> , <a href="#">83</a> , <a href="#">88</a> , <a href="#">98</a> , <a href="#">104</a> , <a href="#">114</a> , <a href="#">117</a> , <a href="#">118</a> , <a href="#">126–129</a> , <a href="#">133</a>
SIMD	<i>Single Instruction, Multiple Data</i> . <a href="#">xii</a> , <a href="#">50</a> , <a href="#">77</a>
Single System Image	Machine multi-nœuds apparaissant comme un seul nœud, avec une seule instance distribuée du système d'exploitation. <a href="#">55</a>
SPMD	<i>Single Program, Multiple Data</i> . <a href="#">57</a> , <a href="#">60</a> , <a href="#">62</a>
SQL	<i>Structured Query Language</i> . <a href="#">xvii</a> , <a href="#">124</a> , <a href="#">125</a>
SQLite	SQLite est un gestionnaire de bases de données. Il implémente la plupart des standards <i>Structured Query Language</i> (SQL). À l'origine écrit dans une bibliothèque C, il est maintenant disponible dans plusieurs langages. Il fait partie de la bibliothèque standard de Python depuis la version 2.3. Pour un didacticiel en Python voir <a href="#">[17]</a> . <a href="#">104</a> , <a href="#">106</a> , <a href="#">124</a> , <a href="#">125</a>
STO	<i>Slater Type Orbital</i> . <a href="#">10</a> , <a href="#">28</a> , <a href="#">38</a>
Travis CI	TravisCI ( <a href="#">[79]</a> ) est un outil de <i>Continuous Integration</i> interfacé avec Github. <a href="#">99</a> , <a href="#">101</a> , <a href="#">119</a> , <a href="#">122</a>
VM	<i>machine virtuelle</i> . <a href="#">64</a>
ØMQ	ZeroMQ, zero-em-queue, <a href="#">[76]</a> est une bibliothèque asynchrone de communication haute performance tolérante aux pannes. <a href="#">xiii</a> , <a href="#">5</a> , <a href="#">6</a> , <a href="#">59–61</a> , <a href="#">72</a> , <a href="#">79</a> , <a href="#">109</a> , <a href="#">121</a>



# 1 | INTRODUCTION

Ce travail de thèse se situe à l'interface entre deux domaines : celui du développement et de l'application de méthodes originales pour l'étude de la structure électronique et celui du Calcul Haute Performance – *High Performance Computing* (HPC) – sous ses différents aspects. Deux objectifs principaux ont motivé cette étude :

1. Participer au développement de méthodes et algorithmes innovants adaptés à l'évolution actuelle et à venir du calcul scientifique et des supercalculateurs ;
2. Aider au déploiement des simulations à très grande échelle, aussi bien sous l'aspect informatique proprement dit :
  - utilisation de nouveaux paradigmes de programmation
  - optimisation des processus mono-cœur
  - calculs massivement parallèles sur grilles de calcul (supercalculateur et *Cloud*)
  - outils d'aide au développement collaboratifque sous l'aspect *utilisateur* :
  - gestion des paramètres d'entrée et de sortie
  - installation des codes
  - interface graphique
  - interfaçage avec d'autres codes.

DÉVELOPPEMENT MÉTHODOLOGIQUE. Le premier objectif qui concerne le développement de nouvelles méthodes me paraît fondamental. En effet, malgré des progrès considérables ces cinquante dernières années en chimie quantique, une méthode électronique générale alliant à la fois la précision chimique requise et un temps de restitution tolérable pour les grands systèmes reste encore à définir. Soulignons que cet objectif représente un défi scientifique et technologique majeur puisque la possibilité d'étudier des systèmes chimiques complexes sur ordinateur (*in silico*) ouvre la voie à des applications très variées à fort impact. Pour citer quelques exemples :

- étude des molécules de la vie (*drug design*, création de médicaments sur ordinateur)
- catalyse (comment rendre possible et/ou augmenter l'efficacité de réactions chimiques en vue de processus industriels plus performants)

Comité thématique		Heure (Millions)
CT 1	Environnement	55
CT 2	Mécanique des fluides, fluides réactifs, fluides complexes	142
CT 3	Simulation biomédicale et application à la santé	7
CT 4	Astrophysique et géophysique	60
CT 5	Physique théorique et physique des plasmas	46
CT 6	Informatique, Algorithmique et mathématiques	22
CT 7	Systèmes moléculaires organisés et biologie	58
CT 8	Chimie quantique et modélisation moléculaire	114
CT 9	Physique, chimie et propriétés des matériaux	108
CT 10	Nouvelles applications et applications transversales du calcul	6

TABLE 2 – GENCI allocation of time in 2012 [64] (in millions core-hours and in french). The repartition among the various scientific domains (*comités thématiques*) is presented.

- nanosciences (définition de composants électroniques à l'échelle moléculaire, *etc.*)

Du fait de cette importance, les simulations électroniques occupent une communauté très importante de chercheurs et d'ingénieurs tant du côté des développements que du côté applicatif. Ainsi, dans les centres de calcul, les simulations de la matière au niveau microscopique représentent une part importante de la consommation des ressources. Pour illustrer ce point je présente dans le [Tableau 2](#) la répartition des heures de calcul allouées en 2012 aux diverses disciplines scientifiques – classées par comité thématique (CT) – par le Grand Équipement National de Calcul Intensif (GENCI), qui est la structure qui fournissant la plus grande part des ressources informatiques de la communauté scientifique française. Comme on le voit le Comité Thématique N° 8 qui nous concerne directement est le deuxième plus important en volume ; notons que le CT N° 9 est également pour partie concerné par les simulations discutées ici.

**Deux grands types d'approches.** Très schématiquement, deux grands types d'approches se partagent actuellement la scène. La première repose sur l'expression de l'énergie comme une fonctionnelle de la densité électronique à un corps<sup>1</sup> et sur sa minimisation par rapport à un ensemble d'orbitales moléculaires (*orbitale moléculaire* (OM)) représentant la répartition électronique. Cette approche connue sous le nom de méthodes de la fonctionnelle de la densité – *Density Functional Theory* (DFT) – a l'avantage décisif de présenter à la fois une augmentation modérée de l'effort numérique en fonction du nombre d'électrons<sup>2</sup> et une précision

1. C'est à dire, la densité de probabilité de trouver un électron à un endroit donné.

2. Typiquement en  $\mathcal{O}(N^3)$ .

raisonnable permettant de mener des études sur des systèmes de taille importante<sup>3</sup>. Malheureusement, l'erreur systématique résultante de l'approximation choisie pour la fonctionnelle de l'énergie<sup>4</sup> dont l'expression exacte est inconnue, est mal contrôlée ; en effet il n'existe pas d'approche générale pour réduire cette erreur de manière systématique et uniforme dans l'ensemble des systèmes moléculaires ; en pratique, les études systématiques doivent donc être effectuées avec précaution. En général, pour une classe de molécules donnée avec des propriétés physico-chimiques similaires plusieurs types de fonctionnelles sont testées par comparaison avec l'expérience sur quelques systèmes et celle qui donne en moyenne les meilleurs résultats est sélectionnée pour une étude plus systématique. Malheureusement, malgré des résultats remarquables, cet aspect *empirique* représente une limitation pour ce type d'approche.

Un deuxième grande classe d'approches est celle des méthodes basées sur la fonction d'onde et appelées méthodes corrélées post-*Hartree-Fock*. Ces approches reposent sur la construction de la fonction d'onde comme un développement sur une base de déterminants et l'optimisation des différents paramètres par différentes techniques<sup>5</sup>. Contrairement aux approches *DFT*, l'erreur résiduelle issue de la troncature de la fonction d'onde est bien mieux contrôlée. Malheureusement, les coûts de calcul sont bien plus importants avec un accroissement de l'effort calculatoire en fonction du nombre d'électrons bien plus défavorable<sup>6</sup>.

**Mes contributions.** La première partie de mon travail de thèse concerne la partie recherche en chimie quantique proprement dite ; j'ai participé à plusieurs développements méthodologiques pour les approches de type fonction d'onde. Plus précisément, j'ai travaillé sur les approches de type Interaction de Configuration (*Interaction de Configuration (IC)*) où la fonction d'onde est développée sur un ensemble de déterminants associés à des occupations diverses des orbitales moléculaires et où les coefficients de ce développement sont obtenus par diagonalisation de la matrice Hamiltonienne. Dans l'approche d'IC la limite idéale est celle où l'ensemble des déterminants est pris en compte – limite dite d'IC complète ou *Full Configuration Interaction (Full-CI)*. Malheureusement, le nombre exponentiellement grand de déterminants interdit en pratique ce type de calcul, sauf – bien évidemment – pour des systèmes très simples. Ainsi sont apparues les méthodes dites « IC sélectionnée » qui consistent à sélectionner les déterminants les plus lourds de la fonction d'onde afin d'arriver à des développements beaucoup plus compacts mais contenant l'essentiel de la physique. Personnellement, j'ai travaillé dans le cadre

3. Jusqu'à quelques milliers d'électrons actifs.

4. Plus précisément l'énergie d'échange et de corrélation.

5. Minimisation de l'énergie variationnelle, calculs perturbatifs, projection, ...

6. Par exemple,  $\mathcal{O}(N^7)$  pour l'approche CCSD(T) considérée comme une très bonne approche pour les molécules proches de leur géométrie d'équilibre.

des méthodes d'IC sélectionnée itérativement – *Configuration Interaction using Perturbative Selection done Iteratively* (CIPSI) – où la contribution des déterminants candidats à entrer dans le développement de la fonction d'onde est faite de manière perturbative au second ordre. Ce type d'approche, qui a été développée dans les années 70 et 80, a été très récemment remis en scène dans notre groupe et connaît plus généralement un regain d'intérêt [95]. La raison semble triple :

1. D'une part cette expression compacte est un atout majeur pour le traitement des grands systèmes ;
2. D'autre part, nous verrons que grâce au processus de sélection et à l'aspect perturbatif, cette variante des méthodes de fonctions d'onde est adaptée au calcul parallèle. Ceci illustre un des message principaux de cette thèse, à savoir la nécessité de revisiter les algorithmes traditionnels et de sélectionner ceux qui sont les mieux adaptés à l'évolution informatique ;
3. Enfin, grâce une évolution récente dans le jeux d'instruction des processeurs, ce type d'algorithme est devenu au moins un ordre de grandeur plus rapide que dans les implémentations originelles.

La deuxième partie de ce travail de recherche repose sur le traitement stochastique de l'équation de Schrödinger. Elle appartient à la classe de méthodes dites *Quantum Monte Carlo*, Monte Carlo quantique (QMC). L'idée fondamentale est de propager les électrons considérés comme des particules ponctuelles à l'aide d'une dynamique stochastique<sup>7</sup> et d'exprimer les moyennes quantiques recherchées comme moyennes temporelles le long des trajectoires. Le point-clef de ce type d'approche est la transformation du problème difficile de la résolution d'une équation aux dérivées partielles à très grand nombre de variables indépendantes<sup>8</sup> en un problème de propagation d'une population de particules fictives appelées marcheurs ou encore *walkers*. Comme les particules fictives n'interagissent pas (ou très peu) entre elles, la dynamique peut être complètement parallélisée avec très peu de communications<sup>9</sup> : on parle alors de parallélisme « embarrassant ». De plus, comme il s'agit d'une approche où l'espace est échantillonné, l'algorithme est naturellement tolérant aux pannes ; si un processeur tombe en panne, la statistique réalisée par celui-ci est perdue mais le calcul peut continuer sur les autres processeurs. Les approches QMC se prêtent alors idéalement au calcul massivement parallèle sur des ordinateurs à très grand nombre de cœurs de calcul ou sur grille ; ce qui les différencie très nettement des méthodes standard présentées plus haut (calculs DFT et post-*Hartree-Fock*). Nous discutons notre choix technologique – l'abandon du *Message Passing Inter-*

7. Soit une combinaison d'un déplacement déterministe et d'un mouvement aléatoire.

8.  $3N$  variables où  $N$  est le nombre d'électrons.

9. Les communications sont toutes asynchrones.

*face* (MPI) au profit du ØMQ – pour ce type d’approche. Il s’agit d’un deuxième exemple encore plus frappant de la nécessité de sélectionner les méthodes adaptées au HPC. D’une manière générale, comme cela sera discuté en détail dans ce travail, les algorithmes stochastiques apparaissent comme une classe de méthodes idéalement adaptées au calcul massivement parallèle. Il est important de noter dès maintenant que les méthodes idéales ne sont pas celles qui minimisent le nombre d’opérations élémentaires, comme cela était le cas dans le paradigme de l’informatique à petit nombre de cœurs, mais celle qui minimisent le temps de restitution. Dans le cas des algorithmes stochastiques le nombre d’opérations peut être gigantesque par rapport à une approche faiblement parallélisable mais le parallélisme massif permet pour un nombre de processeurs suffisamment grand non seulement de compenser cet aspect mais de permettre d’accéder à des performances inatteignables autrement.

AIDE AU DÉVELOPPEMENT. Le deuxième objectif de ma thèse a représenté le cœur de mon travail et a consisté à faciliter le déploiement des simulations, tant au niveau des développeurs que de l’utilisateur.

Comme nous l’avons souligné, les simulations électroniques sont des applications critiques. Du fait de leur importance scientifique et technologique, elles mobilisent une communauté importante de chercheurs et d’ingénieurs ayant des affinités plus ou moins fortes avec l’outil informatique. De plus, la méthode idéale n’existait pas encore, les chercheurs produisent un effort continu de développements méthodologiques et de propositions d’algorithmes nouveaux. Pour finir de complexifier ce problème l’informatique évolue très rapidement, tant au niveau des machines, que des outils utilisés pour en tirer partie. En pratique, ce triptyque (niveau hétérogène des développeurs ; codes changeant sans cesse et pouvant vieillir rapidement, outils informatiques en constante évolution) font qu’il est difficile d’avoir en permanence des codes optimaux en terme d’efficacité. Pire encore, dans ce contexte l’implémentation par les chercheurs d’idées nouvelles peut même être abandonnée pour ces raisons, basement, pratiques. La situation des utilisateurs n’est pas plus enviable : les codes sont souvent assez difficiles à installer sur les supercalculateurs où les droits sont assez restreints ; la gestion des paramètres d’entrée de la simulation peut devenir un véritable sacerdoce ; enfin, l’interface avec des programmes tiers se fait souvent de manière assez artisanale.

Dans ce travail de thèse, je définis cet ensemble d’aspects qui limitent la portée des simulations comme la « complexité ».

Mon objectif principal a donc été de participer à la réduction de cette complexité en re-questionnant la pertinence des dogmes informatiques utilisés dans les codes de calcul scientifique de notre domaine. Bien que cela a été effectué dans le contexte des simulations électroniques où nous



avons beaucoup gagné, les techniques mises en place et/ou développées sont d'intérêt plus général puisqu'elles concernent le développement et la mise en place de simulations massives pour des applications critiques.

CONTENU DE LA THÈSE. Résumons maintenant le contenu de cette thèse. Après cette introduction, je présente dans le [chapitre 2](#) les deux méthodes scientifiques utilisées dans cette étude. J'insiste sur leur bonne<sup>10</sup> adaptation au HPC. Je présente plusieurs développements que j'ai effectués concernant l'utilisation de pseudo-potentiels, l'utilisation de fonctions de base de type Slater et le problème de la cohérence des résultats lors des calculs des différences d'énergie. Je présente en particulier deux applications :

1. le calcul des énergies totales des atomes de transition de la série 3d à l'aide d'une approche mixte IC sélectionnée-QMC qui nous a permis d'obtenir les meilleures énergies publiées à ce jour ;
2. une étude systématique des énergies d'atomisation pour un ensemble de molécules de référence (G2).

Le chapitre suivant ([chapitre 3](#)), plus informatique, est constitué de deux grandes parties. La première partie est une introduction aux aspects fondamentaux concernant l'architecture des processeurs, et ensuite une introduction aux différentes technologies pour implémenter le parallélisme. Je décris ensuite le Challenge *France Grilles* qui nous a permis de réaliser une application concrète de calcul scientifique sur le nuage informatique<sup>11</sup>. La deuxième partie, quant à elle, est une application des notions présentées plus haut. L'utilisation des caches et de la vectorisation ainsi que des bibliothèques de parallélisation (*Open Multi-Processing* (OpenMP), ØMQ) est illustrée pour les deux codes-maison<sup>12</sup>. Nous montrons que cela nous a permis d'augmenter singulièrement la performance de nos simulations.

Le [chapitre 4](#) est nommé DE LA RÉDUCTION DE LA COMPLEXITÉ. Il s'agit de présenter l'ensemble des idées et des outils que j'ai utilisés et/ou développés pour rendre les simulations possibles. Après une introduction présentant nos motivations je divise naturellement la présentation en deux parties distinctes concernant la simplification pour le développeur (discussions autour de *Make*, des paradigmes de programmation et du travail collaboratif) et celui pour l'utilisateur (gestion des fichiers d'entrée, interface graphique, *scripting*). Il s'agit d'une partition commode mais clairement trop schématique ; comme nous le verrons certains outils seront utiles pour les deux communautés.

---

10. Voire, très bonne.

11. *Cloud computing*.

12. Le premier pour l'IC (Quantum Package), le deuxième pour le QMC (QMC=Chem).

Je présente enfin une **CONCLUSION GÉNÉRALE** résumant la problématique générale abordée dans ce travail, les solutions proposées et les perspectives telles que je les imagine.



# 2

## MÉTHODES D'INTERACTION DE CONFIGURATION & MONTE-CARLO QUANTIQUE

2.1	Méthodes d'Interaction de Configuration . . . . .	9
2.1.1	Formulation mathématique . . . . .	9
2.1.2	Base finie : <i>Full Configuration Interaction (Full-CI)</i> . .	11
2.1.3	Troncature de l'espace des déterminants . . . . .	12
2.1.4	Notion d'espace de Hilbert sélectionné . . . . .	13
2.1.5	Sélection perturbative de l'espace des déterminants	14
2.1.6	Conclusion sur les méthodes d'Interaction de Configuration (IC) . . . . .	16
2.2	Méthodes Monte-Carlo quantique (QMC) . . . . .	17
2.2.1	Base des positions des électrons . . . . .	18
2.2.2	Diffusion Monte Carlo (DMC) sans échantillonnage selon l'importance ( <i>no importance sampling</i> ) . . . . .	18
2.2.3	Diffusion Monte Carlo avec échantillonnage selon l'importance ( <i>importance sampling</i> ) . . . . .	20
2.2.4	Approximation des nœuds fixés : <i>Fixed-Node Diffusion Monte Carlo</i> . . . . .	21
2.2.5	Aspects pratiques de l'algorithme FN-DMC . . . . .	22
2.3	Connections entre les deux mondes . . . . .	26
2.3.1	Paradigme CIPSI comme fonction d'onde d'essai .	26
2.3.2	Étude QMC-CIPSI des atomes de transition de la série 3d . . . . .	29
2.3.3	Calcul des différences d'énergie et problème de la compensation des erreurs des nœuds-fixés . . . . .	34
2.4	Conclusion : le meilleur des deux mondes . . . . .	40

### 2.1 MÉTHODES D'INTERACTION DE CONFIGURATION

#### 2.1.1 Formulation mathématique

Les différentes variantes des méthodes d'IC reposent sur le développement de la fonction d'onde électronique sur une base de déterminants dits de Slater. Chaque déterminant est construit comme le produit antisymétrisé de fonctions à un corps appelées orbitales moléculaires (OMs). À

son tour, chaque OM est développée sur une base de fonctions élémentaires dites *fonctions de base atomiques*. Mathématiquement, la fonction d'onde  $\Psi$  s'écrit

$$\Psi = \sum_I c_I D_I, \quad (1a)$$

avec

$$D_I = \det(\Phi_{i_1} \dots \Phi_{i_N}), \quad (1b)$$

où  $N$  est le nombre d'électrons,  $\Phi_k$  une des  $N_{\text{orb}}$  *spin-orbitales* utilisées

$$\Phi_k(\mathbf{r}, \sigma) = \phi_k(\mathbf{r})\sigma, \quad (1c)$$

$\sigma$  représentant le spin de l'électron,  $\sigma = \alpha$  ou  $\beta$ , et  $\phi_k(\mathbf{r})$  la partie spatiale de la spin-orbitale. Cette partie spatiale est développée sur les  $N_{\text{basis}}$  fonctions de base atomiques

$$\phi_k(\mathbf{r}) = \sum_{i=1}^{N_{\text{basis}}} c_{ki} \chi_i(\mathbf{r}). \quad (1d)$$

Dans l'équation 1b, l'indice  $I$  regroupe l'ensemble des indices des orbitales moléculaires apparaissant dans ce déterminant  $I = (i_1, \dots, i_N)$ . Par exemple, dans le cas d'un système à 4 orbitales moléculaires et 2 électrons, les déterminants décrivant un système où les électrons occupent la première et deuxième OM puis la première et la quatrième peuvent donc s'écrire respectivement comme :

$$\begin{aligned} |1100\rangle &= \begin{vmatrix} \Phi_1(\mathbf{x}_1) & \Phi_2(\mathbf{x}_1) \\ \Phi_1(\mathbf{x}_2) & \Phi_2(\mathbf{x}_2) \end{vmatrix} \\ |1001\rangle &= \begin{vmatrix} \Phi_1(\mathbf{x}_1) & \Phi_4(\mathbf{x}_1) \\ \Phi_1(\mathbf{x}_2) & \Phi_4(\mathbf{x}_2) \end{vmatrix} \end{aligned}$$

où  $\mathbf{x} = (\mathbf{r}, \sigma)$ .  $|n_1 n_2 n_3 n_4\rangle$  (avec  $n_i$  valant 0 ou 1 et représentant le nombre d'occupation de l'orbitale  $i$ ) est une notation commode pour représenter la nature du déterminant.

Notons que si la base atomique est *complète*<sup>1</sup> et que l'on engendre tous les déterminants possibles, autrement dit toutes les distributions possibles des électrons parmi toutes les orbitales, on peut alors construire une matrice hamiltonienne qui, une fois diagonalisée, nous donne la *solution exacte de l'équation de Schrödinger*.

---

1. Dans l'espace  $\mathbb{R}^3$ .

### 2.1.2 Base finie : *Full-CI*

Malheureusement, une base complète contient nécessairement un nombre infini de fonctions de base. En pratique, on se place donc dans un sous-espace de taille finie engendré par la base atomique finie. Lorsque que toutes les répartitions possibles parmi toutes les orbitales sont prises en compte, on parle de méthode d'IC complète (*Full-CI*), il s'agit de la solution exacte de l'équation de Schrödinger dans une base finie donnée.

#### 2.1.2.1 Type de base

Une très grande variété de bases atomiques ont été introduites dans la littérature<sup>2</sup> afin d'approcher au mieux la complétude de l'espace à une particule. Dans la très grande majorité des cas, les fonctions de base utilisées sont écrites comme une somme de fonctions gaussiennes ( $e^{-\gamma_k r^2}$ ) multipliées par des polynômes en  $x, y$  et  $z$ ; on parle alors de **Gaussian Type Orbitals** (GTOs). Dans les applications présentées dans ce travail, les bases de gaussiennes seront typiquement celles introduites par DUNNING & H. [52] qui sont connues sous le nom de base *cc-pVnZ*. Ici,  $n$  est un nombre entier supérieur ou égal à 2 dit *cardinal* qui détermine la taille de la base; il existe donc des bases double-zeta *cc-pVDZ* ( $n=2$ ), triple-zeta *cc-pVTZ* ( $n=3$ ), et ainsi de suite.

Dans de plus rares cas, on peut également utiliser des fonctions de base de type exponentielle  $e^{-\gamma_k r}$ ; on parle alors de **Slater Type Orbitals** (STOs). Une application de ce type de base, physiquement mieux adaptée, mais mathématiquement plus difficile à manipuler, sera présentée dans la suite.

**ERREUR D'INCOMPLÉTUDE DE BASE.** Afin de réduire l'erreur due à la taille finie de la base, on effectue en général des séries de calculs d'IC en utilisant des bases de plus en plus grandes. Mathématiquement, cela est justifié mais en pratique cela peut être délicat car l'erreur sur la fonction d'onde est très sensible à la nature de la base et en particulier au jeu d'exposants  $\gamma_k$  choisis. Si l'on veut pouvoir extrapoler les résultats, obtenus avec différentes bases de taille croissante, vers le résultat exact à base infinie, il faut être capable de construire des bases cohérentes<sup>3</sup>, ce qui est loin d'être évident; construire de telles bases a mobilisé de nombreux chercheurs dans le passé.

2. Voir par exemple, le site de l'*Environment Molecular Sciences Laboratory, Pacific Northwest National Laboratory, US Department of Energy (EMSL)* regroupant plusieurs centaines de bases [53].

3. Dites « bien équilibrées », *well-balanced*.

### 2.1.2.2 Nombre de déterminants engendrés

Comme mentionné plus haut, l'ensemble des déterminants peut être obtenu en considérant toutes les façons possibles de distribuer  $N$  électrons ( $N = N_\alpha + N_\beta$ ) dans  $M$  orbitales. Mathématiquement, cela s'écrit :

$$N_{\text{dets}} = \binom{M}{N_\alpha} \times \binom{M}{N_\beta} = \frac{M!}{N_\alpha!(M - N_\alpha)!} \frac{M!}{N_\beta!(M - N_\beta)!}. \quad (2)$$

Ce nombre de déterminants croît factoriellement avec le nombre d'électrons, rendant la méthode *Full-CI* utilisable en pratique seulement sur des systèmes très petits [54]. Une idée naturelle est de ne pas effectuer l'IC complète, mais de se restreindre à un sous-ensemble de déterminants.<sup>4</sup>

### 2.1.3 Troncature de l'espace des déterminants

Vu l'impossibilité pratique de réaliser le calcul *Full-CI* pour un système général, la question qui se pose maintenant est celle du choix du sous-espace de déterminants à utiliser. La nature de ce sous-espace définit les différents type de méthodes d'IC rencontrées.

**PAR DEGRÉ D'EXCITATION.** La façon la plus répandue pour engendrer les différents déterminants est celle dite *par degré d'excitation*. L'idée est de définir un déterminant de référence, typiquement le déterminant *Hartree-Fock*<sup>5</sup>. Ensuite, on engendre les déterminants en substituant les orbitales occupées du déterminant *Hartree-Fock* par des orbitales non occupées (aussi appelées *orbitales virtuelles*). Si l'on substitue une seule orbitale, on parle de simple excitation ; deux orbitales de double excitations *etc.* jusqu'aux  $N$ -excitations (où  $N$  est nombre d'électrons). Par exemple, dans notre exemple présenté plus haut nous avons fait une simple excitation de la 2<sup>e</sup> orbitale vers la 4<sup>e</sup>, transformant le déterminant *Hartree-Fock*  $|1100\rangle$  en  $|1001\rangle$ . Les méthodes dites *mono-référence* utilisent un ensemble fini d'excitations uniquement sur le déterminant *Hartree-Fock*. Les méthodes *multi-référence* (MR) quant à elles, appliquent des excitations sur un ensemble de déterminants, généralement choisis pour décrire correctement l'interaction entre plusieurs états quasi-dégénérés<sup>6</sup>.

Au niveau le plus bas, on peut restreindre l'espace des déterminants aux seuls déterminants ayant 0 ou 1 excitation (mono-excitations) par rapport à cette référence. On parle alors d'approche CIS (*Configuration In-*

4. On peut dire que l'approche *Full-CI* tronque l'espace des fonctions de base alors que l'IC tronque l'espace des déterminants.

5. Qui correspond au déterminant qui a le poids le plus fort dans la fonction d'onde et qui est construit en occupant les orbitales moléculaires d'énergies les plus basses.

6. Le choix de la seule référence *Hartree-Fock* est alors insatisfaisante.

*teraction using Single excitations*). On peut y rajouter les double-excitations, ce qui définit alors la méthode CISD. Si l'on augmente de manière systématique le degré maximum d'excitation<sup>7</sup>, on tend vers le *Full-CI*; ceci constitue donc une manière automatique de réduire l'erreur liée à la troncature de l'espace.

PAR SOUS-ENSEMBLE D'ORBITALES MOLÉCULAIRES. Une autre façon de tronquer l'espace est de considérer toutes les excitations possibles, comme dans le *Full-CI*, mais dans un sous-ensemble d'orbitales moléculaires<sup>8</sup>. Ce sous-ensemble d'orbitales moléculaires définit alors ce qui est appelé l'*espace actif* (*active space*) et la méthode est appelée *Complete Active Space CI* (CAS-CI). Ce type d'approche est bien adapté au cas de systèmes qui sont intrinsèquement mal décrits par le seul déterminant *Hartree-Fock*. On peut aller plus loin en combinant la résolution du CAS-CI avec l'optimisation des orbitales moléculaires par minimisation de l'énergie totale, on parle alors de méthode CAS-SCF<sup>9</sup>.

COMMENT CHOISIR ? Ce qui définit le choix d'une méthode est un compromis entre la précision recherchée et le coût du calcul qu'on est prêt à investir. Ce coût est essentiellement déterminé par la taille de l'espace de déterminants choisi. En effet, il faut faire le calcul de la matrice hamiltonienne ( $H_{IJ} = \langle \mathbf{D}_I | \mathcal{H} | \mathbf{D}_J \rangle$ ) et sa diagonalisation.

La matrice hamiltonienne étant symétrique, diagonale dominante, et de très grande taille<sup>10</sup> et considérant le fait qu'on ne s'intéresse qu'aux premiers vecteurs propres de plus basse énergie, on utilise le plus souvent pour sa diagonalisation l'algorithme de DAVIDSON [44] qui est une méthode de type KRYLOV avec préconditionnement.

#### 2.1.4 Notion d'espace de Hilbert sélectionné

Si l'on considère le cas usuel d'une approche IC où l'espace *Full-CI* est réduit en utilisant un degré d'excitation maximal par rapport à un petit ensemble de déterminants de référence, on a alors une croissance de l'effort numérique de la manière suivante :

$$N_{\text{dets}} = N_{\text{ref}} \times N_{\text{occ}}^d N_{\text{virt}}^d$$

7. CIS, CISD, CISDT, CISDTQ, ...

8. En général quelques orbitales moléculaires occupées de haute énergie et quelques orbitales virtuelles de basse énergie.

9. Si l'on combine les méthodes de type CAS-SCF avec des méthodes de type degré d'excitation (CISD, CISDTQ, ...) on parle alors de méthodes de type *Multi-Reference CI* (MR-CI).

10. Typiquement, jusqu'à  $\sim 10^9$ .



où  $d$  est le degré d'excitation maximal et  $N_{\text{ref}}$  est le nombre de déterminants de référence.

Pour les systèmes de quelques dizaines d'électrons avec des tailles de base raisonnables, on ne peut aller en pratique guère au-delà du CAS+SD. Pourtant, une observation fondamentale convient d'être faite : bien que le nombre de déterminants croisse très vite, la très grande majorité des déterminants ont un poids très faible<sup>11</sup> dans la fonction d'onde finale issue de l'étape de diagonalisation. En pratique il peut être judicieux d'abandonner la logique de construction *a priori* de sous-espaces, basés sur toutes les  $n$ -excitations possibles, et plutôt d'essayer de sélectionner les déterminants importants au cours du calcul. Ceci est l'objet de la section suivante. Notons que le fait qu'une fraction extrêmement petite de déterminants contribue à la fonction d'onde d'IC apparaît chez de nombreux auteurs dès les années 1960 ; par exemple, CLAVERIE *et al.* [36], GERSHGORN & SHAVITT [65], et BENDER & DAVIDSON [16].

### 2.1.5 Sélection perturbative de l'espace des déterminants

**HISTORIQUE.** L'idée-clef est de sélectionner les déterminants de manière itérative en commençant par le déterminant *Hartree-Fock*, puis de chercher l'ensemble des déterminants qui contribuent le plus énergétiquement à la fonction d'onde. Très rapidement le nombre de déterminants sélectionnés augmente et ajouter un nouveau déterminant correspond à une petite modification de la fonction d'onde courante, il est donc naturel d'utiliser un schéma perturbatif pour le calcul la contribution énergétique. On peut citer les travaux de WHITTEN & HACKMEYER [124], HACKMEYER & WHITTEN [72], et ceux de BUENKER & PEYERIMHOFF [24, 25].

Dans ce travail, l'approche présentée est essentiellement celle développée par MALRIEU *et al.* sous le nom de méthode d'interaction de configuration par sélection perturbative, *Configuration Interaction using Perturbative Selection done Iteratively* (CIPSI) [55, 75] ; méthode qui a été également développée en Italie par PERSICO, CIMIRIGLIA et ANGELI [4, 5, 35]<sup>12</sup>.

**ALGORITHME CIPSI.** Décrivons maintenant l'algorithme CIPSI utilisé. En quelques mots l'algorithme a la structure suivante :

**ÉTAPE 1 (DIAGONALISATION)** On dispose à l'itération  $n$  d'un espace de référence (dit variationnel)  $\mathcal{V}_n$  composé d'un ensemble de déterminants. À la première itération ( $n = 0$ ) on débute avec un seul déterminant (en général le déterminant *Hartree-Fock*) ou un ensemble

11. Il s'agit des coefficients  $c_I$  dans l'équation 1a.

12. Ils ont particulièrement travaillé sur l'introduction de contributions du troisième ordre ainsi que sur des techniques diagrammatiques pour des calculs CIPSI beaucoup plus performants.

réduit de déterminants. On diagonalise la matrice  $\mathcal{H}$  dans cet espace et on obtient l'énergie variationnelle  $E_0(\mathcal{V}_n)$  du système ainsi que l'état propre :

$$|\Psi_0(\mathcal{V}_n)\rangle = \sum_{i \in \mathcal{V}_n} c_i^n |D_i\rangle \quad (3a)$$

ÉTAPE 2 (GÉNÉRATION) Un ensemble de déterminants  $|D_{i_c}\rangle$  qui n'appartiennent pas à  $\mathcal{V}_n$  sont engendrés (on appellera cet espace  $\mathcal{P}$ ) Ces déterminants sont obtenus en appliquant la matrice  $\mathcal{H}$  sur le vecteur fondamental courant  $|\Psi_0(\mathcal{V}_n)\rangle$  vérifiant

$$\langle \Psi_0(\mathcal{V}_n) | \mathcal{H} | D_{i_c} \rangle \neq 0 \quad (3b)$$

ÉTAPE 3 (PERTURBATION) On calcule la contribution énergétique au second-ordre de perturbation associée à chaque déterminant connecté :

$$\delta e(|D_{i_c}\rangle) = - \frac{\langle \Psi_0(\mathcal{V}_n) | \mathcal{H} | D_{i_c} \rangle^2}{\langle D_{i_c} | \mathcal{H} | D_{i_c} \rangle - E_0(\mathcal{V}_n)} \quad (3c)$$

ÉTAPE 4 (SÉLECTION) On sélectionne le (ou les) déterminant(s)  $|D_{i_c}^*\rangle$  associé(s) à la plus grande contribution  $|\delta e|$  à l'espace de référence  $\mathcal{V}_n$  (ou ceux dont la contribution est grande et proche du maximum avec un seuil donné)

$$\mathcal{V}_n \rightarrow \mathcal{V}_{n+1} = \mathcal{V}_n \cup \{|D_{i_c}^*\rangle\} \quad (3d)$$

ÉTAPE 5 (ITÉRATION) On revient à l'étape 1.

CONDITION D'ARRÊT Il existe plusieurs conditions d'arrêt possibles. On peut choisir de s'arrêter quand le nombre de déterminants de l'espace variationnel atteint une taille suffisante, dans ce cas le test se fait juste à l'étape 1 de diagonalisation ; ou bien sur un critère énergétique, par exemple si l'énergie perturbative est plus petite qu'un certain seuil, dans ce cas-là le test est effectué à l'étape 3.

On notera  $N_{\text{det}}$  le nombre final de déterminants dans  $\Psi_0$  et  $E_0$  l'énergie variationnelle correspondante. Au-delà de cette énergie variationnelle, il est possible d'obtenir une énergie totale plus précise en rajoutant la contribution énergétique totale calculée perturbativement au second-ordre, pour obtenir l'énergie totale dite **CIPSI** :

$$E_0(\text{CIPSI}) = E_0 + E_{\text{PT2}} = E_0 - \sum_{i \in M} \frac{\langle \Psi_0 | \mathcal{H} | D_i \rangle^2}{\langle D_i | \mathcal{H} | D_i \rangle - E_0}, \quad (3e)$$

où  $M$  dénote l'ensemble des déterminants n'appartenant pas à l'espace de référence et connectés au vecteur de référence  $|\Psi_0\rangle$  par  $\mathcal{H}$ ; soit les simples et doubles excitations.

**Difficulté algorithmique.** L'une des difficultés techniques la plus importante de cet algorithme se situe lors de l'étape de génération des déterminants en effet cet ensemble de déterminants doit être unique. Or la généalogie des déterminants est pour le moins incestueuse : un même déterminant peut être engendré par une quadruple excitation sur un déterminant de référence, ou bien encore par une double excitation sur un déterminant qui était un déterminant précédemment doublement excité. Ainsi, s'assurer de l'unicité des déterminants, ceci efficacement et en parallèle, est un vrai challenge.

Finalement le stockage même, de ces déterminants engendrés<sup>13</sup> se relève problématique ; ce nombre de déterminants peut rapidement devenir très important.

#### 2.1.6 Conclusion sur les méthodes d'IC

Nous avons présenté l'algorithme **CIPSI** qui permet de sélectionner perturbativement des déterminants dans un espace de référence. Notons que la notion de sélection est générale et qu'on peut utiliser cet algorithme, aussi bien pour effectuer un calcul CISD ou CAS-SCF sélectionné, que pour faire un *Full-CI* sélectionné.<sup>14</sup> Concernant les calculs *Full-CI*, nous avons vu qu'ils deviennent rapidement prohibitifs dès que le système croît à cause de l'explosion factorielle de la taille de l'espace de Hilbert. À partir de quelques milliards de déterminants, il n'est même plus possible de stocker les vecteurs nécessaires à la création de la matrice hamiltonienne et *a fortiori* sa diagonalisation ; ceci représente actuellement les limites pratiques du *Full-CI*.<sup>15</sup>

**AU-DELÀ DU *Full-CI* DÉTERMINISTE.** Afin de dépasser ces limitations il est possible d'abandonner l'aspect *déterministe* de l'algorithme, c'est à dire le fait que le produit matrice-vecteur  $(\sum_j H_{ij}c_j)$  au cœur de l'algorithme soit évalué de manière exacte<sup>16</sup>, au profit d'un échantillonnage probabiliste des produits élémentaires  $H_{ij}c_j$ <sup>17</sup>. Cette idée a été très récemment proposée et développée par le groupe d'ALAVI *et al.* sous le

13. L'espace  $\mathcal{P}$ .

14. Dans la suite de cette thèse, si aucun espace n'est précisé lors de l'utilisation de l'algorithme **CIPSI**, il s'agit de l'espace *Full-CI*.

15. On peut avoir une bonne approximation de cette énergie grâce à l'énergie perturbative du **CIPSI**, mais on ne peut en avoir l'énergie variationnelle.

16. Comme la somme sur *toutes* les coordonnées du vecteur.

17. Produits effectués uniquement quand le coefficient  $c_j$  est important, le carré  $c_j^2$  normalisé représentant la probabilité dans l'algorithme d'obtenir l'état  $j$ .

nom de méthode d'Interaction de Configuration Complète par Quantum Monte Carlo (FULL-CI QMC), voir [18, 19, 37–39]

Sans entrer dans les détails de cette approche, on peut en donner l'idée principale en disant qu'il s'agit d'une approche assez similaire de la méthode CIPSI mais où l'étape de sélection est faite de manière probabiliste. Pour chaque déterminant l'ensemble des déterminants qui lui sont connectés par  $\mathcal{H}$  est considéré et un déterminant parmi ceux-là est choisi aléatoirement avec une probabilité reliée à l'élément de matrice de  $\mathcal{H}$  entre déterminants. Pour une description détaillée de l'algorithme je renvoie les lecteurs aux références originales citées plus haut. Notons que dans cette méthode, les propriétés recherchées peuvent être obtenues comme une simple moyenne à partir de l'échantillonnage de l'espace des déterminants. Il n'y a plus de vecteurs de dimension colossale à stocker et le calcul peut être découpé à loisir sur des cœurs de calcul *indépendants*. Il s'agit donc d'un algorithme très bien adapté aux nouvelles architectures massivement parallèles. Remarquons enfin que cette méthode qui a été proposée très récemment, est emblématique du type de démarche au cœur de mon travail, à savoir la nécessité de trouver des nouveaux algorithmes adaptés aux nouvelles architectures massivement parallèles.

Dans la section suivante je présente le deuxième type d'approche que j'ai utilisée : les approches *Quantum Monte Carlo*, Monte Carlo quantique (QMC). Remarquablement, ce type de méthode peut également être vu comme une approche de type *Full-CI*. De plus, comme le FULL-CI QMC il s'agit d'une approche stochastique où la sélection est faite de manière probabiliste. La différence essentielle réside cependant dans le fait qu'au lieu de considérer une base de déterminants construits sur des orbitales moléculaires, une autre base, celle des positions des électrons – beaucoup plus grande – est utilisée dans les approches QMC.

## 2.2 MÉTHODES MONTE-CARLO QUANTIQUE (QMC)

Method	Representation	State of $\Psi$	Unknown
IC	Determinants : $ D_I\rangle = \det(\Phi_{i_1}, \dots, \Phi_{i_N})$	Finite sum : $\Psi = \sum_I c_I  I\rangle$	Coef : $c_I$
QMC	Electronic positions : $ \mathbf{r}_1, \dots, \mathbf{r}_N\rangle = \delta(\mathbf{r} - \mathbf{r}_1) \dots \delta(\mathbf{r} - \mathbf{r}_N)$	Infinite sum (integral) : $\Psi = \int d\mathbf{r}'_1 \dots d\mathbf{r}'_N \Psi(\mathbf{r}'_1, \dots, \mathbf{r}'_N) \delta(\mathbf{r}_1 - \mathbf{r}'_1) \dots \delta(\mathbf{r}_N - \mathbf{r}'_N)$	Wave function : $\Psi(\mathbf{r}'_1, \dots, \mathbf{r}'_N)$

TABLE 3 – Comparison between the representations used in Configuration Interaction and *Quantum Monte Carlo*, Monte Carlo quantique (QMC)

### 2.2.1 Base des positions des électrons

Au contraire des méthodes d'IC qui représentent les fonctions d'onde électroniques sur un ensemble *fini* de déterminants, les méthodes Monte-Carlo quantique – QMC – utilisent la base des positions des électrons.

Ici dans la base de Dirac c'est la fonction d'onde  $\Psi(\mathbf{r}_1, \mathbf{r}_2, \dots, \mathbf{r}_N)$  qu'il s'agit de déterminer. Ces idées sont résumées dans le Tableau 3. Deux remarques me semble importantes :

1. Dans les méthodes d'IC la base utilisée est tronquée, il existe donc une erreur systématique même quand toutes les excitations sont considérées. En revanche, dans les méthodes QMC la base est complète<sup>18</sup>. Lorsque les estimateurs probabilistes ont convergé pour un nombre suffisamment grand de pas Monte Carlo, on dispose donc *a priori* de la solution exacte de l'équation de Schrödinger avec une barre d'erreur liée au caractère fini de l'échantillonnage.
2. Contrairement à la base IC, la base QMC n'est pas antisymétrisée par rapport aux échanges des électrons, il faut donc trouver un moyen pour antisymétriser la fonction d'onde spatiale. Nous reviendrons sur ce point important plus loin.

### 2.2.2 Diffusion Monte Carlo (DMC) sans échantillonnage selon l'importance (*no importance sampling*)

Un algorithme QMC peut être vu comme un algorithme itératif revenant à appliquer à chaque itération un ensemble de règles stochastiques pour une population de répliques du système. Comme nous travaillons dans l'espace des positions, une réplique est définie ici comme l'ensemble des positions des électrons représentées par le vecteur  $\mathbf{R} = (\mathbf{r}_1, \mathbf{r}_2, \dots, \mathbf{r}_N)$ . Dans la communauté QMC, les répliques sont appelées « marcheurs » (*walkers*).

Sous sa forme la plus élémentaire la méthode Diffusion Monte Carlo (DMC, la variante QMC la plus populaire) permet d'engendrer des marcheurs qui, en moyenne, se répartissent avec la distribution dans l'espace de configurations données par la fonction d'onde exacte inconnue,  $\Phi_0$ . L'algorithme DMC est particulièrement simple. On peut le décomposer de la manière suivante :

1. On part d'une population de marcheurs  $\mathbf{R}_i$  *a priori* arbitraire<sup>19</sup> ;
2. Pour chaque marcheur, on effectue une diffusion libre<sup>20</sup> pour chaque électron. En pratique, cela revient à tirer une nouvelle position à

18. Toutes les positions possibles des électrons dans l'espace ordinaire sont considérées.

19. La position de chaque électron de chaque *walker* peut par exemple être tirée aléatoirement.

20. Aussi appelée mouvement *brownien*.

l'aide d'une gaussienne de largeur  $\tau$  qui correspond au pas de temps (*time-step*) choisi :

$$p(\mathbf{R} \rightarrow \mathbf{R}', \tau) = \frac{1}{(2\pi\tau)^{3N/2}} e^{-\frac{(\mathbf{R}-\mathbf{R}')^2}{2\tau}}, \quad (4a)$$

notons que cette loi gaussienne dans l'espace à  $3N$  dimensions est en fait le produit de  $3N$  gaussiennes à une dimension. On peut donc tirer la nouvelle position en tirant de manière *indépendante*  $3N$  nombres gaussiens à une dimension, ce qui peut être fait par exemple en utilisant l'algorithme de BOX-MULLER [21];

3. Une fois que tous les marcheurs ont bougé, on effectue l'étape dite de *branching*<sup>21</sup>. Chaque marcheur est éliminé, laissé intact ou répliqué un certain nombre de fois selon la valeur du potentiel qui lui est associé. Plus le potentiel est favorable – c'est-à-dire de valeur basse – plus le marcheur est répliqué un grand nombre de fois. Au contraire, dans les régions défavorables du point de vue du potentiel les marcheurs sont éliminés avec grande probabilité. D'un point de vue mathématique la règle est la suivante. Soit un marcheur en  $\mathbf{R}$  et soit  $m(\mathbf{R})$  le nombre de marcheurs après l'étape de *branching*. On prend :

$$m(\mathbf{R}) = \lfloor e^{-\tau V(\mathbf{R})} + \chi \rfloor, \quad (4b)$$

où  $\lfloor x \rfloor$  est la partie entière de  $x$  et  $\chi$  un nombre aléatoire uniforme sur  $[0, 1]$  de telle façon qu'en moyenne on ait :

$$\langle m \rangle = e^{-\tau V(\mathbf{R})}; \quad (4c)$$

4. On itère ensuite un grand nombre de fois ces deux étapes pour engendrer le plus grand nombre possible de positions de marcheurs et obtenir une distribution à grand temps.

On peut facilement montrer que cet algorithme stochastique revient à simuler l'équation de Schrödinger en *temps imaginaire* ( $t \rightarrow it$ ) suivante :

$$\frac{\partial \Phi(\mathbf{R}, t)}{\partial t} = \frac{1}{2} \nabla^2 \Phi(\mathbf{R}, t) - (V(\mathbf{R}) - E_T) \Phi(\mathbf{R}, t), \quad (5)$$

où  $\Phi(\mathbf{R}, t)$  représente la densité de marcheurs au point  $\mathbf{R}$  au temps  $t$  et  $E_T$  une énergie de référence arbitraire. Et il est important de remarquer qu'à grand temps, la densité stationnaire vérifiant :  $\frac{\partial \Phi(\mathbf{R}, t)}{\partial t} = 0$  est donnée par  $\Phi_0(\mathbf{R})$  – qui est la fonction d'onde exacte – pourvu que  $E_T$  soit égale à l'énergie exacte  $E_0$ <sup>22</sup> ! En effet, l'annulation du second membre

21. Ou processus de mort-naissance.

22. Cette valeur peut être ajustée dynamiquement lors de la simulation.

de l'Équation 5 est équivalent à écrire l'équation de Schrödinger stationnaire pour l'état fondamental.

Malheureusement ce type d'approche ne fonctionne pas du tout en pratique, car d'une part les marcheurs se promènent de manière uniforme dans l'espace de configurations rendant les simulations trop longues et que, d'autre part, les fluctuations du potentiel coulombien sont beaucoup trop importantes rendant la diminution des barres d'erreur très longue. On introduit afin de remédier à ces problèmes la notion d'échantillonnage selon l'importance.

### 2.2.3 Diffusion Monte Carlo avec échantillonnage selon l'importance (*importance sampling*)

Afin de réaliser un échantillonnage physique de l'espace de configurations, on introduit une fonction d'essai approchée  $\Psi_T$  pour représenter la fonction d'onde exacte inconnue. Maintenant au lieu de regarder l'évolution dans le temps de la densité  $\Phi(\mathbf{R}, t)^2$  on se focalise sur la densité mixte :

$$f(\mathbf{R}, t) \equiv \Psi_T(\mathbf{R})\Phi(\mathbf{R}, t) \quad (6a)$$

Après quelques manipulations algébriques sur l'Équation 5, on obtient l'équation d'évolution suivante pour  $f(\mathbf{R}, t)$  :

$$\frac{\partial f(\mathbf{R}, t)}{\partial t} = \frac{1}{2} \nabla^2 f(\mathbf{R}, t) - \nabla[\mathbf{b}(\mathbf{R})f(\mathbf{R}, t)] - (E_L(\mathbf{R}) - E_T)f(\mathbf{R}, t) \quad (6b)$$

Dans cette équation  $\mathbf{b}$  – appelé le « vecteur dérive » (*drift vector*) – est donné par :

$$\mathbf{b}(\mathbf{R}) = \frac{\nabla \Psi_T}{\Psi_T}, \quad (6c)$$

La quantité  $E_L(\mathbf{R})$  est appelée quand à elle « énergie locale » et est définie de la manière suivante :

$$E_L(\mathbf{R}) = \frac{H\Psi_T}{\Psi_T}, \quad (6d)$$

On note deux changements importants par rapport à l'équation sans *importance sampling* (Équation 5) :

1. Le premier concerne le fait que le terme de diffusion est complété par un terme  $(-\nabla(\mathbf{b}f))$  dit de dérive (*drift*) . On peut facilement se convaincre que ce terme a pour effet de transporter de manière déterministe la densité le long du vecteur *drift*. En pratique ce terme permet de pousser les marcheurs dans les directions de grande va-

leur de  $|\Psi_T|$ . Cela répond donc à la première difficulté concernant la nature de l'échantillonnage<sup>23</sup>.

2. Le second changement est que le potentiel  $V$  « nu » (*bare*) est transformé en un potentiel « renormalisé » (ou écranté) donné par l'énergie locale  $E_L(\mathbf{R})$ .

L'énergie locale joue un rôle fondamental dans toute approche QMC. Elle est homogène à une énergie et se réduit à une constante – l'énergie exacte – quand la fonction d'essai est égale à un des états propres du système. On voit ainsi que les fluctuations statistiques sont grandement réduites avec l'*importance sampling*.

Les étapes fondamentales de l'algorithme DMC utilisé ici sont donc :

- Une étape de diffusion avec *drift* ;
- Puis une étape de *branching* avec l'énergie locale.

Une fois atteint le régime stationnaire pour les marcheurs, la densité (Équation 6a) est donnée par :

$$f(\mathbf{R}, t) \equiv \Psi_T(\mathbf{R})\Phi_0(\mathbf{R}), \quad (7)$$

où  $\Phi_0$  est la fonction d'onde exacte.

Il est facile de vérifier que la moyenne de l'énergie locale donne alors un estimateur non biaisé de l'énergie exacte  $E_0$

$$E_0 = \frac{\langle \Phi_0 | H | \Psi_T \rangle}{\langle \Phi_0 | \Psi_T \rangle} = \langle \langle E_L(\mathbf{R}) \rangle \rangle_{\Psi_T \Phi_0}, \quad (8)$$

où la moyenne  $\langle \langle \dots \rangle \rangle_{\Psi_T \Phi_0}$  désigne la valeur moyenne Monte Carlo sur la densité stationnaire. Nous n'entrerons pas plus dans les détails ici<sup>24</sup> ; pour une description beaucoup plus détaillée des méthodes QMC on peut se reporter à [30, 59, 73, 120]

#### 2.2.4 Approximation des nœuds fixés : *Fixed-Node Diffusion Monte Carlo*

Une remarque importante concernant l'algorithme présenté dans la section précédente est que le vecteur dérive, Équation 6c, diverge aux endroits où la fonction d'onde d'essai s'annule. On appelle « hypersurface nodale » ou plus simplement « nœuds » (*nodes*) l'ensemble des configurations dans  $\mathbf{R}$  où survient cette annulation. Mathématiquement, on sait que les nœuds des fonctions d'onde [80] divisent l'espace total de dimension  $3N$  en sous-domaines nodaux de dimension  $3N$ . Les nœuds

23. Il s'agit du terme introduisant la notion d'*importance sampling*.

24. Contrôle de la population, diminution de l'erreur de pas fini, etc.



forment donc une variété de dimension  $(3N - 1)$  qui découpent l'espace de configuration. Comme conséquence de la nature « gradient » du vecteur dérive, les marcheurs près des nœuds subissent une « force » qui les éloignent des nœuds avec une intensité d'autant plus forte qu'ils sont proches de la surface nodale. En pratique, les marcheurs ne peuvent donc pas quitter pendant la simulation le sous-volume nodal où ils se trouvent au départ. La simulation QMC se décompose donc en un sous-ensemble de simulations *indépendantes* dans des volumes nodaux différents. Mathématiquement, cette situation revient à dire que l'équation de Schrödinger est résolue avec des conditions aux limites supplémentaires par rapport aux conditions standard, à savoir la détermination d'une fonction d'onde s'annulant aux nœuds de la fonction d'onde d'essai,  $\Psi_T$ .

Une autre remarque importante est le fait que les zéros des fonctions d'onde d'essai usuelles sont en général approchés et que donc la simulation *Fixed-Node Diffusion Monte Carlo* (FN-DMC) sera <sup>25</sup> biaisée. On parle alors d'approximation des nœuds fixés (*fixed-node approximation*). Les nœuds sont approchés parce qu'ils ne sont pas entièrement déterminés par l'antisymétrie de la fonction d'onde. Les nœuds dits d'échange résultent de la coalescence de deux électrons  $i$  et  $j$  de même spin :

$$\mathbf{r}_i = \mathbf{r}_j \quad (9)$$

Ils définissent donc une variété de dimension  $(3N - 3)$ . Ces nœuds sont de mesure nulle par rapport aux nœuds totaux qui définissent une variété de dimension  $3N - 1$ . L'essentiel de l'hypersurface dépend donc de la nature de la fonction d'onde utilisée.

## 2.2.5 Aspects pratiques de l'algorithme FN-DMC

### 2.2.5.1 Choix de la fonction d'onde d'essai

Au final un calcul DMC se réduit à l'évolution temporelle d'une population de marcheurs et à la prise de valeurs moyennes temporelles. La seule quantité réellement fondamentale de la méthode est la fonction d'essai. En effet, cette fonction d'essai détermine à la fois la quantité de calculs à réaliser pour une erreur statistique donnée <sup>26</sup> et la petitesse du biais *fixed-node* final qui dépend de la qualité des zéros de cette fonction. Une remarque importante est qu'on dispose d'une très grande liberté de choix pour cette fonction ; dans l'algorithme, la seule contrainte est d'être capable de calculer les valeurs de la fonction d'onde d'essai, ses dérivées premières – pour le calcul du *drift* – et son laplacien – pour le calcul de l'énergie locale – et ceci pour une configuration quelconque des

<sup>25</sup>. Légèrement.

<sup>26</sup>. Plus la fonction d'essai est bonne, plus le nombre de pas Monte Carlo à réaliser pour obtenir cette erreur sera petite.

électrons. Ainsi, à l'inverse des méthodes basées sur la fonction d'onde, il n'y a pas de calcul d'intégrales bielectroniques en très grand nombre à effectuer.

**FACTEUR DE JASTROW.** Cette liberté a évidemment été largement exploitée et de nombreuses formes pour la fonction d'onde ont été testées dans la littérature. Brièvement, l'une des plus populaire est la forme dite de Jastrow Slater qui a la forme suivante :

$$\Psi_T = e^{J(\mathbf{r}_1, \dots, \mathbf{r}_N)} \times \sum_{k=1}^{N_{\text{det}}} c_k \begin{vmatrix} \phi_{k_1^\alpha}(\mathbf{r}_1) & \dots & \phi_{k_1^\alpha}(\mathbf{r}_{N_\alpha}) \\ \vdots & \ddots & \vdots \\ \phi_{k_{N_\alpha}^\alpha}(\mathbf{r}_1) & \dots & \phi_{k_{N_\alpha}^\alpha}(\mathbf{r}_{N_\alpha}) \end{vmatrix} \begin{vmatrix} \phi_{k_1^\beta}(\mathbf{r}_{N_\alpha+1}) & \dots & \phi_{k_1^\beta}(\mathbf{r}_N) \\ \vdots & \ddots & \vdots \\ \phi_{k_{N_\beta}^\beta}(\mathbf{r}_{N_\alpha+1}) & \dots & \phi_{k_{N_\beta}^\beta}(\mathbf{r}_N) \end{vmatrix} \quad (10)$$

Dans cette expression le préfacteur  $e^J$  est appelé « facteur de Jastrow ». De nombreuses formes pour  $J$  ont été proposées. Une forme minimale typique est la suivante :

$$J(\mathbf{r}_1, \dots, \mathbf{r}_N) = \sum_{i < j} \frac{a_{\sigma_{ij}} r_{ij}}{1 + b_{\sigma_{ij}} r_{ij}} - \sum_a c_a \sum_i r_{ia}, \quad (11)$$

où :

- Les indices latins  $i$  et  $j$  sont utilisés pour indiquer les électrons et l'indice  $a$  pour les noyaux ;
- Les paramètres  $a_{\sigma_{ij}}$  sont introduits pour imposer la condition de *cusp* électron-électron, c'est à dire  $a_{\sigma_{ij}} = 1/2$  pour des électrons de spin opposés et  $a_{\sigma_{ij}} = 1/4$  pour des électrons de même spin ;
- Les paramètres  $b_{\sigma_{ij}}$  sont introduits pour prendre en compte l'écrantage de l'interaction électron-électron à grandes distances interélectroniques ;
- Les paramètres  $c_a$  pour introduire le réajustement de la densité électronique après que les termes  $r_{ij}$  ont été introduits dans la fonction d'onde.

Des formes plus sophistiquées peuvent être choisies et en général sont écrites sous la forme générique suivante<sup>27</sup>

$$J(\mathbf{r}_1, \dots, \mathbf{r}_N) = \sum_{i < j} v_{e-e}(r_{ij}) + \sum_a v_{e-n}(r_{ia}) + \sum_{i < j} \sum_a v_{e-e-n}(r_{ij}, r_{ia}, r_{ja}) \quad (12)$$

---

27. Voir, par exemple [111].

où les fonctions,  $v_{e-e}$ ,  $v_{e-n}$ , et  $v_{e-e-n}$  décrivent les corrélations électron-électron à deux corps, les corrélations électron-noyaux et les corrélations à 3 corps, électron-électron-noyaux, respectivement. Chaque fonction est en général développée à l'aide de fonctions élémentaires<sup>28</sup>. Par exemple, une forme implémentée dans nos programmes est la suivante :

$$J(\mathbf{r}_1, \dots, \mathbf{r}_N) = \frac{1}{2} \sum_{j \neq i} \sum_a \left[ \sigma_{ij}(\tilde{r}_{ij}) - p_a(\tilde{r}_{ia}) - p_a(\tilde{r}_{ja}) + g_{a\sigma_{ij}}^{(1)} \tilde{r}_{ia}^2 \tilde{r}_{ja}^2 + g_{a\sigma_{ij}}^{(2)} (\tilde{r}_{ia}^2 + \tilde{r}_{ja}^2) \tilde{r}_{ij}^2 \right], \quad (13a)$$

où  $\tilde{r}_{ia}$  et  $\tilde{r}_{ij}$  sont les distances électrons-noyaux et électrons-électrons renormalisées :

$$\tilde{r}_{ij} \equiv \frac{r_{ij}}{1 + b_{\sigma_{ij}} r_{ij}} \quad (13b)$$

$$\tilde{r}_{ia} \equiv \frac{r_{ia}}{1 + b_a r_{ia}} \quad (13c)$$

et les fonctions  $s_{\sigma_{ij}}(r)$  et  $p_a(r)$  sont exprimées de la manière suivante

$$s_{\sigma_{ij}}(r) = \frac{1}{N_{\text{nucl}}} (e_{\sigma_{ij}}^{(1)} r + e_{\sigma_{ij}}^{(2)} r^2 + e_{\sigma_{ij}}^{(3)} r^3 + e_{\sigma_{ij}}^{(4)} r^4) \quad (13d)$$

$$p_a(r) = \frac{1}{N_{\text{nucl}}} (f_a^{(1)} r + f_a^{(2)} r^2 + f_a^{(3)} r^3 + f_a^{(4)} r^4) \quad (13e)$$

Dans ces formules  $N_{\text{nucl}}$  est le nombre de noyaux et les quantités  $\{b_{\sigma_{ij}}, b_a, e_{\sigma_{ij}}^{(l)}, f_a^{(l)}, g_{a\sigma_{ij}}^{(l)}\}$  jouent le rôle de paramètres ; soit pour cet exemple,  $10 + 9N_{\text{nucl}}$  paramètres au total.

**Calculs des éléments.** Dans l'expression de la fonction d'onde d'essai, Équation 10, la partie déterminantale est une somme finie de  $N_{\text{det}}$  déterminants de Slater construits à l'aide d'orbitales moléculaires : les  $c_k$  étant les différents coefficients donnant les amplitudes des déterminants et  $N_\alpha$ ,  $N_\beta$  représentent respectivement le nombre d'électrons  $\alpha$  et  $\beta$ . Cette partie est obtenue par un calcul *Density Functional Theory* (DFT) standard ou par une méthode *ab initio* basée sur la fonction d'onde.<sup>29</sup>

Les différentes orbitales moléculaires  $\phi_{k\sigma}(\sigma = \alpha, \beta)$  sont quant à elles extraites d'un ensemble d'orbitales dites actives,  $\phi_1(\mathbf{r}), \dots, \phi_{N_{\text{act}}}(\mathbf{r})$ . Notons que le formalisme utilisé dans les calculs Monte Carlo quantique est un formalisme sans spin [91]. La fonction d'onde dépend uniquement des coordonnées d'espace,  $\mathbf{r}_i$  et les orbitales moléculaires utilisées sont des orbitales spatiales ne dépendant que des 3 coordonnées spatiales ordinaires (x, y, et z). Pour obtenir une fonction en accord avec le

28. Polynômes, termes de Padé, somme d'exponentielles, etc.

29. *Hartree-Fock*, CASSCF, petite IC, etc.

principe de Pauli, l'antisymétrie de la fonction d'onde sous l'échange des coordonnées spatiales d'une paire arbitraire de même spin doit être imposée [91]. Ici, cette propriété est vérifiée parce que le terme de Jastrow global est par construction totalement symétrique sous l'échange des paires d'électrons et que le produit de deux déterminants de Slater, un pour les électrons  $\sigma = \alpha$  et un autre pour les électrons  $\sigma = \beta$  réalise l'antisymétrie.

FORMES ALTERNATIVES. Une grande variété de formes alternatives visant à mieux décrire la fonction d'onde exacte ont été introduites. Sans entrer dans les détails on peut citer :

- les fonctions d'onde de type géminal de CASULA *et al.* [32];
- la fonction d'onde sous forme d'un Pfaffien de BAJDICH *et al.* [14];
- la fonction d'onde avec terme de *backflow* de RIOS *et al.* [103];
- les formes *Valence Bond* (GVB) de ANDERSON & GODDARD III [2];
- la version *linear scaling* de FRACCHIA *et al.* [60];
- la forme *Jastrow-valence-bond* de BRAÏDA *et al.* [22];
- ou encore la forme *multi-Jastrow* de BOUABÇA *et al.* [20].

### 2.2.5.2 Optimisation de la fonction d'onde

Une fois la forme de la fonction d'onde choisie, ses différents paramètres<sup>30</sup> sont en général optimisés. Le critère utilisé peut être la minimisation de l'énergie variationnelle associée à la fonction d'onde ( $\langle \Psi_T | H | \Psi_T \rangle$ ) sa variance ( $\langle \Psi_T | H^2 | \Psi_T \rangle - \langle \Psi_T | H | \Psi_T \rangle^2$ ), ou encore une combinaison des deux. Noter qu'optimiser des centaines ou milliers de paramètres linéaires et non-linéaires dans un contexte où la quantité à minimiser est calculée de manière stochastique est loin d'être facile ; entre autre à cause de la présence du bruit statistique. Plusieurs solutions ont été proposées, citons simplement ici la méthode récente de UMRIGAR *et al.* [122] et les différentes approches mentionnées dans ce travail.

### 2.2.5.3 Coût numérique

Comme nous l'avons vu à chaque pas Monte Carlo, le coût numérique est essentiellement donné par le calcul de la fonction d'essai, de ses dérivées premières et de son laplacien qui dépendent de l'ensemble des positions des électrons. Le nombre total de pas étant très grand<sup>31</sup>, le calcul de ces quantités doit être évidemment très rapide. C'est en pratique ce qui restreint le choix des fonctions d'onde d'essai à des formes assez compactes. Une fois choisie la fonction d'onde, nous avons alors un

30. Paramètres du Jastrow, coefficients des déterminants, coefficients des orbitales moléculaires, exposants des fonctions de base, *etc.*

31. Typiquement quelques centaines de millions ou quelques milliards.

temps élémentaire pour le calcul des dérivées à chaque pas Monte Carlo. La convergence des moyennes Monte Carlo telle que celle de l'énergie, Équation 8, obéit alors aux lois des calculs probabilistes. Le théorème fondamental au cœur des calculs est le théorème de la limite centrale qui indique que la distribution statistique de « blocs » suffisamment grands représentant des sommes partielles de la quantité à moyenner<sup>32</sup> doit avoir une forme gaussienne quand on a atteint le régime stationnaire.

Une fois atteint ce régime on peut évaluer l'erreur statistique à partir de la largeur de cette distribution gaussienne : deux facteurs limitent la précision des calculs :

1. Le fait que les positions successives des électrons sont très corrélées lors de la simulation. Cela veut dire que le nombre de configurations réellement indépendantes est beaucoup plus petit que le nombre réellement effectuées. Cet aspect est quantifié par le temps d'autocorrélation ( $T_c$ ) de la quantité ( $X$ ) moyennée ;
2. Le deuxième facteur est l'ampleur des variations de la quantité locale  $X$  lors de la simulation. Cet aspect est quantifié par la « variance de  $X$  », notée  $\sigma^2$  c'est à dire la valeur moyenne  $\langle\langle X^2 \rangle\rangle - \langle\langle X \rangle\rangle^2$ .  $T_c$  et  $\sigma^2$  peuvent être facilement estimées lors de la simulation ; l'erreur statistique  $\delta X$  sur la moyenne de  $X$  est alors donnée par :

$$\delta X \sim \frac{\sigma}{\sqrt{N/T_c}} \quad (14)$$

où  $N$  est le nombre de pas Monte Carlo effectués. Notons qu'il est possible d'utiliser d'autres estimateurs  $\tilde{X}$  pour la même moyenne à calculer, pourvu que  $\langle\langle \tilde{X} \rangle\rangle = \langle\langle X \rangle\rangle$ . On parle alors d'estimateurs améliorés (*improved estimators*) quand la variance  $\tilde{\sigma}$  est plus petite que  $\sigma$ , c'est-à-dire que les fluctuations statistiques de l'estimateur amélioré sont plus petites que celles de l'estimateur de départ pour un même nombre de pas Monte Carlo. Voir par exemple, ASSARAF & CAFFAREL [12, 13].

## 2.3 CONNECTIONS ENTRE LES DEUX MONDES : QMC AVEC FONCTIONS D'ONDE D'ESSAI D'IC

### 2.3.1 Paradigme CIPSI comme fonction d'onde d'essai

Les différentes sources d'erreur présentes lors d'un calcul QMC sont :

---

32. Par exemple, l'énergie locale.

- l’erreur statistique due à un nombre fini  $N_{MC}$  de pas Monte Carlo, cette erreur diminuant en  $\sim \frac{1}{\sqrt{N_{MC}}}$  ;
- l’erreur résultant de l’utilisation de nœuds fixés approchés (*fixed-node approximation*), autrement dit les zéros de la fonction d’onde ;
- l’erreur due à un pas de temps fini  $\tau$  (*time-step error*) ;
- l’erreur due à l’utilisation d’un nombre fini  $M$  de marcheurs<sup>33</sup> ;
- l’erreur due au générateur de nombres *pseudo*-aléatoires<sup>34</sup>.

L’erreur statistique peut être – au moins en théorie – réduite en augmentant le nombre de pas Monte Carlo ; l’erreur de pas temps fini peut être facilement contrôlée en effectuant plusieurs calculs à des pas de temps différents et en extrapolant les résultats pour un pas de temps nul ; de même l’erreur de population finie peut être contrôlée par extrapolation des résultats obtenus pour différentes tailles de population ; l’erreur due au générateur est considérée comme négligeable<sup>35</sup> pourvu que le générateur soit de qualité suffisante<sup>36</sup>.

**NŒUDS FIXÉS APPROCHÉS.** La seule erreur fondamentale de l’approche FN-DMC est l’erreur *fixed-node*. En effet, cette erreur est, tout comme l’erreur de pas et de population finis, une erreur systématique – un biais dans les résultats qui demeure pour une statistique infinie – mais, qui contrairement à ces dernières, ne peut pas être extrapolée à zéro facilement en variant un seul paramètre<sup>37</sup>. L’erreur des nœuds fixés résulte en effet de la forme approchée d’une hypersurface nodale de dimension  $3N - 1$  (où  $N$  est le nombre d’électrons) qui est d’une topologie très difficile à paramétriser.

La méthode usuelle pour diminuer cette erreur est de considérer des formes de fonctions d’onde d’essai possédant le meilleur contenu physique possible et ensuite d’optimiser au mieux l’énergie variationnelle ou la variance de l’hamiltonien (voir, section précédente sur l’optimisation de la fonction d’onde). Comme nous l’avons souligné précédemment, il s’agit d’un processus délicat puisqu’il s’agit d’optimiser un grand nombre de paramètres<sup>38</sup> dont la plupart sont non-linéaires dans un contexte où la fonction-objectif à minimiser est calculée de manière stochastique.

Dans ce travail de thèse, j’ai participé à la poursuite de l’exploration systématique des résultats obtenus en utilisant des fonctions d’onde de type IC comme fonction d’essai. Cette proposition originale récemment faite par notre groupe a été initiée lors du travail de thèse de GINER [66].

33. Cette erreur peut être absente dans certaines variantes de l’algorithme.

34. Voir [97] pour une comparaison de différents algorithmes.

35. Dans les barres d’erreur typiques des simulations.

36. Un générateur recommandé est par exemple celui de L’ECUYER [87].

37. Comme  $\tau$  ou  $M$  peuvent l’être.

38. Plusieurs centaines ou milliers

À première vue l'utilisation de fonctions d'IC en QMC n'est pas naturelle car contrairement aux fonctions d'onde habituellement utilisées dans la communauté, ce sont des fonctions d'onde très peu compactes qui se développent sur un nombre important de déterminants<sup>39</sup>. Il est donc à craindre que le calcul des dérivées premières et secondes de la fonction d'onde à chaque pas prennent en pratique un temps irréaliste ; c'est la raison principale pour laquelle ces fonctions n'avaient pas été utilisées jusqu'alors.

Deux idées importantes rendent en fait possible cette solution :

1. l'utilisation d'un algorithme de type IC sélectionné comme CIPSI qui permet de construire des fonctions d'onde d'IC beaucoup plus compactes – au sens du nombre de déterminants – en sélectionnant les déterminants les plus importants dans le développement ; au contraire des méthodes d'IC traditionnelles qui utilisent l'ensemble – très grand et défini *a priori* – des déterminants associés à des classes d'excitation<sup>40</sup>.
2. le développement d'un algorithme très performant du calculs des dérivées d'une somme d'un grand nombre de déterminants. Pour donner un exemple, dans le cas de l'atome de cuivre (29 électrons) le calcul pour 56 000 déterminants se fait en un temps comparable à celui d'un calcul d'une vingtaine de déterminants avec une approche directe sans amélioration. Cet algorithme sera présenté plus loin.

Il a été illustré sur un premier ensemble de systèmes que l'utilisation de nœuds des fonctions IC permet d'obtenir de très bons résultats [66–68, 107].

L'avantage premier de l'utilisation de ce type de fonction d'essai est la suppression totale de l'étape d'optimisation stochastique des approches habituelles : l'optimisation des coefficients des déterminants se fait au niveau déterministe par diagonalisation de la matrice hamiltonienne. Cette procédure est évidemment très simple, complètement robuste et aboutit à un jeu de coefficients unique<sup>41</sup>. Un second avantage est le fait que les nœuds varient continûment quand la géométrie du système moléculaire est variée, ce qui conduit à des surfaces de potentiel électroniques lisses, un résultat beaucoup plus difficile à obtenir en pratique quand les paramètres ont un caractère non-linéaire et sont optimisés de manière indépendante pour chaque géométrie dans un contexte stochastique.

Durant mon travail de thèse j'ai poursuivi l'exploration de la qualité des nœuds CIPSI dans deux cas :

39. Jusqu'à quelques millions de déterminants, voire plus

40. Toutes les simples, doubles, triples, *etc.* excitations

41. Contrairement au cas où des paramètres non-linéaires sont utilisés

- celui des atomes dits de transition de la troisième ligne du tableau périodique (Sc-Zn), à la fois avec des fonctions de base de type gaussiennes (GTOs) et de type Slater (STOs) [107];
- celui des 55 molécules de l'ensemble dit G2 de CURTISS *et al.* [42] avec des calculs tous-électrons ou avec des pseudo-potentiels.

### 2.3.2 Étude QMC-CIPSI des atomes de transition de la série 3d

Dans ce travail nous avons calculé les énergies totales tous-électrons non-relativistes des atomes de la série 3d : Sc-Zn par méthode QMC avec utilisation de fonctions d'essai CIPSI. Les résultats obtenus représentent les meilleures énergies variationnelles<sup>42</sup> publiées à ce jour pour ces atomes. Ils illustrent l'avantage de coupler les méthodes QMC et CIPSI. D'un point de vue pratique les calculs ont été effectués dans le cadre d'un Mésos-Challenge organisé par l'Equipex Equip@Meso sur la toute nouvelle machine Eos du méso-centre CALMIP. Les résultats sont publiés dans l'article disponible en annexe [107]. En vue de la réalisation de tels calculs j'ai implémenté dans le code Quantum Package les fonctions de base de type Slater STOs au lieu des traditionnelles bases gaussiennes GTOs. Ce type de fonctions de base permet grâce à un meilleur comportement à petites et grandes distances de rendre le développement en déterminants encore plus compact et de diminuer le nombre de fonctions de base élémentaires.

#### 2.3.2.1 Fonctions de base de type Slater

Traditionnellement, les fonctions de base utilisées pour les calculs avec fonctions d'onde sont de forme gaussienne. La raison principale est purement calculatoire. En effet, dans ce cas, les intégrales atomiques qui apparaissent dans les expressions des éléments de matrice de l'hamiltonien sont calculables analytiquement. Cependant, il serait plus logique et plus physique d'utiliser des fonctions de type exponentielle puisque ce sont ce type de fonctions qui apparaissent lors de la résolution de l'équation de Schrödinger; par exemple celle pour l'atome d'hydrogène. Ces fonctions de type exponentielle sont appelées en chimie quantique, fonctions de Slater (STOs); elles ont été introduites en 1930 par le physicien SLATER [114].

Les fonctions STOs possèdent deux propriétés remarquables que n'ont pas les fonctions gaussiennes GTOs :

- Une décroissance exponentielle à grande distance;
- La condition dite de *cusp*<sup>43</sup> pour une distance électron-noyau nulle.

42. Les plus basses.

43. Voir par exemple, KUTZELNIGG [85], KUTZELNIGG & KLOPPER [86]



Malheureusement, la non-séparabilité des fonctions de Slater<sup>44</sup> au contraire des fonctions gaussiennes, rend le calcul des – nombreuses – intégrales multicentriques très coûteux. Malgré cela, nous avons décidé d'implémenter les intégrales des fonctions de Slater à un centre afin de pouvoir les utiliser pour les métaux de transition.

IMPLEMENTATION ET VALIDATION. En pratique, le travail a été effectué en trois étapes :

1. Calcul analytique des intégrales monocentriques à 1-électron (intégrales à 3 dimensions) et 2-électrons (intégrales à 6 dimensions) pour une forme générale de la fonction de Slater (en coordonnées sphériques) donnée par

$$\phi_{nlm} = c_n Y_{lm}(\theta, \phi) r^n e^{-\gamma r}$$

où  $\gamma$  est l'exposant de la fonction de Slater,  $Y_{lm}$  les harmoniques sphériques sous forme réelle et  $c_n$  un coefficient de normalisation facile à évaluer.

2. Intégration dans le code `Quantum Package`; cette étape fut grandement facilitée grâce à l'outil `IRPF90` dont nous parlerons dans un prochain chapitre. À cette occasion, il a fallu également développer tout un environnement approprié. Nous avons stocké dans une base de données les fichiers de bases atomiques de la littérature; nous avons défini un format de lecture de ces fichiers. Enfin, une légère étape d'optimisation des routines `Fortran` fut nécessaire afin d'accélérer le calcul de ces intégrales.

3. Intégration dans le code `QMC=Chem`

Afin de valider notre implémentation nous avons comparé nos résultats au niveau *Hartree-Fock* avec ceux de BUNGE *et al.* [27] où l'on peut trouver non seulement les énergies *Hartree-Fock* mais aussi la définition exacte des bases de Slater utilisées et les coefficients des orbitales atomiques sur ces fonctions de base<sup>45</sup>. En pratique, nous sommes en accord avec ces résultats jusqu'à la sixième décimale après la virgule, ce qui valide notre implémentation. De plus dans un précédent papier [26], BUNGE & ESQUIVEL ont présenté le résultat obtenu pour l'énergie de l'atome de Néon d'un calcul d'IC comprenant les simples et double excitations (CISD) et tronqué pour cause de limitations informatiques. Ils trouvent une valeur de  $-128.880 E_h$ , alors que notre CISD complet donne une énergie de  $-128.8917 E_h$ . Cette énergie est cohérente – car légèrement plus basse – et renforce donc notre confiance dans la validité de notre implémentation.

44. Le produit de deux fonctions Slater non centrées sur le même noyau n'est pas une fonction Slater.

45. Ceci pour tous les atomes compris entre l'Hélium ( $Z = 2$ ) et le Xénon ( $Z = 54$ ).

### 2.3.2.2 *Calculs de référence pour les énergies totales des atomes de transition de la série 3d*

Notons que disposer de calculs de référence de grande précision pour les énergies totales de ces atomes n'est pas un pur exercice académique mais est aussi important en pratique. En effet, ces énergies<sup>46</sup> servent comme référence lors de la calibration des méthodes de la chimie quantique. On peut par exemple citer la recherche de meilleures fonctionnelles **DFT** afin de mieux décrire les termes d'échange et de corrélation, ou encore la quantification des effets de bases finis et ceux liés à l'erreur de troncature résultant de l'utilisation d'un degré d'excitation fini dans les méthodes **IC**. On peut également obtenir, en comparant ces énergies aux énergies expérimentales, des informations concernant les effets relativistes.

**CHOIX DES BASES UTILISÉES.** L'un des critères essentiel à un bon calcul d'**IC** est le choix de la base. On peut trouver dans la littérature des bases de fonctions Slater (par exemple, [27] ou [88]) mais malheureusement elles ne sont adaptées qu'à la description des orbitales moléculaires occupées. À notre connaissance il n'existe pas d'ensemble cohérent de fonctions de base permettant de faire des calculs d'**IC** où une bonne description de l'espace des orbitales virtuelles est nécessaire : il fut donc décidé de les construire. C'est à ce moment, que j'ai appris que la génération de bases équilibrées<sup>47</sup> tient plus du domaine de l'art que de la génération systématique. Une approche très pragmatique a donc été adoptée :

1. Un ensemble de calculs d'**IC** pour l'énergie de nos atomes avec plusieurs bases gaussiennes ont été effectués ;
2. La base donnant les meilleurs résultats, à savoir (ANO-R00S-aug-tz [99]) a été utilisée pour trouver les paramètres de nouvelles fonctions de Slater qui ont alors été rajoutées aux bases de BUNGE *et al.*
3. Puis les exposants de ces fonctions de base ont été optimisés grâce au programme NOMAD [46] en minimisant l'énergie **CIPSI** obtenue avec 10 000 déterminants sélectionnés.

Dans le **Tableau 4**, nous comparons les résultats obtenus pour l'énergie *Hartree-Fock* des atomes de la série 3d avec la base de BUNGE *et al.* et notre base optimisée. Nous montrons également l'énergie **CIPSI** à 10 000 déterminants. Une fois la base définie, nous pouvons réaliser les calculs des atomes proprement dit.

46. Qui sont tous-électrons non-relativistes et à noyau fixe.

47. Représentant bien l'espace à une particule.

Atom	HF Bunge	HF Our	Diff.	CIPSI
Sc [s <sup>2</sup> d <sup>1</sup> ]	-759.7364	-759.7376	-0.0012	-760.4398
Ti [s <sup>2</sup> d <sup>2</sup> ]	-848.3950	-848.3971	-0.0021	-849.1561
V [s <sup>2</sup> d <sup>3</sup> ]	-942.8843	-942.8866	-0.0023	-943.6874
Cr [s <sup>1</sup> d <sup>5</sup> ]	-1043.3563	-1043.3563	0.0000	-1044.2231
Mn [s <sup>2</sup> d <sup>5</sup> ]	-1149.8662	-1149.8662	0.0000	-1150.7714
Fe [s <sup>2</sup> d <sup>6</sup> ]	-1262.4466	-1262.4485	-0.0019	-1263.4309
Co [s <sup>2</sup> d <sup>7</sup> ]	-1381.4160	-1381.4030	0.0130	-1382.4677
Ni [s <sup>2</sup> d <sup>8</sup> ]	-1506.8602	-1506.8635	-0.0033	-1507.9958
Cu [s <sup>1</sup> d <sup>10</sup> ]	-1638.9637	-1638.9612	0.0025	-1640.2100
Zn [s <sup>2</sup> d <sup>10</sup> ]	-1777.8481	-1777.8480	0.0001	-1779.1058

TABLE 4 – Comparaison of both *Hartree-Fock* energies obtained using the Bunge basis set [27] and the optimized basis set obtained in this work. The CIPSI energy obtained in this work with 10 000 determinants is also reported. Energy in Hartree.

PROCÉDURE POUR LE CALCUL FN-DMC ET RESSOURCES INFORMATIQUES. La procédure systématique suivante a été utilisée pour le calcul des énergies FN-DMC :

1. Un calcul CIPSI sélectionnant 1 million de déterminants a été effectué ;
2. Puis le calcul QMC a été réalisé en conservant 99.5 % de la norme de la fonction d'onde CIPSI<sup>48</sup>.

Informatiquement, cela a été réalisé dans le cadre d'un Meso-Challenge organisé par l'*Equipex Equip@Meso* pendant lequel l'intégralité du nouveau supercalculateur Eos du mesocentre CALMIP, à 12 000 cœurs, était à notre disposition pendant 48 heures. Ce calcul a été réalisé grâce à notre code QMC=Chem. Dans une prochaine partie, nous expliciterons le type de parallélisme utilisé afin de tirer au maximum partie de l'ensemble des cœurs avec une efficacité quasi-maximale.

RÉSULTATS ET PERSPECTIVES. Les résultats que nous avons obtenus sont reportés dans le Tableau 5 et comparés aux résultats de référence de BUENDIA *et al.* [23]. On peut remarquer plusieurs points :

- Premièrement, l'énergie FN-DMC utilisant les nœuds de la fonction CIPSI est systématiquement plus basse que celle de la fonction *Hartree-Fock*. Ceci confirme bien l'intérêt à utiliser des fonctions d'IC.
- Deuxièmement, nos énergies sont plus basses que celles de BUENDIA *et al.* qui étaient jusqu'à alors les énergies les plus basses publiées. Nous retrouvons typiquement entre 92 % et 94 % de l'éner-

48. Soit entre 10 000 et 100 000 déterminants.

Atom <sup>a</sup>	<i>Hartree-Fock</i>		OEP <sup>b</sup>		CIPSI		FN en. gain <sup>c</sup>
Sc [s <sup>2</sup> d <sup>1</sup> ]	-760.5265(13)	[89.2(2)%]	-760.5288(6)	[89.50(7)%]	-760.5718(16)	[94.4(2)%]	-0.0453(21)
Ti [s <sup>2</sup> d <sup>2</sup> ]	-849.2405(14)	[89.6(2)%]	-849.2492(7)	[90.55(7)%]	-849.2841(19)	[94.2(2)%]	-0.0436(24)
V [s <sup>2</sup> d <sup>3</sup> ]	-943.7843(13)	[89.6(1)%]	-943.7882(6)	[89.95(6)%]	-943.8234(16)	[93.4(2)%]	-0.0391(21)
Cr [s <sup>1</sup> d <sup>5</sup> ]	-1044.3292(16)	[91.0(2)%]	-1044.3289(6)	[90.93(6)%]	-1044.3603(17)	[93.9(2)%]	-0.0311(23)
Mn [s <sup>2</sup> d <sup>5</sup> ]	-1150.8880(17)	[90.4(2)%]	-1150.8897(7)	[90.54(6)%]	-1150.9158(20)	[92.9(2)%]	-0.0278(26)
Fe [s <sup>2</sup> d <sup>6</sup> ]	-1263.5589(19)	[90.1(2)%]	-1263.5607(6)	[90.26(5)%]	-1263.5868(21)	[92.4(2)%]	-0.0279(28)
Co [s <sup>2</sup> d <sup>7</sup> ]	-1382.6177(21)	[90.5(2)%]	-1382.6216(8)	[90.85(6)%]	-1382.6377(24)	[92.1(2)%]	-0.0200(32)
Ni [s <sup>2</sup> d <sup>8</sup> ]	-1508.1645(23)	[91.6(2)%]	-1508.1743(7)	[92.27(5)%]	-1508.1901(25)	[93.4(2)%]	-0.0256(34)
Cu [s <sup>1</sup> d <sup>10</sup> ]	-1640.4271(26)	[92.4(2)%]	-1640.4266(7)	[92.34(4)%]	-1640.4328(29)	[92.7(2)%]	-0.0057(39)
Zn [s <sup>2</sup> d <sup>10</sup> ]	-1779.3371(26)	[91.9(2)%]	-1779.3425(8)	[92.24(5)%]	-1779.3386(31)	[92.0(2)%]	-0.0015(40)

TABLE 5 – FN-DMC total energies for the 3d series of transition metal atoms together with the percentage of correlation energy recovered – in square brackets – for different nodal structures. Energy in hartree.

- a. Atom given with its electronic configuration, the common argon core [Ar]=(1s<sup>2</sup>2s<sup>2</sup>2p<sup>6</sup>3s<sup>2</sup>3p<sup>6</sup>) being not shown.
- b. Ref. [23].
- c. Difference between FN-DMC energies obtained with Hartree-Fock nodes (column 2) and newly proposed CIPSI nodes (column 5).

gie de corrélation. Nous avons donc réussi à obtenir les énergies totales tous-électrons non-relativistes les plus basses de la littérature.

- Enfin, on peut noter que dans le cas des deux atomes les plus lourds (Cu et Zn) le gain FN-DMC obtenu en utilisant les nœuds CIPSI plutôt que les nœuds *Hartree-Fock* est négligeable. Dans ces deux cas, la distribution électronique n'est pas loin d'être sphérique et dans ce cas les nœuds *Hartree-Fock* sont de qualité comparables.

Je vois deux moyens pour améliorer ces résultats :

- La première est de prendre plus de déterminants pour le calcul FN-DMC. En effet, après ce papier publié en 2014, un effort particulier a été fait pour augmenter l'efficacité de nos calculs à grand nombre de déterminants<sup>49</sup>. Il serait donc maintenant possible d'augmenter le nombre de déterminants et donc la qualité des nœuds.
- Le deuxième point est en rapport avec la base utilisée : si l'on pouvait avoir accès à des bases plus grandes ou de meilleure qualité et construites de manière plus systématique ces résultats devraient pouvoir être améliorés.

49. Voir papier [106] disponible à l'Appendice D.

### 2.3.3 Calcul des différences d'énergie et problème de la compensation des erreurs des nœuds-fixés

Nous avons vu dans la partie précédente que rajouter des déterminants dans la fonction d'essai du FN-DMC en les sélectionnant perturbativement permet d'améliorer de manière systématique les nœuds<sup>50</sup>. Bien que les énergies totales absolues soient intéressantes (voir section précédente) il est bien connu que les quantités qui sont réellement pertinentes s'expriment comme des différences d'énergie par exemple :

- différences d'énergie entre niveaux d'énergie d'états excités<sup>51</sup> ;
- variation de l'énergie en fonction de la géométrie de la molécule<sup>52</sup>.

La problématique qui s'ouvre alors à nous est d'explorer la manière dont les erreurs sur les énergies totales se compensent – ou non – lorsqu'on évalue leurs différences. Le point fondamental en chimie quantique est que l'erreur absolue sur les énergies totales est, sauf exceptions pour des (très) petits systèmes, toujours plus grande ou beaucoup plus grande que les différences recherchées. En pratique, cela veut dire que pour avoir des résultats fiables, il faut toujours qu'il y ait une forme de compensation d'erreurs. Il est important de souligner que cela est vrai pour toute méthode de structure électronique<sup>53</sup> et pas seulement pour les approches QMC.

Dans un premier temps, nous présentons nos résultats concernant la compensation d'erreurs dans le cas du calcul des énergies d'atomisation, et dans un second temps nous présenterons nos travaux sur l'utilisation des pseudo-potentiels en QMC, qui par élimination des électrons de cœur peut également être compris comme une problématique de compensation d'erreur<sup>54</sup>.

#### 2.3.3.1 Calcul des énergies d'atomisation pour 55 molécules organiques : benchmark G2

L'ensemble dit G2 est un ensemble de molécules qui a été introduit en chimie quantique pour tester la qualité des méthodes électroniques [42]. Il est constitué de 55 molécules organiques comprenant des atomes de la première ligne (Li - F) et de la seconde (Na-Cl). Le problème nous concernant est relatif au calcul de l'énergie d'atomisation de l'ensemble de ces molécules. Rappelons que l'énergie d'atomisation d'une molécule est la différence d'énergie entre la molécule dans sa configuration

50. Ceci produit une diminution de l'énergie *fixed-node*.

51. Spectroscopie, photochimie, etc.

52. Barrières de réaction, surfaces d'énergie potentielle, etc.

53. Post-*Hartree-Fock*, DFT, etc.

54. La validité de l'approximation tenant au fait que l'on peut considérer les énergies des électrons de valence comme la différence entre l'énergie totale tous-électrons et celle des électrons de cœur.

Method	MAD	Det.
<i>Hartree-Fock</i>	90.42	1
CIPSI	55.45	1000
	32.85	10 000
<i>Full-CI</i> <sup>a</sup>	22.41	
FN-DMC <sup>b</sup>	19.6(19)	1000
	14.7(14)	1
	11.8(34)	

TABLE 6 – MAD (in kcal mol<sup>-1</sup>) of the atomization energies for different levels of all-electron calculations on a sub section of G2<sup>c</sup>. (cc-pVDZ basis set).

- a. With PT2 approximation ;
- b. The last calculation is done with the E<sub>PT2</sub> ratio technique ;
- c. H, Li, Be, C, N, O, F, Na, Si, P, S, Cl, Li<sub>2</sub>, F<sub>2</sub>, HCl, H<sub>2</sub>O<sub>2</sub>, N<sub>2</sub>H<sub>4</sub>, CS, SiO, ClF, P<sub>2</sub>, SO<sub>2</sub>, Cl<sub>2</sub>.

nucléaire d'équilibre et la somme des énergies totales des atomes séparés la constituant. Physiquement, il s'agit de l'énergie minimale nécessaire pour casser la molécule. Les énergies d'atomisation calculées par la méthode évaluée sont alors comparées aux énergies d'atomisation de référence qui sont obtenues à partir des énergies d'atomisation expérimentales auxquelles on enlève les contributions relativistes<sup>55</sup> et non purement électroniques<sup>56</sup>.

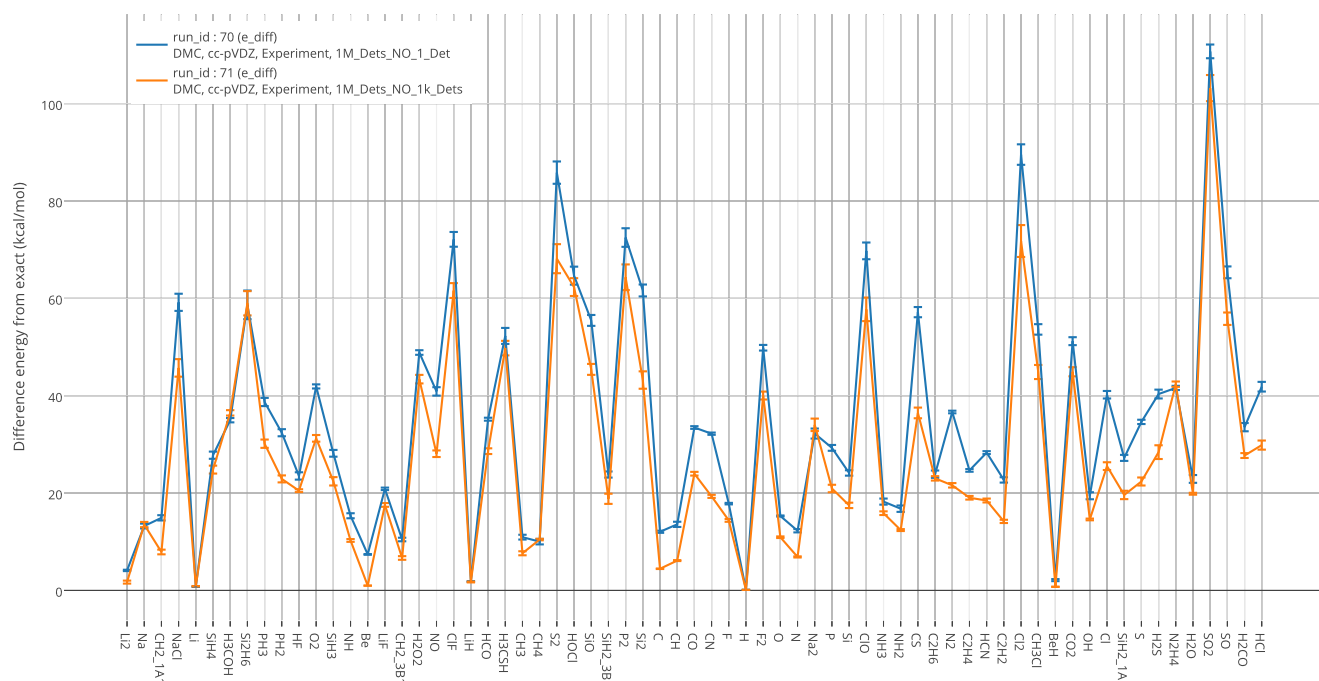
Afin de quantifier les erreurs on utilise traditionnellement le *Mean Absolute Deviation* (MAD) qui est calculé en faisant la moyenne sur les 55 molécules de la valeur absolue de la différence entre l'énergie d'atomisation calculée et celle de référence ; *Lower is better* comme disent les anglais.

Dans le Tableau 6, nos résultats concernant quelques MAD à différents niveaux de calcul sont présentés. Pour chaque calcul la (relativement petite) base cc-pVDZ est utilisée. Au niveau *Hartree-Fock*, le MAD est important et se situe autour de 90 kcal mol<sup>-1</sup>. Quand on effectue des calculs CIPSI avec un nombre croissant de déterminants, le MAD décroît continûment jusqu'à atteindre de l'ordre de 22.4 kcal mol<sup>-1</sup> dans le régime quasi-*Full-CI*.

Au niveau FN-DMC, la situation est différente. En utilisant les nœuds de la fonction *Hartree-Fock* mono-déterminantale on trouve un MAD de l'ordre de 15 kcal mol<sup>-1</sup>. Notons que ceci peut déjà être vu comme un bon résultat puisqu'on a une réduction de plus de 30% de la valeur obtenue par *Full-CI*. Cependant, à la différence des calculs IC, le MAD ne

55. Pour ces petits systèmes, essentiellement le couplage spin-orbite.

56. Les effets de masse finie et énergies de point-zéro.



**FIGURE 1** – Fixed-node FN-DMC energy differences in  $\text{kcal mol}^{-1}$  with the estimated exact non-relativistic values [57, 96] using either a mono-determinantal trial wave function (blue) or a 1000-determinant wave function (orange).

décroît pas forcément quand on augmente le nombre de déterminants sélectionnés. Par exemple, avec 1000 déterminants le MAD remonte à  $20 \text{ kcal mol}^{-1}$ . Le point très important à remarquer est que ceci est vrai bien que toutes les énergies totales *fixed-node* à 1000 déterminants obtenues pour les 55 molécules soient plus basses (ou égales) que les énergies totales *fixed-node* obtenues avec le seul déterminant *Hartree-Fock*. Ceci est illustré dans la figure Figure 1, où la différence des énergies FN-DMC des 55 molécules et des atomes constituant les molécules, avec l'énergie non-relativiste exacte estimée – avec une précision suffisante pour notre propos – est représentée. Il est clair que la courbe orange qui correspond aux calculs FN-DMC effectués avec 1000 déterminants est systématiquement plus basse<sup>57</sup> que la courbe bleue FN-DMC mono-déterminantale. Ce point important est maintenant discuté dans la section suivante.

AMÉLIORATION DES COMPENSATIONS D'ERREURS PAR UNE APPROCHE À  $E_{PT2}$  CONSTANTE. L'origine du résultat précédent est facile à déterminer, il s'agit d'un problème de mauvaise compensation d'erreurs due à la différence de qualité entre les nœuds utilisés pour la molécule et ceux

<sup>57</sup>. Ou égale dans les fluctuations.



des atomes séparés. En d'autres termes, prendre le même nombre de déterminants pour la molécule et ses atomes n'assure pas la cohérence des erreurs *fixed-node* entre le système et ses parties. On imagine bien que ce problème est général et est un des points importants des approches QMC appliquées au calcul des différences d'énergie. Ainsi, dans un travail précédent portant sur le calcul FN-DMC de la courbe de potentiel de la molécule  $F_2$  [67], GINER *et al.* ont rencontré un problème similaire : en fonction de la distance internucléaire, certaines régions sont plus ou moins difficiles à décrire avec un nombre donné de déterminants. Par exemple, à qualité de résultats égale, il faudra plus de déterminants pour décrire la région intermédiaire où la fonction d'onde perd son caractère ionique que dans la région d'équilibre. Une solution naturelle est donc de considérer dans le calcul QMC un nombre de déterminants variable en fonction de la distance. Évidemment se pose le choix du critère à utiliser pour déterminer quantitativement la dépendance de ce nombre le long de la courbe. La proposition a été de sélectionner, pour une distance donnée, le nombre de déterminants de la fonction d'onde par rapport à sa contribution énergétique totale calculée au second ordre, telle qu'elle apparaît dans l'Équation 3e. Cette quantité sera notée ici  $E_{PT_2}$ .

Ce choix est motivé par la raison suivante : l'expérience numérique montre que pour un nombre de déterminants suffisamment grand la somme de l'énergie variationnelle plus l'énergie  $E_{PT_2}$ , (Équation 3e) donne une approximation très raisonnable de l'énergie totale *Full-CI*<sup>58</sup>. Imposer en fonction de la distance une  $E_{PT_2}$  constante revient donc à construire une courbe d'énergie variationnelle presque parallèle à la courbe *Full-CI* ; c'est ce que nous appelons l'approche à  $E_{PT_2}$  constante. Nous avons vu qu'appliquée au calcul de la courbe de potentiel de  $F_2$  cette idée permet d'améliorer les résultats FN-DMC par rapport à ceux obtenus à nombre de déterminants constant[67]. Durant ma thèse j'ai repris cette idée pour le calcul des énergies d'atomisation *moléculaire* de l'ensemble G2. Illustrons cette idée sur un exemple concret, bien qu'imaginaire. Supposons qu'on cherche à calculer l'énergie d'atomisation de la molécule de thiotimoline [11]. On effectue d'abord un calcul *Full-CI*<sup>59</sup> pour déterminer l'énergie *Full-CI* de cette molécule, on trouve par exemple ( $E_{Th}^{ref} = 100$ ) ; on calcule également l'énergie des deux atomes la constituant (A, B) donnant respectivement  $E_A^{ref} = 40$  et  $E_B^{ref} = 10$ . Il nous faut maintenant construire nos fonctions d'essai pour le calcul QMC. On choisit tout d'abord un nombre de déterminants maximum pour la molécule de thiotimoline –typiquement quelques dizaines de milliers de déterminants. Supposons que l'énergie variationnelle obtenue soit alors de  $E_{Th}^{approx} = 80$ . Il nous faut donc déterminer l'énergie des atomes que l'on doit atteindre pour créer des fonctions d'onde d'es-

58. C'est à dire que les termes d'ordres supérieurs à 2 sont petits.

59. À la CIPSI.



sai qui conduisent à une énergie d'atomisation constante. La formule est la suivante :

$$E_{\text{atom}}^{\text{target}} = \left( 1 - \frac{E_{\text{mol}}^{\text{ref}} - E_{\text{mol}}^{\text{approx}}}{\sum E_{\text{atom}}^{\text{ref}}} \right) \times E_{\text{atom}}^{\text{ref}} \quad (15)$$

Ce qui nous donne ici

$$E_A^{\text{target}} = \left( 1 - \frac{100 - 80}{40 + 10} \right) \times 40 = 24$$

$$E_B^{\text{target}} = 6$$

On peut vérifier qu'en procédant de la sorte on garde la même énergie de corrélation  $100 - (40 + 10) = 50 \equiv 80 - (24 - 6)$ .

Ce principe a été utilisé et le résultat obtenu est présenté dans le [Tableau 6](#) pour le calcul FN-DMC de la dernière ligne : le MAD est bien amélioré puisqu'il tombe à une valeur approximativement égale à  $12 \text{ kcal mol}^{-1}$ . Malheureusement la baisse du MAD n'est pas flagrante.

### 2.3.3.2 Pseudo-potentiels

Les électrons de valence sont ceux qui participent aux liaisons chimiques, et donc sont ceux qui confèrent leurs propriétés chimiques aux molécules. Les électrons de cœur quant à eux ne jouent aucun rôle en ce qui concerne la réactivité chimique de basse énergie. Cependant, puisqu'ils sont très proches des noyaux ils subissent un potentiel attractif très fort et ont une grande énergie cinétique ; ce sont donc eux qui sont responsables de la plus grande partie de l'énergie totale électronique.

De plus, en QMC, si la fonction d'essai  $\Psi_T$  n'est pas la fonction d'onde exacte, l'énergie locale  $\frac{H\Psi_T}{\Psi_T}$  fluctue autour de sa moyenne avec une amplitude proportionnelle à l'énergie totale électronique. Plus l'énergie totale est grande en valeur absolue, plus ces fluctuations sont importantes. Les électrons de cœur sont donc des sources de fluctuations de l'énergie qui ralentissent considérablement et inutilement la convergence des calculs QMC. Pour supprimer cette source de fluctuations, les électrons de cœur ne sont plus traités explicitement, mais sont remplacés par un potentiel effectif qui reproduit en moyenne leur interaction avec les électrons de valence. Une charge effective est également attribuée aux noyaux.

**THÉORIE DES PSEUDO-POTENTIELS.** Nous avons utilisé les pseudo-potentiels développés par BURKATZKI *et al.* [28, 29] qui sont des pseudo-potentiels spécifiquement conçus pour les calculs QMC. En effet, ils possèdent un terme en  $-Z/r$  qui permet de supprimer la divergence du potentiel lorsqu'un électron s'approche du noyau. Ainsi, il n'est plus nécessaire de remplir la condition de *cusp* électron-noyau pour que la va-

Basis	MAD
cc-pVDZ <sup>a</sup>	83.63
cc-pVTZ <sup>a</sup>	74.32
vtz-BFD <sup>b</sup>	73.37

TABLE 7 – MAD (in kcal mol<sup>-1</sup>) of the atomization energies for *Hartree-Fock* FN-DMC level of calculation for a subset of molecules extracted from the G2 set <sup>c</sup> depending on the basis set used.

- a. All electrons ;
- b. With Pseudo-potentiel ;
- c. Na, Mg, Li, CH, Be, H<sub>2</sub>O<sub>2</sub>, ClF, Si, SiO, P<sub>2</sub>, C, B, F, H, F<sub>2</sub>, O, N, P, S, SO<sub>2</sub>, CS, Cl<sub>2</sub>, BeH, Cl, Al, N<sub>2</sub>H<sub>4</sub>, HCl.

riance de l'énergie locale soit finie, ce qui permet d'effectuer des calculs QMC en base d'orbitales gaussiennes bien mieux conditionnés.

Le potentiel additionnel est le suivant :

$$V_{\text{pseudo}}(\mathbf{r}) = \sum_a \left( \frac{Z - Z_a^{\text{eff}}}{|\mathbf{r} - \mathbf{R}_a|} + V_a^{\text{loc}}(\mathbf{r}) + V_a^{\text{non-loc}}(\mathbf{r}) \right) \quad (16a)$$

Pour chaque noyau  $a$  comportant un pseudo-potentiel, le premier terme correspond au remplacement de la charge nucléaire par une charge effective  $Z_{\text{eff}}$ , le deuxième terme  $V^{\text{loc}}$  est un potentiel local et  $V^{\text{non-loc}}$  est un potentiel non local :

$$V_a^{\text{loc}}(\mathbf{r}) = \sum_{k=1}^{k_{\text{max}}} \mathbf{r}^{k-2} v_k \exp(-\zeta_k |\mathbf{r} - \mathbf{R}_a|^2) \quad (16b)$$

$$V_a^{\text{non-loc}}(\mathbf{r}) = \sum_{k=1}^{k_{\text{max}}} \sum_{l=0}^{l_{\text{max}}} \mathbf{r}^{k-2} v_{kl} \exp(-\zeta_k |\mathbf{r} - \mathbf{R}_a|^2) \sum_{m=-l}^l |Y_{lm}\rangle \langle Y_{lm}| |\Psi\rangle \quad (16c)$$

IMPLÉMENTATION ET VÉRIFICATION. Le processus d'implémentation fut le même que pour les STO de la section précédente. L'implémentation a été validée en comparant nos résultats au niveau *Hartree-Fock* et CISD sur toutes les molécules du G2, entre notre programme Quantum Package et deux programmes faisant référence dans le milieu de la chimie quantique<sup>60</sup>, un accord à 10<sup>-8</sup> E<sub>h</sub> fut constaté.

Method	MAD	Det.
<i>Hartree-Fock</i>	63.86	1
CIPSI	39.32	10 000
<i>Full-CI</i> <sup>a</sup>	9.94	
FN-DMC <sup>b</sup>	8.20(128)	1 000 000
	4.27(206)	

TABLE 8 – MAD (in kcal mol<sup>-1</sup>) of the atomization energies for different levels of calculation on G2 with pseudo-potentials. (vtz-BDF basis set)

a. With PT2 approximation ;

b. The last calculation is done with the constant-E<sub>PT2</sub> technique ;

### 2.3.3.3 Résultats avec des pseudo-potentiels

On voit dans le tableau [Tableau 7](#) deux résultats importants. D'une part, le fait que l'augmentation de la base permet de diminuer le MAD, même au niveau FN-DMC, et d'autre part la bonne construction des bases adaptées au pseudo-potentiels. En effet, on a le même ordre de grandeur pour le MAD pour les bases cc-pVTZ et vtz-BFD : de l'ordre de 74 kcal mol<sup>-1</sup>. Les contributions énergétiques redondantes ont bien été enlevé de façon cohérente par les pseudo potentiels.

La réduction du nombre d'électrons permet une augmentation de la taille de la base pour un prix calculatoire constant, et il fut ainsi possible de réaliser des calculs FN-DMC en utilisant la base vtz-BDZ et un nombre important de déterminants. Les résultats sont montrés dans le tableau [Tableau 8](#). On voit que l'on a réussi à obtenir un MAD de l'ordre de 4 kcal mol<sup>-1</sup> en combinant l'augmentation de la base et la méthode à E<sub>PT2</sub> constante présentée plus haut.

## 2.4 CONCLUSION : LE MEILLEUR DES DEUX MONDES

En guise de conclusion à cette partie « scientifique », nous aimerions insister sur le fait que les calculs QMC peuvent être extrêmement précis, mais à deux conditions :

1. La première condition est que l'on puisse diminuer suffisamment l'erreur statistique en faisant suffisamment de pas Monte Carlo ; cela en pratique est rendu possible grâce à l'adaptation idéale du QMC au calcul massivement parallèle. Dans l'exemple discuté précédemment des atomes de la série 3d, les meilleures énergies pour ces atomes ont pu être obtenues grâce à l'utilisation de l'ensemble

60. GAUSSIAN [63] et GAMESS [71].

des 12 000 cœurs de calcul du supercalculateur de CALMIP sur une durée de 48h.

2. La deuxième condition est tout aussi importante. Il s'agit d'être capable de construire des nœuds pour la fonction d'onde d'essai approchée qui soient d'une qualité suffisante. C'est un problème hautement non-trivial que nous avons proposé d'attaquer en utilisant les nœuds de fonctions d'onde d'IC dont les déterminants sont sélectionnés sur un critère énergétique perturbatif. L'avantage d'une telle approche est qu'il n'est alors plus nécessaire d'avoir recours à l'étape d'optimisation stochastique des approches QMC traditionnelles ; nous pouvons nous reposer sur un processus d'optimisation déterministe qui permet d'obtenir de manière stable et automatique l'ensemble unique des coefficients optimisés. C'est un point qui nous paraît très important. Cette démarche peut donc être vue comme la mise en commun du meilleur du monde QMC et du monde *ab initio*.



# 3 | ALGORITHMIQUE & IMPLÉ- MENTATION

3.1	Processeurs et Parallélisation . . . . .	43
3.1.1	Architecture des processeurs . . . . .	44
3.1.2	Cœurs de calcul . . . . .	50
3.1.3	Distribution d'un calcul entre plusieurs cœurs . . .	54
3.2	Supercalculateurs et <i>cloud computing</i> . . . . .	63
3.2.1	Supercalculateurs . . . . .	64
3.2.2	Cloud computing . . . . .	64
3.3	Application : CIPSI & Quantum Package . . . . .	68
3.3.1	Adressage des intégrales . . . . .	68
3.3.2	Construction de la matrice hamiltonienne . . . . .	70
3.3.3	Problématique de l'unicité . . . . .	73
3.3.4	Parallélisation utilisant OpenMP . . . . .	74
3.3.5	Conclusion . . . . .	74
3.4	Application : QMC et QMC=Chem . . . . .	75
3.4.1	Optimisation du processus monocœur . . . . .	75
3.4.2	Parallélisation de type client-serveur . . . . .	80

## 3.1 PROCESSEURS ET PARALLÉLISATION

Dans un premier temps nous allons nous concentrer sur les processeurs. Cette section ne se veut pas une présentation ultra-technique, il s'agit au contraire d'expliquer le plus simplement possible le pourquoi des architectures et le comment de leur exploitation ; dans le but d'améliorer leur utilisation par les développeurs que nous sommes. Dans un deuxième temps, nous verrons les technologies utilisées afin de pouvoir bénéficier de la puissance de plusieurs processeurs simultanément.

DESSEIN INTELLIGENT. Il faut se rendre compte que l'évolution des processeurs a été gouvernée par un principe simple : la maximisation du nombre d'opérations flottantes par seconde (flops) mesurées par le *benchmark High Performance LINPACK (HPL)*. On peut en voir une manifestation dans l'évolution annuelle toujours exponentielle depuis l'année 1994 du nombre de flops mesuré par ce *benchmark* Figure 2a. En effet, la puissance d'un supercalculateur étant mesurée à l'aune de ce benchmark, les constructeurs ont tout fait pour améliorer leur score à ce test, afin d'en tirer un argument de vente. Cela ne serait pas un problème si ce *benchmark* reflétait l'utilisation réelle d'un vrai code de calcul ;

malheureusement, cela n'est pas le cas. En effet l'HPL consiste en la résolution en double précision d'un système linéaire de type  $AX = B$  où  $A$  et  $B$  sont des matrices denses et  $X$  est un vecteur. Le goulot d'étranglement est ici le produit de matrices.<sup>1</sup> Nous verrons donc que les processeurs (et les supercalculateurs) actuels ont *de facto* une architecture particulièrement bien adaptée aux produits de matrices denses. Cependant, les codes scientifiques ne faisant pas *seulement* des produits de matrices denses, le gain de performance des codes ne suit pas systématiquement les performances théoriques mesurées par les *benchmarks* des machines. Néanmoins, une bonne connaissance des stratégies développées par les constructeurs est nécessaire afin de pouvoir écrire du code efficace, même s'il ne s'agit pas de produits de matrices denses.

**Précision sur le produit de matrices.** Les caractéristiques du produit de matrices denses en double précision permettront de justifier certains choix technologiques pris par les constructeurs. Je me permets donc de les détailler ici.

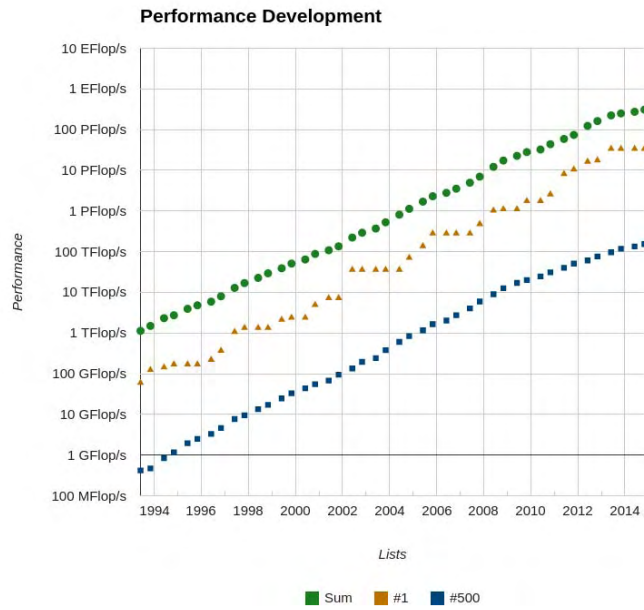
```
! To compute C = C + A*B
do i = 1:n
do j = 1:n
! load C(i,j)
    do k = 1:n
! load A(i,k) and B(k,j)
        C(i,j) = C(i,j) + A(i,k)*B(k,j);
    end do
! store C(i,j)
end do
end do
```

Cela nécessite de lire  $\mathcal{O}(N^2)$  données depuis la mémoire pour réaliser  $\mathcal{O}(N^3)$  opérations. Il apparaît donc qu'il est possible d'avoir un nombre de cycles *Central Processing Unit* (CPU) dédiés au calcul  $\mathcal{O}(N)$  fois plus grand que le nombre de cycles dédiés au mouvement des données, si les transferts de données sont optimisés.

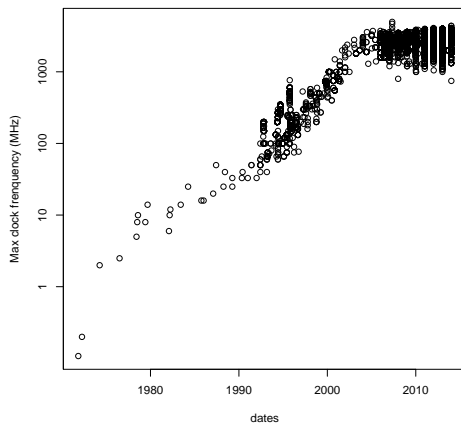
### 3.1.1 Architecture des processeurs

Les processeurs actuels utilisent toujours le modèle d'architecture théorisé par TURING en 1936 [121] et construit par GODFREY & HENDRY en 1945 ([69] et Figure 3) où les instructions sont stockées en mémoire comme des nombres et exécutées par un CPU.

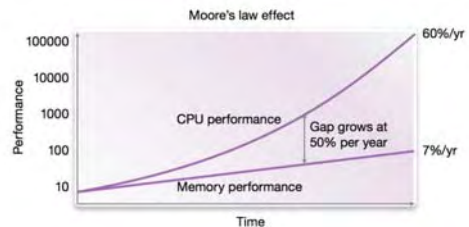
1. La routine *Linear Algebra PACKage* (LAPACK) qui effectue des produits denses en double précision est appelée DGEMM acronyme de *Double precision General Matrix Multiplication* [50].



(a) Top 500 supercomputer performance evolution



(b) Evolution of the maximum clock frequency [43]



(c) CPU performance vs memory performance

FIGURE 2 – Top500 performance, clock frequency, and memory wall

JEUX D'INSTRUCTION. Ces instructions peuvent être des opérations mathématiques<sup>2</sup>, des opérations logiques<sup>3</sup>, ou bien encore des opérations concernant la lecture et l'écriture en mémoire. Il fut un temps où ces instructions étaient très généralistes<sup>4</sup> et on devait les combiner afin d'en créer des plus spécifiques. Dans les CPU actuels, c'est de moins en moins

2. Par exemple des additions (**ADD**), des multiplications (**MUL**).

3. Des *et* (**AND**), des *ou* (**OR**), des *ou exclusif* (**XOR**)

4. Seulement des additions sur des entiers par exemple



vrai. Par exemple, il est maintenant possible de générer un nombre aléatoire grâce à une instruction spécifique<sup>5</sup>. Cette augmentation du jeu d'instruction permet une accélération des codes ; une instruction *hardware* sera souvent beaucoup plus efficace que son pendant algorithmique. Ces instructions sont effectuées dans ce qu'on appelle des *Unités d'Exécutions* ; il en existe plusieurs sortes en fonction du type d'opération à effectuer. À l'aube de l'informatique c'est à dire dans l'architecture de VON NEUNMAN, elle était unique : il s'agissait de l'*Arithmetic Logic Unit* (ALU), qui opérait sur des entiers. Il en existe maintenant des plus spécifiques. Par exemple des unités opérant sur les données flottantes (les *Floating-Point Units*) ; ou bien encore celles présentes sur les processeurs plus spécialisés s'occupant d'opérations cryptographiques. Dans les architectures actuelles, il existe même plusieurs unités de même type présent dans la puce, nous y reviendrons.

Notons qu'une telle architecture est *séquentielle* : les instructions sont exécutées l'une après l'autre dans une séquence prédéterminée. Pour qu'un tel modèle fonctionne efficacement, il est nécessaire que les données et les instructions puissent transiter suffisamment rapidement ; ainsi des mécanismes ont été créés afin d'optimiser cette communication.

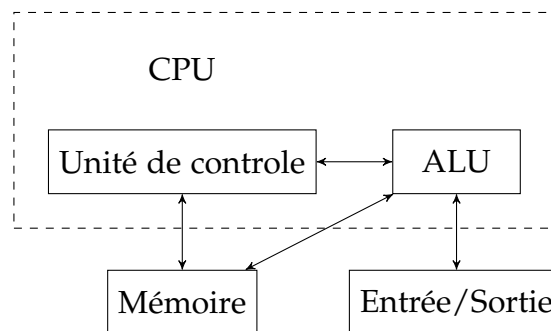


FIGURE 3 – Von Neumann architecture

#### 3.1.1.1 Caches<sup>6</sup>

Techniquement, il est plus facile d'augmenter la fréquence d'un processeur que d'augmenter la vitesse de communication des données. De ce fait, la fréquence des processeurs a augmenté beaucoup plus rapidement que la vitesse de migration des données ; il existe désormais un déséquilibre important entre le temps d'accès à une donnée et le temps que prend une opération : C'est ce que l'on appelle la *barrière de la mémoire* (*memory wall*) – voir Figure 2c). En effet, fabriquer de la mémoire avec un latence suffisamment faible pour alimenter rapidement les processeurs est devenu extrêmement coûteux ; un compromis a été trouvé en équipant les ordinateurs avec de la mémoire peu coûteuse – offrant une grande latence (de l'ordre de 200–300 cycles d'horloge) – et une petite quantité de mémoire à faible latence (entre 4 et 15 cycles) située entre le CPU et la mémoire centrale. La mémoire à faible latence constitue un *cache*, c'est-à-dire qu'elle contient à

5. L'instruction `rand`.

6. Pour une discussion plus détaillée, je ne saurais que conseiller l'article de DREPPER [51].

tout moment une copie temporaire de données provenant de la mémoire, permettant de réduire les temps d'accès à ces données.

**LOCALITÉ.** Une *ligne de cache* est la plus petite quantité de données qui peut être transférée entre la mémoire et le cache. Généralement, une ligne de cache contient 64 octets tels que le premier élément de la ligne de cache correspond à une adresse en mémoire qui est un multiple de 64 octets : les lignes de cache sont dites *alignées* sur des frontières de 64 octets. Lorsqu'un octet doit être chargé de la mémoire vers le cache toute la ligne de cache qui le contient est chargée. Ainsi, si le prochain accès à la mémoire fait référence à une donnée proche, celle-ci a de fortes chances d'être déjà dans le cache : on parle de *localité spatiale*. Les caches utilisant plusieurs lignes de cache, il existe aussi une *localité temporelle* permettant le remplacement d'anciennes données par des nouvelles. Plusieurs algorithmes ont été créés afin de déterminer quelle ligne de cache doit être évincée. La plus commune actuellement est la méthode dite *Least Recently Used* (LRU) : ce sont les données inutilisées les plus anciennes qui sont remplacées.

**HIÉRARCHISATION.** Vu le coût élevé des mémoires à faible latence, les processeurs x86 ont plusieurs niveaux de cache contenant des latences de plus en plus importantes mais des capacités de plus en plus grandes (Figure 4) :

**REGISTRES** quelques octets dans le cœur de calcul avec accès immédiat ;

**L1I** Cache de premier niveau contenant uniquement des instructions, typiquement de 16 ko<sup>7</sup> avec une latence de 4 cycles ;

**L1D** Cache de premier niveau contenant uniquement des données, typiquement de 16 ko avec une latence de 4 cycles ;

**L2** Cache de deuxième niveau contenant des données et des instructions, typiquement de 256 ko avec une latence de 12 cycles ;

**L3** Cache de troisième niveau contenant des données et des instructions, typiquement de 2.5 Mo par cœur avec une latence de 45 cycles.<sup>8</sup>

**PREFETCHERS.** Afin de réduire encore la latence, les caches ont des mécanismes (*prefetchers*) leur permettant d'aller chercher des données au niveau supérieur avant que celles-ci soient demandées par le niveau inférieur ; par exemple faire transiter des données situées dans la mémoire vers le cache L3. Les *prefetchers* sont capables de détecter des séquences

7. Il est amusant de remarquer qu'à l'heure où l'on parle de téraoctets de données, de gigaoctets de mémoire, les caches font au maximum quelques mégaoctets.

8. Le cache peut être partagé par plusieurs cœurs.

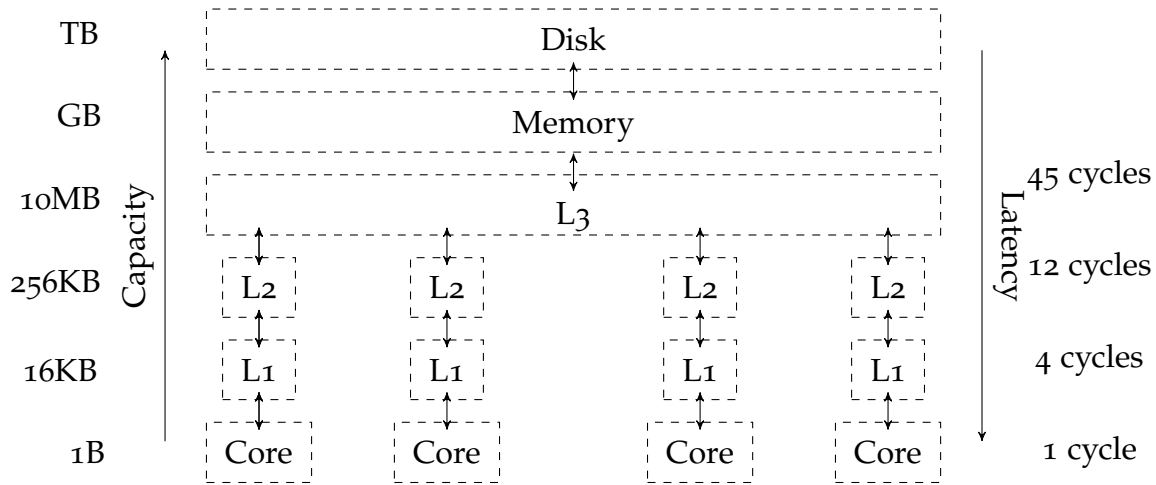


FIGURE 4 – Memory hierarchy

	Random	Prefetched
Memory	75 à 80	3.0
L3	13.9	1.7
L2	3.5	1.6
L1	1.2	1.2

TABLE 9 – Measurement of the latency for different cache levels (On my workstation / in nanoseconds). The measurements were made with LM-Bench [92].

*régulières* d'accès aux données, et donc de prédire quelles adresses seront demandées dans un futur proche.

OPTIMISATION DE L'UTILISATION DU CACHE. Cette architecture hiérarchique de la mémoire est telle que les accès les plus efficaces sont les accès à des données contiguës (*stride-one*). En effet, on profite à la fois de la localité spatiale (toutes les lignes de caches se suivent en mémoire), de la localité temporelle (toutes les données sont encore dans le cache) et des *prefetchers* (le chemin d'accès aux données est répétitif). Le produit de matrices dense est un bon exemple de l'utilisation optimale des caches. Reprenons le cas du produit de matrices denses utilisé dans le *benchmark* HPL :

$$(\mathbf{AB})_{ij} = \sum_{k=1}^N A_{ik} B_{kj}. \quad (17a)$$

Informatiquement, nous avons trois boucles imbriquées ( $i, j, k$ ). Si la boucle la plus interne tourne suivant  $i$ , les accès en écriture à  $(\mathbf{AB})$  et les accès en lecture à  $A$  sont dans les conditions optimales<sup>9</sup>. Si la matrice  $\mathbf{AB}$ , et les vecteurs  $A$  et  $B$  sont suffisamment petits pour tenir simultanément dans le cache  $L_1$ , ce calcul est optimal ; et l'on peut bénéficier de la performance crête du processeur. Dans le cas général, où les éléments sont trop grands, le produit peut être exprimé comme une somme de produits de sous-matrices suffisamment petites pour que les éléments tiennent dans les caches (*blocking*) :

$$\mathbf{AB}_{IJ} = \sum_K \mathbf{A}_{IK} \mathbf{B}_{KJ}. \quad (17b)$$

Ainsi, il est possible de faire quasiment disparaître les latences dues aux accès à la mémoire dans les produits de matrices denses pour toutes tailles de matrices, ce qui permet à ce produit d'être extrêmement performant. De plus, le calcul de chaque sous-matrice  $\mathbf{AB}_{IJ}$  est indépendant des autres sous-matrices et compte le même nombre d'instructions si les sous-matrices sont de mêmes tailles, donc le produit de matrices se prête extrêmement bien au parallélisme.

Les algorithmes ayant des accès imprévisibles et distants en mémoire – *Non-Uniform Memory Access* (NUMA) – auront des latences entre 250 fois et 300 fois plus importantes. On voit, d'une part que l'organisation des données et leur accès en mémoire est crucial pour la performance, et d'autre part que de telles différences de temps d'accès aux données

9. En effet il faut savoir qu'en Fortran les données sont *Column-Major Order*. C'est-à-dire que les éléments des colonnes sont stockés de façon contiguë ; à l'inverse du langage C où l'on doit lire ligne par ligne.

peuvent mener à des situations où réduire le nombre d'opérations faites par un programme peut conduire à une augmentation du temps de restitution si le schéma d'accès à la mémoire est dégradé. Autrement dit, il est parfois préférable de *faire plus de calculs* plutôt que de *relire des données* stockées en mémoire.

### 3.1.2 Cœurs de calcul

Jusqu'en 2004, la fréquence d'horloge des processeurs n'a cessé d'augmenter [Figure 2b](#). Au début de cette période, les algorithmes ont été développés dans un contexte d'architecture principalement monoprocesseur. Ainsi, lorsqu'une nouvelle machine apparaissait, les calculs tournaient automatiquement plus rapidement. Bien qu'un record de 500 GHz ait été battu avec un transistor refroidi à  $-269^{\circ}\text{C}$  avec de l'hélium liquide [[113](#)], cette augmentation de fréquence n'est pas compatible avec une diffusion à grande échelle des processeurs en raison des problèmes de consommation électrique et de dissipation thermique. En effet, cette dissipation thermique est donnée par

$$D = F \times V^2 \times C, \quad (18)$$

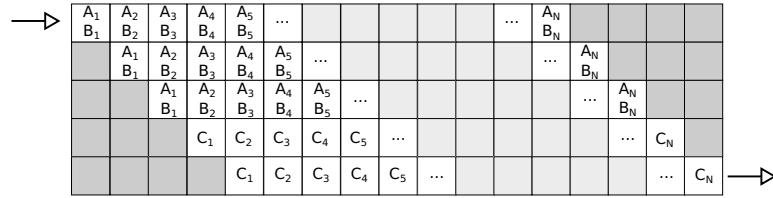
Où :

- D** représente la dissipation thermique ;
- F** est la fréquence du processeur ;
- v** correspond à la tension ;
- c** est une constante liée à la finesse de gravure.

À première vue, on a l'impression que la dissipation thermique varie linéairement avec la fréquence ; pourtant la tension nécessaire afin d'assurer la stabilité du processeur dépend linéairement de sa fréquence ainsi la dissipation thermique d'un processeur a un comportement cubique en fonction de sa fréquence. Ce qui signifie que pour doubler le nombre d'opérations effectuées par seconde, doubler le nombre d'unités de calcul aura un bien meilleur bilan thermique et électrique que de doubler la fréquence du processeur. De ce fait, la puissance de calcul disponible continue à doubler tous les 18 mois non plus en augmentant la fréquence des processeurs, mais bien en multipliant le nombre d'unités de calcul ([Figure 2b](#)). Il existe deux façons de faire :

- Celle consistant à augmenter le nombre de cœurs de calcul <sup>10</sup> afin d'augmenter le nombre d'opérations disponible par processeur. Cette barrière de la fréquence a fait naître vers 2006 les processeurs multicœurs généralistes. Utiliser pleinement la puissance de ces processeurs implique nécessairement de recourir au calcul parallèle,

10. Un cœur correspond ici à la définition précédente des CPU.

FIGURE 5 – Pipeline d'exécution de  $C_i = A_i \times B_i$ .

ce qui ne permet plus aux anciens programmes de tourner automatiquement plus vite sur du matériel plus récent.

- Augmenter le nombre d'ALU par cœur. Les cœurs actuels de type x86 sont des processeurs super-scalaires, c'est-à-dire qu'ils sont capables d'exécuter plusieurs instructions simultanément (*instruction level parallelism*). En effet, puisqu'il n'est pas raisonnable d'un point de vue énergétique d'augmenter la fréquence des cœurs de calcul, les processeurs deviennent de plus en plus complexes<sup>11</sup>. Par exemple, afin de maximiser la quantité de calcul effectuée par cycle d'horloge : la micro architecture HASWELL permet aujourd'hui d'effectuer au maximum 32 flops par cycle en simple précision et 16 en double précision. Il s'agit de parallélisation, directement effectuée par le cœur du processeur. Nous allons, dans la suite de cette partie, parler de deux grandes stratégies permettant cette parallélisation automatique : le *pipelining* et la *vectorisation*.

### 3.1.2.1 Pipelining

Nous avons vu qu'il existe maintenant une augmentation du nombre d'unités d'exécution. Le *pipelining* est une technique permettant de profiter au maximum et de façon simultanée de plusieurs unités.

Les opérations telles que l'addition et la multiplication sont divisées en sous-opérations élémentaires qui sont exécutées par des ALU différentes. Par exemple, la multiplication est sous-divisée en plusieurs étapes :

1. Séparation des mantisses<sup>12</sup> et des exposants ;
2. Multiplication des mantisses ;
3. Addition des exposants ;
4. Normalisation du résultat ;
5. Ajout du signe.

Dans cet exemple, la multiplication est effectuée en 5 cycles d'horloge ; on dit alors qu'elle a une latence de 5 cycles. Cette quantité est fixe, on

11. *pipelining*, prédiction de branchement, vectorisation, exécution *out of order*, etc.

12. Le terme mantisse désigne la différence entre le logarithme d'un nombre et la partie entière de ce logarithme, c'est-à-dire sa partie fractionnaire.

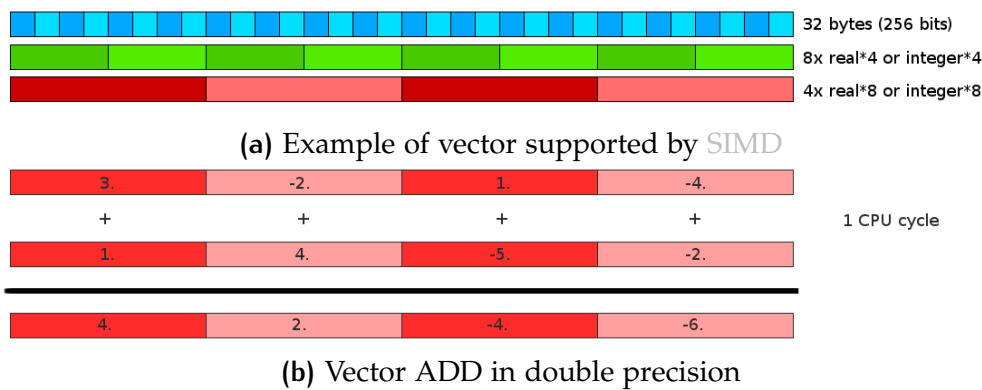


FIGURE 6 – SIMD schema

ne pourra jamais faire moins. Par contre, si chaque étape est exécutée par des unités différentes, il est possible de démarrer une seconde multiplication dès que la première multiplication entre dans l'étape numéro 2. Ainsi, lorsque plusieurs multiplications sont effectuées en séquence, la Figure 5 montre qu'on peut obtenir *1 résultat par cycle*, mais avec une latence de 5 cycles.

### 3.1.2.2 Vectorisation

La vectorisation permet au processeur d'effectuer simultanément plusieurs opérations identiques sur des données différentes. C'est la parallélisation dite *Single Instruction, Multiple Data (SIMD)*. Par exemple, le jeu d'instructions AVX opère sur des vecteurs de 256 bits (Figure 6a) ; ce qui permet de traiter 4 additions flottantes en double précision en un cycle CPU (Figure 6b). La vectorisation est une des clés pour écrire du code efficace sur les architectures modernes, une fois que les accès aux données ont été optimisés. Les instructions vectorisées peuvent être engendrées automatiquement par le compilateur ; on parle alors de vectorisation automatique. Dans la suite, nous verrons comment le compilateur vectorise les boucles et comment en tirer partie afin d'avoir le code le plus efficace possible.

**VECTORISATION DES BOUCLES.** Afin de pouvoir vectoriser les instructions présentes dans une boucle faisant  $N$  tours (première étape de la Figure 7), le compilateur découpe cette boucle en trois boucles indépendantes (étape 2 de la même figure) :

**PEEL LOOP** Pour pouvoir exécuter une instruction vectorielle, il est nécessaire que les opérandes aient un alignement particulier en mémoire. Par exemple, le premier élément du vecteur doit avoir une adresse mémoire multiple de 16 octets pour le jeu d'instructions SSE, 32 octets pour le jeu d'instructions AVX ou 64 octets pour le jeu d'instructions AVX-512. Ainsi la *peel loop* correspond à une exécution scalaire des

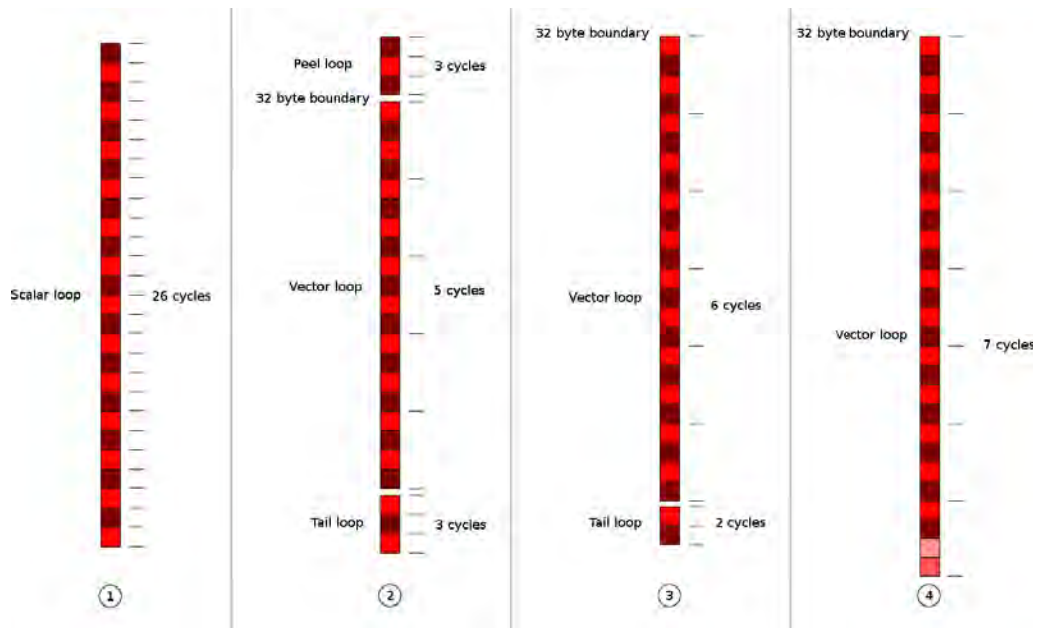


FIGURE 7 – Vectorization with and without peel and tail loops

premières itérations de la boucle jusqu'à ce que le bon alignement soit atteint.

**VECTOR LOOP** Il s'agit de la boucle déroulée  $n$  fois utilisant des instructions vectorielles pour des vecteurs de taille  $n$

**TAIL LOOP** Il s'agit des  $m$  dernières itérations réalisées avec des instructions scalaires.

**Optimisation.** On peut déjà remarquer que la stratégie d'exécution dépend de l'alignement et du nombre de tours de boucle. Ainsi, si le nombre de tours est écrit « en dur » dans le code, la stratégie optimale peut être choisie à la compilation ; sinon elle doit être effectuée dynamiquement à l'exécution créant un surcoût non négligeable. On a donc ici la première possibilité d'optimisation : pour de petits tableaux, il peut être préférable de créer des routines spécifiques pour chaque dimension plutôt que de créer des routines générales. De plus, il est possible d'aligner soi-même les vecteurs sur des adresses mémoires spécifiques, permettant la suppression de la *peel loop*<sup>13</sup>. Ceci est l'étape 3 de Figure 7. Ensuite pour faire disparaître la *tail Loop*, on peut augmenter artificiellement la taille des tableaux afin que leurs dimensions soit un multiple du nombre d'opérations pouvant être effectuées en un cycle CPU par la vectorisation. Par exemple, si l'on prend l'exemple de Figure 7, l'on peut remplacer les deux dernières opérations scalaires peuvent être remplacé par une opération vectorielle si le tableau est artificiellement allongé.

13. Cette information doit être passée au compilateur, par exemple par l'utilisation de directives.



### 3.1.3 Distribution d'un calcul entre plusieurs cœurs

#### 3.1.3.1 *Le futur et ces contraintes*

Dans cette partie, nous restreindrons le terme de parallélisation à la distribution d'un calcul entre plusieurs cœurs. Nous verrons les différentes stratégies possibles afin d'achever cette parallélisation. Avant de rentrer dans ces détails, j'aimerais faire un point sur la situation actuelle au niveau de la puissance de calcul et les nouveaux enjeux en découlant.

Il est aujourd'hui possible d'avoir accès à des supercalculateurs accueillant plusieurs centaines de milliers de cœurs et une puissance totale de plus en plus importante (Figure 2a). Néanmoins, nous avons vu que la vitesse de la mémoire n'a pas évolué aussi rapidement que la vitesse des processeurs, causant des difficultés pour profiter pleinement des nouveaux processeurs ; on peut trouver une similitude dans le domaine de la parallélisation : les codes de calcul n'ont pas évolué aussi vite que les ressources disponibles rendant l'utilisation de cette puissance disponible de plus en plus problématique<sup>14</sup>. On parle pour bientôt de supercalculateurs de puissance exaflopique. Les programmes qui « passeront à l'échelle » devront remplir certaines conditions :

**ENTIÈREMENT PARALLÉLISABLE** La loi d'AMDAHL [1] permet de prédire le rapport entre le temps d'exécution du programme sur  $n$  processeurs par rapport au temps d'exécution du même programme séquentiel. On parle d'« accélération » ou encore de *speedup*. Si l'on définit  $s$  comme étant le pourcentage du code étant séquentiel et  $p$  comme étant la partie parallélisée on obtient l'accélération grâce à :

$$S(n) = \frac{1}{s + \frac{p}{n}} \quad (19)$$

Regardons les cas-limites donnés par cette formule. Si l'on a un code purement séquentiel ( $s = 1, p = 0$ ) alors l'accélération est unitaire. Peu importe le nombre de processeurs que l'on utilise, le code ne pourra jamais aller plus vite que sur un seul processeur. À l'inverse si le code est purement parallélisable ( $s = 0, p = 1$ ), l'accélération est strictement égale au nombre de processeurs. Ainsi, si l'on souhaite diminuer le temps de restitution – c'est-à-dire le temps qu'attend l'utilisateur avant d'avoir son résultat – d'un facteur 10, il suffit d'utiliser dix fois plus de processeurs. Il s'agit du cas de figure idéal<sup>15</sup>. Il est intéressant de regarder cette loi pour des valeurs intermédiaires de  $p$ <sup>16</sup>. Ainsi si l'on a un code parallélisé à hauteur de 50 % ( $s = 0.5, p = 0.5$ ), on ne

<sup>14</sup>. Par exemple, GAUSSIAN, un code de chimie quantique très populaire, n'est que très peu parallélisé.

<sup>15</sup>. Il est parfois possible d'obtenir une accélération super linéaire ; en effet en découplant des tableaux on peut réussir à les faire tenir dans le cache.

<sup>16</sup>. Pour une représentation graphique, voir la Figure 8.

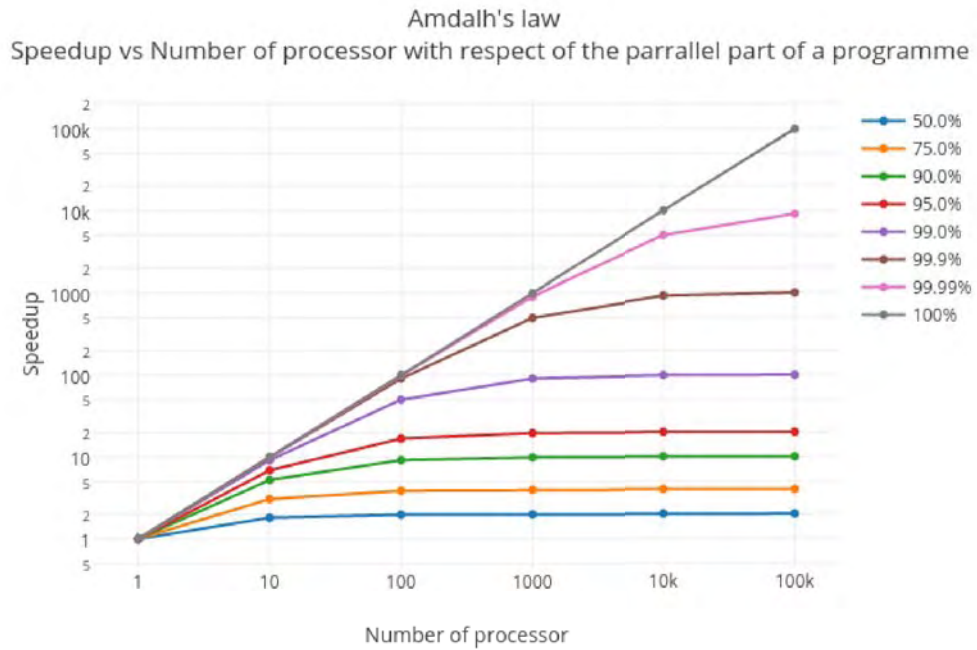


FIGURE 8 – ANDAHL'S law representation

pourra jamais avoir une accélération plus grande qu'un facteur deux et cela, quelque soit le nombre de processeurs utilisés. De même avec un code parallélisé à 99 % ( $s = 0.01$ ,  $p = 0.99$ ) il est impossible d'avoir une accélération plus grande que 100. On voit en plus que cette évolution n'est pas linéaire. Il nous faut environ 10 000 processeurs pour atteindre ce *speedup* et si l'on utilise « seulement » cent processeurs, on aura une accélération limitée à un facteur 50 ...

Cette simple loi nous apprend donc que pour passer à l'échelle, tout le code doit pouvoir être découpé en sous-parties indépendantes les une des autres ; autrement dit des codes où les communications réseaux seront négligeables et où toutes les sous-parties seront asynchrones. Il y a un terme pour décrire ce genre de parallélisation : *embarrassingly parallel*. « Embarrassant », car d'un côté il a une efficacité parallèle bien supérieure comparativement à d'autres programmes faisant un usage important des communications, et la parallélisation de ce genre de programme est beaucoup plus facile à écrire ; mais de l'autre côté, cela est aussi embarrassant, car ce type de calcul « sous-utilise » les ressources réseau d'un supercalculateur<sup>17</sup>. Pour utiliser un nombre important de ressources, il ne s'agira plus d'améliorer les algorithmes existants, mais de développer des approches originales.

17. Je me souviens qu'après notre sélection pour le Meso-challenge organisé par le centre de calcul CALMIP qu'un de nos concurrents m'a fait remarquer qu'il ne comprenait pas pourquoi nous étions les lauréats. Bien sûr que nous allions avoir un *scaling* linéaire même sur 12 000 cœurs, à quoi bon nous donner les ressources ... ?

**RÉSILIENT** Lorsque l'on utilise plusieurs dizaines de milliers de processeurs (et toutes les autres ressources requises), les pannes sont statistiquement inévitables. Prenons par exemple un calcul tournant sur 200 000 cœurs, donc sur 10 000 nœuds à 20 cœurs équipés chacun de 8 barrettes de mémoire à 8 GiB. Si l'on considère que le temps moyen entre les pannes – *Mean Time Between Failure* (MTBF) – est de 500 ans<sup>18</sup>, alors on est presque sûr d'avoir une panne due à la mémoire au bout de trois jours de calcul ! De ce fait, lors d'une panne, si possible le programme ne devrait pas s'arrêter ; il devrait pouvoir continuer en tenant compte de la panne. En un mot, il doit être résilient.

**MODULABLE** Ceci est peut-être un corollaire vertueux du second point, mais dans l'idéal un code doit pouvoir se permettre de gagner ou de perdre des ressources en cours de simulation ; on parle alors d'une gestion *élastique* des ressources. En effet les ressources disponibles sont fluctuantes, dépendant du taux d'utilisation de la machine par d'autres utilisateurs. Si le parallélisme est utilisé pour minimiser le temps de restitution, alors il est souhaitable de pouvoir démarrer le calcul le plus tôt possible sur une faible quantité de ressources quitte à ajouter des ressources supplémentaires en cours de calcul<sup>19</sup>.

### 3.1.3.2 Parallélisme à mémoire partagée

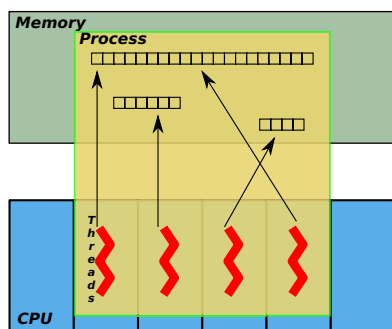


FIGURE 9 – Standard representation of parallelization

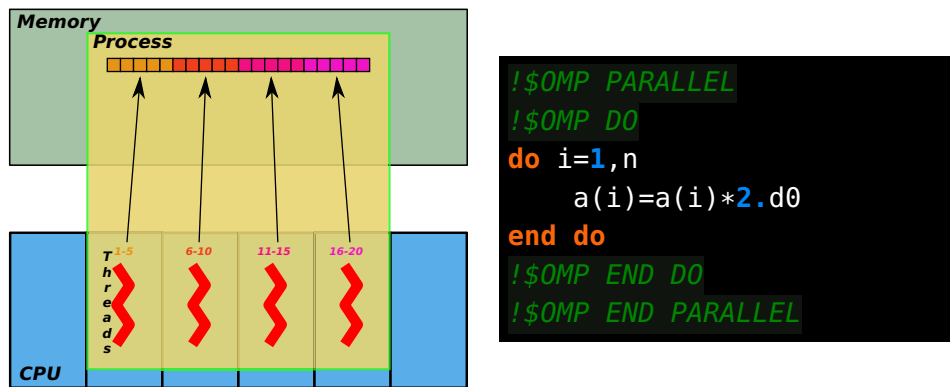
Un processus est une instance d'un programme qui est en cours d'exécution ; il peut contenir un ou plusieurs fils d'exécution (*threads*) qui s'exécutent simultanément et partagent la même plage mémoire. Si les *threads* sont utilisées pour paralléliser un programme, on parle de *parallélisme à mémoire partagée*. Ce type de parallélisation est très bien adapté aux algorithmes nécessitant beaucoup de communications collectives et de synchronisations, en effet la mémoire étant partagée les copies de mémoire peuvent être évitées. La difficulté de l'usage des *threads*

est le fait que l'on doit éviter les écritures concurrentes aux mêmes adresses mémoires.

L'*Open Multi-Processing* (OpenMP) est une *Application Program Interface* (API) de haut niveau permettant de faciliter la programmation parallèle à mémoire partagée. Elle fonctionne majoritairement par l'utilisation de directives de compilation données en commentaire dans le code

18. Le constructeur Kingston garantit : "Our process works so well that our mean time between failure rating exceeds 500 years !"

19. Un exemple de ce type de code est montré Figure 11.



CODE 1 – OpenMP example in Fortran

source<sup>20</sup>. Ces directives permettent de délimiter des blocs qui seront parallélisés ainsi que différentes options<sup>21</sup>. L'OpenMP génère le code qui permettra la création de *threads* pour effectuer les calculs situés dans ces blocs. Dans le cas illustré CODE 1, la parallélisation de la boucle ne prend que quatre lignes de code.; on voit que chaque *thread* accède simultanément à une zone différente de la mémoire partagée.

DÉSAVANTAGE. Le fait qu'utiliser OpenMP nécessite de la mémoire partagée, ne permet pas à lui seul de programmer un supercalculateur standard. En effet, il ne permet de paralléliser un programme que sur un nœud de calcul, mais ne permet pas de sortir du nœud. Notons que certains constructeurs proposent des supercalculateurs de type *Single System Image* qui permettent d'utiliser de l'OpenMP sur plusieurs nœuds de calcul, mais dans ce cas il est impératif de prendre en considération les effets d'accès non-uniformes à la mémoire (NUMA) pour ne pas observer de dégradation de performance trop importante sur ce type de machine.

### 3.1.3.3 Parallélisme à mémoire distribuée

Pour des raisons principalement économiques, les supercalculateurs actuels n'ont pas une mémoire globale accessible par tous les cœurs, mais sont plutôt des agglomérats (*clusters*) de nœuds interconnectés par un réseau rapide. Il est donc nécessaire d'utiliser un *parallélisme à mémoire distribuée* afin d'échanger des données entre les nœuds.

SOCKETS. Les *sockets* sont l'outil historique permettant la communication de différents processus, éventuellement à travers le réseau. Elles sont traditionnellement utilisées selon une architecture de type client/-

20. Cette écriture par insertion de commentaires a l'avantage de permettre d'utiliser le même code source pour produire du code séquentiel ou parallèle.

21. Ces directives permettent aussi de catégoriser les variables (publique, privée) ou de définir le type de découpage des tâches voulu (statique, dynamique ou guidé).

```

import socket

serversocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# AF_INET is the adresse family
# SOCK_STREAM is a connection-based protocol.

serversocket.bind(('localhost', 8089))
serversocket.listen(5)
# become a server socket, maximum 5 connections

while True:
    connection, address = serversocket.accept()
    buf = connection.recv(64)
    if len(buf) > 0:
        print buf

```

(a) Socket Server

```

import socket

clientsocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
clientsocket.connect(('localhost', 8089))
clientsocket.send('hello')

```

(b) Socket Client

## CODE 2 – Socket Client Server

serveur. Un serveur crée une *socket* et des clients viennent s’y connecter. Ces *sockets* s’utilisent grâce à une API ; la plus vieille et donc maintenant celle faisant office de standard est celle dite de BERKELEY datant de 1983. Elle définit quelques fonctions essentielles :

1. Celles permettant de créer une *socket* pour le serveur <sup>22</sup> ;
2. Celles permettant de se connecter et de se déconnecter à une *socket* au niveau du client ;
3. Celles permettant d’envoyer et de recevoir des messages pour les deux côtés.

Un exemple en Python est présenté CODE 2a pour le serveur et CODE 2b pour le client. Il s’agit de l’exemple typique d’une communication point-à-point bloquante. Le serveur crée une *socket*, attend des connexions puis affiche le message reçu. Le client quant à lui, se connecte à une adresse et envoie le message. On voit dans cet exemple que les *sockets* sont une technologie de bas niveau ; il faut par exemple s’assurer d’avoir déjà lancé le serveur avant de pouvoir lancer les clients.

22. La création proprement dite. Il s’agit de l’association de cette *socket* à une adresse par exemple, une adresse IP.

MESSAGE PASSING INTERFACE. (MPI) Au début des années 1990 avec l'émergence des ordinateurs à mémoire distribuée de nombreux réseaux à haute performance ont vu le jour. Des bibliothèques – souvent propriétaires – permettant de communiquer à travers ces réseaux furent *de facto* développées. Malheureusement, leur nombre important fut un frein à l'extension de ce type d'architecture. Il fut ainsi décidé de standardiser la communication par envoi de messages au travers de l'API *Message Passing Interface* (MPI)<sup>23</sup>. Cette standardisation a permis un grand bond en avant et a été l'étape cruciale qui a permis la diffusion rapide des machines à mémoire distribuée.

Les bibliothèques MPI ne reposent pas sur TCP/IP, mais ont été conçues avec pour but de diminuer au maximum la latence des communications entre machines. Avec un réseau de communications à faible latence<sup>24</sup> il est possible d'avoir des latences entre machines de l'ordre de la micro-seconde, seulement environ dix fois la latence d'un accès aléatoire à la mémoire.

MPI utilise le paradigme *Single Program, Multiple Data* (SPMD). Le code est divisé en plusieurs processus qui se verront chacun assigné un rang qui fera office d'adresse. Ce modèle est très bien adapté à la communauté de la dynamique des fluides, de la climatologie ou de la mécanique où chaque nœud d'un maillage doit effectuer la même tâche et doit communiquer des données avec les nœuds voisins. MPI est donc devenu aujourd'hui le standard de communication dans le *High Performance Computing* (HPC). Des fonctions permettent de communiquer entre ces processus par envoi de messages. Typiquement il faut deux fonctions : une pour envoyer le message, l'autre pour le recevoir. C'est un utilitaire d'assez bas niveau, donc il convient de passer à ces fonctions un nombre assez important de paramètres<sup>25</sup> ; le plus important restant le rang du processus permettant l'exécution de la bonne séquence d'instructions du programme que l'on souhaite paralléliser. Le cas de figure le plus simple d'une communication point à point est illustré CODE 3. MPI a débuté avec des routines permettant d'envoyer et de recevoir des messages de façon explicite<sup>26</sup> et des communications collectives<sup>27</sup>. Plus tard, des communications non-bloquantes ont été introduites, des entrées/sorties sur disque en parallèle<sup>28</sup>, etc.

L'APRÈS MPI. Quelques 20 années après la première spécification MPI, la situation a évolué :

23. Aujourd'hui, il existe plusieurs implémentations de cette API (INTEL MPI, BULLX MPI, SGI MPT, OPENMPI, MPICH, MVAPICH, ...).

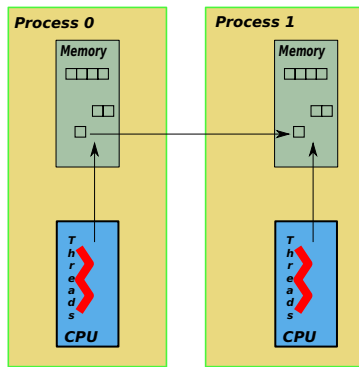
24. Typiquement de type INFINIBAND.

25. Le nombre d'éléments à envoyer, le type d'élément à envoyer, etc.

26. C'est à dire bloquante.

27. *Broadcast, reduce, etc.*

28. Nommés MPI I/O.



Send :

```
call MPI_SEND(data, count,
               MPI_DOUBLE_PRECISION,
               2, tag, MPI_COMM_WORLD,
               ierr )
```

Receive :

```
call MPI_RECV(data, count,
               MPI_DOUBLE_PRECISION,
               1, tag, MPI_COMM_WORLD,
               status, ierr )
```

CODE 3 – MPI representation

- Bien qu’il y ait des avancées technologiques, les réseaux sont de plus en plus lents comparativement aux CPU. Il n’est donc plus question de minimiser au maximum la latence en passant par des protocoles de bas niveau, mais plutôt de gérer les communications de façon asynchrone afin de masquer ces latences; ceci suit la même logique que l’apparition des caches dans les processeurs.
- MPI ne permet pas encore une gestion robuste des pannes<sup>29</sup>. En effet, le design de MPI repose sur le fait que tous les processus doivent se synchroniser à la fin du calcul<sup>30</sup>; si un processus ne peut pas atteindre ce point de rendez-vous, la bibliothèque va choisir de stopper le programme pour ne pas consommer inutilement des ressources. Ceci est un comportement naturel et très satisfaisant pour certaines applications, mais il y a certains cas où l’utilisateur veut pouvoir gérer le comportement du programme en cas de panne. Prenons par exemple, les algorithmes stochastiques où le résultat est obtenu en moyennant les sous moyennes obtenues sur des trajectoires réalisées sur différents nœuds<sup>31</sup>. Pour ces algorithmes, perdre un nœud ne change pas le résultat, on perd simplement un peu de statistique; le calcul doit donc pouvoir continuer. Étant donné que plus la quantité de ressources utilisées est grande, plus la probabilité de panne est grande, les programmes MPI ne seront jamais « scalables » à très grande échelle tant que le problème de la résilience ne sera pas résolu. Aujourd’hui, faire tourner un programme MPI sur un million de cœurs est une prouesse, tandis qu’avoir un service web avec un million de clients n’est pas exceptionnel.

29. Bien que certaines équipes de recherche travaillent sur ce sujet [58].

30. Il s’agit d’appeler la fonction `MPI_Finalize`.

31. Typiquement le cas du *Quantum Monte Carlo*, Monte Carlo quantique (QMC) décrit précédemment.



- Le modèle imposé par MPI implique que la quantité de ressources soit mise à disposition avant le départ du calcul, et que ces ressources soient constantes au cours du calcul.

Pour toutes ces raisons, le MPI a été une révolution à son époque, mais ne semble optimum – en règle générale – que pour une parallélisation sur quelques dizaines ou centaines de nœuds. Si l'on veut utiliser toutes les ressources disponibles avec un bon *speedup*, il est nécessaire de changer de paradigme. En un mot, il nous faudrait combiner la flexibilité des sockets et la vitesse du MPI.

#### 3.1.3.4 Note sur PGAS

Pour se rapprocher du modèle à mémoire partagée, le modèle *Partitioned Global Address Space* (PGAS) a été développé<sup>32</sup>. Dans ce modèle, tous les *processus* voient virtuellement un espace mémoire commun et peuvent lire et écrire dans cette zone de façon transparente. Ce *bulk* commun est physiquement partagé entre tous les processus. Ainsi, les développeurs utilisant le modèle PGAS peuvent facilement implémenter des algorithmes écrits pour des machines à mémoire partagée dans un contexte distribué.

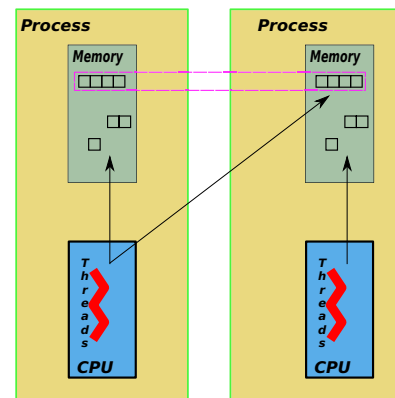


FIGURE 10 – PGAS représentation

#### 3.1.3.5 ØMQ

UNE TECHNOLOGIE DE HAUT NIVEAU. ØMQ est un *message-oriented middleware* c'est-à-dire une bibliothèque permettant l'envoi et la réception asynchrones de messages entre plusieurs systèmes distribués. À l'opposé de MPI, la bibliothèque ØMQ a été écrite dans un contexte où les communications sont lentes et sous-tendues par des réseaux peu fiables. Elle a été développée principalement pour une utilisation dans le domaine de la finance<sup>33</sup>. Ainsi, la performance et la robustesse ont été des critères importants pendant le processus de développement. La rapidité des communications ne repose donc pas seulement sur l'envoi de messages « nus » sur du matériel haut de gamme, mais plutôt sur des algorithmes déterminant quand et comment envoyer les données. En effet, les lettres MQ de ØMQ signifient *message queuing*. Les messages ne sont pas envoyés directement, mais sont d'abord stockés dans une queue d'envoi permettant des optimisations éventuelles. Par exemple, un train

32. Unified Parallel C, CoArray Fortran, Titanium, Global Arrays, etc.

33. Trading à haute fréquence, etc.



de petits messages sera envoyé comme un seul paquet, ce qui permettra d'avoir de très bonnes performances pour l'envoi fréquent de ces petits messages. Ici, le modèle SPMD n'est pas imposé (rien n'oblige donc que les codes qui doivent communiquer entre eux soient écrits dans le même langage<sup>34</sup>) et les ressources peuvent être gérées dynamiquement à la demande, ce qui permet l'élasticité des ressources. Ainsi, en plus des stratégies présentes dans la bibliothèque ØMQ, cela laisse libre cours à l'utilisateur d'implémenter sa stratégie de tolérance aux pannes.

ØMQ permet de communiquer de machine à machine, de processus à processus, de *thread* à *thread* en utilisant des protocoles adaptés<sup>35</sup>. Pour passer d'un modèle de communication à un autre, il suffit simplement de changer un paramètre dans l'appel de l'API. Contrairement aux *sockets* TCP, il n'y a pas d'ordre préétabli concernant la création et la liaison (*binding*) des *sockets* ØMQ. Ainsi, on peut lancer le client avant que le serveur n'ait ouvert sa *socket* et la connexion sera établie lorsque le serveur sera prêt. Cela découle de la tolérance aux pannes implémentée dans la bibliothèque : si la connexion entre le client et le serveur est interrompue, les données continueront à s'accumuler dans les queues d'envoi puis seront expédiées une fois que le serveur sera de nouveau opérationnel. Pour ne rien gâcher, elle utilise une API calquée sur celle des *sockets* BERKELEY présentée plus haut rendant sa prise en main très aisée. Dans les CODE 4b et CODE 4a, nous retrouvons l'exemple de la communication point-à-point entre un client et un serveur ; la différence avec l'exemple sur les *sockets* est qu'ici le serveur envoie une réponse au client dès la réception d'un message.

UNE TECHNOLOGIE ADAPTÉE AU CALCUL MASSIVEMENT PARALLÈLE. Cette bibliothèque est naturellement tolérante aux pannes, condition nécessaire à l'utilisation d'un nombre très important de nœuds. De plus, cette bibliothèque est très souple : la plupart des stratégies de parallélisation<sup>36</sup> sont faciles à mettre en place. Pour atteindre l'*exascale*, des codes monolithiques ne seront peut-être pas le Graal. En effet, le cœur calculatoire restera fondamentalement dans un langage de bas niveau (type Fortran ou C), mais la gestion du parallélisme ou le comportement en cas de panne peuvent être un exercice plus difficile à faire dans ce genre de langage que dans un langage de plus haut niveau. De même, il est impensable de faire une interface graphique en Fortran, pourtant cette interface devra pouvoir communiquer avec le code. La philosophie UNIX ne dit ni plus ni moins que cela : "[the heart of unix philosophy] is the idea that the power of a system comes more from the relationships among programs than from the programs themselves".

34. Il existe à l'heure actuelle une vingtaine de langages qui supportent cette bibliothèque.

35. *Sockets* UNIX ou TCP, copie de message en mémoire, etc.

36. *Publish/subscribe*, modèle Client-Serveur, SPMD, *divide and conquer*, etc.

```

import zmq

# ZeroMQ Context (equivalent to MPI_INIT)
context = zmq.Context()

# Socket to REPLY to clients
sock = context.socket(zmq.REP)
sock.bind("tcp://127.0.0.1:5678")

while True:
    message = sock.recv()
    print message
    sock.send("world ")

```

(a) Server

```

import zmq

# ZeroMQ Context
context = zmq.Context()

# Client socket to make a REQuest
sock = context.socket(zmq.REQ)
sock.connect("tcp://127.0.0.1:5678")

# Send a "message" using the socket
sock.send("hello ?")
# Get the response
message = sock.recv()
print message

```

(b) Client

CODE 4 – ØMQ REQ/REP Server and Client in Python

ØMQ paraît donc avoir un avenir radieux dans le domaine du HPC massivement parallèle; nous verrons une application de cette bibliothèque dans le cas de notre code QMC=Chem ayant tourné avec succès<sup>37</sup> sur des dizaines de milliers de processeurs.

## 3.2 SUPERCALCULATEURS ET *cloud computing*

Dans la première section nous avons parlé des processeurs en eux-mêmes, puis des implémentations permettant de les utiliser en parallèle. Dans cette section nous parlerons des infrastructures permettant cette

<sup>37</sup>. Soit avec une accélération presque idéale et sans *crash*.

parallélisation, elle peuvent se découper entre deux grandes catégories : d'une part les supercalculateurs et d'autre part les grilles de calcul distribuées ; ou pour utiliser un mot tendance le *cloud computing*.

On parlera rapidement des supercalculateurs et plus en détail du *cloud computing* dans le domaine du HPC.

### 3.2.1 Supercalculateurs

Un supercalculateur est – à ce jour – un ensemble de nœuds de calcul reliés entre eux par un réseau performant possédant souvent un système de fichiers distribué (Tableau 10). Il existe deux principaux challenges que doivent résoudre les supercalculateurs actuels :

**CONSOMMATION ÉLECTRIQUE** La relation entre la fréquence d'un processeur et sa consommation électrique n'est pas linéaire. Ainsi, même sans parler des problèmes de refroidissement, il est plus rentable énergétiquement pour la même puissance d'avoir un grand nombre de processeurs à basse fréquence qu'un processeur à très grande cadence. Les supercalculateurs à base de processeurs à basse consommation<sup>38</sup> sont donc en train de se développer. Au laboratoire nous avons pu avoir ce genre de serveur et nous constatons une diminution du coût énergétique<sup>39</sup> de l'ordre de 4 par rapport à un serveur de calcul traditionnel [74] ;

**MÉMOIRE PAR NŒUD** Comme le nombre de processeurs par nœud évolue beaucoup plus vite que la mémoire par nœud, on constate une diminution de la quantité de mémoire par processeur. Ainsi les algorithmes nécessitant beaucoup de mémoire utilisant seulement le paradigme SPMD – de type « pur MPI » – ne peuvent pas bénéficier de l'évolution des supercalculateurs. Il est nécessaire d'avoir en plus recours à une parallélisation à mémoire partagée pour utiliser tous les cœurs de calcul. Cela implique donc plusieurs degrés de parallélisme dans ces applications.

### 3.2.2 Cloud computing

D'un point de vue d'utilisateur de supercalculateur, le *cloud computing* permet d'avoir accès à un supercalculateur dont les nœuds de calcul sont le plus souvent virtuels et dont les latences des communications réseaux sont colossales<sup>40</sup>. Cette latence élevée a un prix pour le HPC, dans le livre de 2008 [77] écrit par le développeur principal de la suite logicielle

38. Comme les processeurs de type ATOM ou ARM.

39. Soit une augmentation du nombre de Gflop/s par watt.

40. Pour la COMMISSION GÉNÉRALE DE TERMINOLOGIE ET DE NÉOLOGIE, il s'agit d'un « Mode de traitement des données d'un client, dont l'exploitation s'effectue par l'internet, sous la forme de services fournis par un prestataire. »[93]. Ce qui signifie que

Name	Organisation	Nodes	Core	Architecture
Curie	TGCC/CEA/GENCI	$\approx 5000$	$2 \times 8$	Sandy bridge
Occigen	CINES	$\approx 2100$	$2 \times 12$	Haswell
Eos	CALMIP	$\approx 600$	$2 \times 10$	Ivy bridge
UVProd (in Eos)	CALMIP	1	$1 \times 384$	Nehalem
130.120.xxx.45	LCPQ	68	$1 \times 8$	Atom (HP Moonshot)
lpqsv26.ups-tlse.fr	LCPQ	$\approx 30$	$2 \times \{8, 10, 12\}$ , $4 \times 12$	Haswell, Ivy bridge, Sandy bridge, Nehalem, AMD, ...

TABLE 10 – Overview of the computational resources used in this thesis.

Where	CPU	Clock (GHz)	Cores	Cache (MiB)
CALMIP	Xeon E5-2680 v2	2.8	$2 \times 10$	25
Desktop	Xeon E3-1271 v3	3.6	$1 \times 4$	8
Moonshot	Atom C2730	2.4	$1 \times 8$	4
IPHC	Westmere (E/L/X)56xx (Nehalem-C)	2.5	$2 \times 8$	4
LAL	QEMU Virtual CPU (cpu64-rhel6)	2.7	$1 \times 8$	4

TABLE 11 – Characteristics of processors (cloud and other)

de chimie quantique *Massively Parallel Quantum Chemistry* (MPQC), les auteurs nous font part de cette remarque :

*A grid network's performance, however, is too low (the bisection width is too small and the latency too large) to be of direct interest for the quantum chemistry applications discussed in this book.*

En effet, les codes de chimie quantique « traditionnels » font un usage important du réseau. Cette caractéristique les rend très peu pertinents pour une utilisation sur les grilles de calcul du fait de l'énorme latence du réseau. Néanmoins, il existe dans notre domaine un type de code permettant une utilisation optimale de ces grilles de calcul : les méthodes stochastiques<sup>41</sup> qui présentent un parallélisme dit *embarrassant*.

**CLOUD CHALLENGE.** Afin de promouvoir l'utilisation du *cloud* pour le calcul HPC auprès de la communauté académique, FRANCE GRILLES [61] a lancé en 2014 un appel pour un *Cloud Challenge* pour lequel nous avons été sélectionné. Le nuage de FRANCE GRILLES est réparti dans cinq

l'utilisateur n'a pas besoin d'avoir de serveur propre à sa disposition. Le *National Institute of Standards and Technology* américain rajoute [78] que les ressources disponibles doivent être élastiques, payées à la demande et le plus souvent virtualisées.

41. Par exemple les méthodes QMC définies dans le chapitre 2.

centres physiques à travers la France<sup>42</sup> disposant d’architectures matérielles différentes (Tableau 11); dans notre cas, nous avons réalisé des *benchmarks* sur les machines de l’IPHC et du LAL. Notre motivation pour participer à ce challenge est bien sûr de démontrer la très grande souplesse des simulations stochastiques et de notre implémentation ainsi que nous préparer au passage à l’échelle; nous voyons la grille comme un bon moyen de parvenir à ce but. Notons également que si nos codes fonctionnent bien sur la grille de calcul ils fonctionneront encore mieux sur les supercalculateurs actuels et futurs. En effet la grille oblige à avoir des communications non-bloquantes du fait de la latence énorme et permet de tester une implémentation de gestion élastique des ressources. De plus, si notre code peut s’installer facilement sur les machines virtuelles du *cloud*, c’est un bon signe de portabilité. Enfin, l’utilisation du *cloud* du fait de sa grande hétérogénéité, peut être intéressante durant le processus de développement notamment pour effectuer des tests de non-régression sur différentes architectures.

### 3.2.2.1 Déroutement d’un calcul

Pour faire du *cloud computing*, il faut créer une image de *machine virtuelle* (VM) contenant le(s) code(s) de calcul. Plusieurs VMs serontinstanciées via un ordonnanceur. Ainsi, le lancement d’un calcul dans un *cloud* ou sur un supercalculateur est peu différent : dans le *cloud*, le *job* qui est mis en queue est une instanciation de machine virtuelle. Lorsque la VM démarre, elle exécute automatiquement le programme et la VM est détruite après la terminaison du programme.

Nous avons créé un petit serveur XML-RPC en Python [10] afin communiquer avec ces VMs dès leur démarrage. Le serveur se charge de distribuer différents fichiers d’entrée au démarrage des VMs<sup>43</sup>. Nous avons donc créé une VM généraliste qui à chaque démarrage se connecte au serveur qui lui communiquera son fichier d’entrée. La récupération des résultats s’effectue grâce à des *rsync* fréquents vers le serveur.

### 3.2.2.2 Résultats

Dans un premier temps de nombreux *runs* indépendants de la méthode *Configuration Interaction using Perturbative Selection done Iteratively* (CIPSI) furent réalisés sans difficultés, validant la communication avec notre serveur. Notre plus grande réussite fut le lancement d’un calcul QMC utilisant simultanément deux centres de France Grilles ainsi que notre ordinateur de bureau. Il est intellectuellement amusant de se rendre compte qu’une seule simulation peut être distribuée sur des ma-

42. LAL/GRIF, Orsay; IPHC, Strasbourg; CC-IN2P3, Villeurbanne; IRIT, Toulouse; LUPM, Montpellier.

43. Nous ne voulons pas créer une VM par calcul à effectuer.

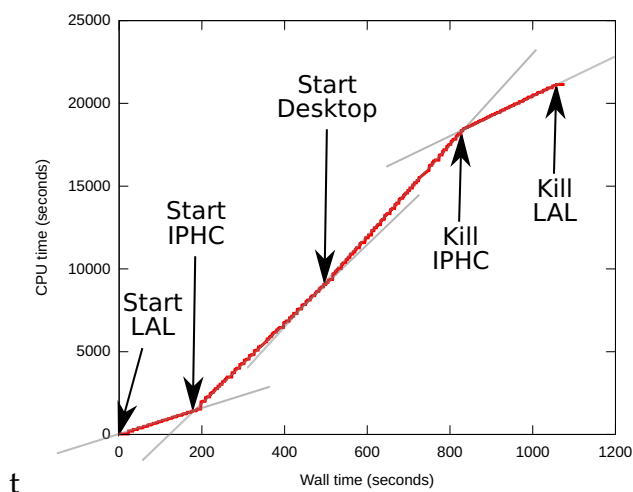


FIGURE 11 – Adding some resources dynamically to a QMC=Chem simulation. The y axis represents the total CPU time.

chines séparées par plusieurs centaines de kilomètres. De plus, les ressources furent mises à disposition de façon élastique comme le montre la Figure 11. Ceci a démontré la capacité de notre code QMC=Chem à bénéficier de ressources modulaires. En outre, de plus nombreux *benchmarks* ont été réalisés [110] afin de comparer l’efficacité des codes sur les différentes machines – virtuelles ou non. Les conclusions sont que nous bénéficions de cette hétérogénéité proposée par le *cloud*. En effet, le Quantum Package a besoin d’avoir accès à des instructions CPU récentes<sup>44</sup>. L’IPHC propose des machines virtuelles donnant un accès direct au processeur (sans virtualisation), ce qui permet au Quantum Package d’avoir de bonnes performances. À l’inverse, le code QMC est un code *CPU intensif* et seule la fréquence est importante, rendant le LAL plus adapté.

### 3.2.2.3 Conclusion

À l’heure actuelle, calculer dans le *cloud* pour des calculs de production n’est pas compétitif en comparaison des supercalculateurs qui permettent de mobiliser des ressources très importantes dans un délai très court<sup>45</sup>. Cependant, cet aspect va certainement changer à l’avenir et il est intéressant d’envisager que le nuage pourra servir de complément de ressources pour les calculs massivement parallèles. De plus, la modification des algorithmes afin qu’ils puissent permettre aux applications de tourner dans le *cloud* aura des conséquences bénéfiques sur les supercalculateurs ; de ce fait et dans une optique de passage à l’échelle, l’infonuagique est d’une grande utilité pour améliorer les codes. Enfin,

44. Particulièrement, l’instruction `popcount` ; nous en parlerons plus en détails dans la section 3.3.

45. Le temps de restitution doit être raisonnable.

le *cloud* semble être un très bon candidat pour l'aide au processus de développement<sup>46</sup>, dont nous parlerons au chapitre suivant.

### 3.3 APPLICATION : CIPSI & QUANTUM PACKAGE

Dans cette partie, nous examinons les points chauds de l'algorithme CIPSI présentés dans le chapitre 2 précédent, les moyens utilisés pour les contourner et leur implémentation dans notre code open-source, *Quantum Package*. Ainsi nous discuterons le problème du stockage des intégrales moléculaires puis nous nous intéresserons au processus de sélection dans cet algorithme, ce qui nous amènera à nous poser la question de la meilleure représentation en machine des déterminants. Enfin, nous dirons quelques mots sur le comment de la parallélisation de ce code de calcul.

#### 3.3.1 Adressage des intégrales

Dans un code de chimie quantique, il est nécessaire de stocker les intégrales moléculaires. Il s'agit d'un tableau à quatre indices (pq|rs) contenant en général beaucoup de valeurs – presque – nulles. Il semble donc naturel de les stocker dans un format dit *creux* où seuls les éléments non nuls seront présents.

Informatiquement et de façon générale, afin de stocker des données il faut créer une fonction d'adressage permettant de faire la liaison entre une clé, dans notre cas le quadruplet (pq|rs), et une adresse mémoire contenant la valeur associée, par exemple la valeur de l'intégrale moléculaire. Dans le cas d'un format de type creux, nous nous assurerons que seules les intégrales non nulles bénéficieront d'une clé.

**DICTIONNAIRES ET AFFILIÉS.** Les formats de données les plus utilisés pour stocker des valeurs associées à une clé sont soit les tables de hachage<sup>47</sup> –insertion en  $\mathcal{O}(1)$ , recherche en  $\mathcal{O}(1)$  – soit les arbres binaires de recherche<sup>48</sup> –insertion  $\mathcal{O}(\log(N))$ , recherche en  $\mathcal{O}(\log(N))$ . Une première implémentation du *Quantum Package* utilisait ces deux types de données. Néanmoins cette façon de stocker les intégrales a plusieurs inconvénients :

- Même si le temps d'accès est théoriquement constant avec la table de hachage, le fait que l'on n'accède jamais à des intégrales contiguës en mémoire pose d'énormes problèmes de performance dus

<sup>46</sup>. Lancement des tests de non-régression, tests concernant la portabilité du code, etc.

<sup>47</sup>. Aussi appelées *dictionnaire* en Python.

<sup>48</sup>. Aussi appelés *map* dans la bibliothèque standard C++.

à la latence des accès aléatoires rendant le temps d'accès à une intégrale important ;

- Les tables de hachage ont une taille définie *a priori* ; ainsi une insertion peut nécessiter le redimensionnement de cette table, ce qui engendre à son tour une modification de la fonction de hachage et nécessite alors de redistribuer les données dans le nouveau tableau. Cette étape étant coûteuse pour les grands tableaux, elle impacte l'efficacité ;
- De plus dans les tables de hachage et les arbres binaires de recherche, les insertions ne peuvent pas être effectuées facilement en parallèle.

Ainsi il a été choisi d'abandonner ce type de format et de développer notre propre structure de données.

**TABLEAU DE LISTES.** Au vu des défauts des tables de hachage, il convient de définir les critères que devront remplir notre fonction d'adressage :

- On doit pouvoir accéder rapidement aux intégrales. Il faut donc que ces intégrales soient contiguës en mémoire afin de bénéficier des effets de cache. En effet, ce n'est pas le coût du calcul de l'adresse mémoire qui est important mais le temps d'accès à la valeur via la mémoire<sup>49</sup> ;
- La création des intégrales moléculaires est un point chaud du *Quantum Package*. L'insertion des intégrales dans notre format de données doit donc pouvoir se faire rapidement et en parallèle.

Afin de satisfaire toutes ces contraintes nous allons utiliser un tableau de listes. L'idée fondamentale est de répartir les intégrales en nombreux petits sous-groupes, appelés blocs. Ce découpage en blocs permet d'une part l'insertion en parallèle, chaque bloc pouvant être modifié par un processus indépendant ; et une utilisation efficace des caches du fait de leur petite taille. Il nous faut maintenant définir la clé et la fonction de décryptage de cette clé afin d'arriver à l'adresse mémoire de l'intégrale (pq|rs).



FIGURE 12 – Resarch of an integral key

**ALGORITHMIQUE.** Pour faire cela, le quadruplet (pq|rs) est transformé en un entier unique sur 64 bits. Les 49 bits de poids fort donnent di-

49. Voir *Tableau 9* sur les différentes latence.



```

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

```

CODE 5 – Determinant driven algorithm

rectement accès au bloc. Les 15 bits de poids faible quant à eux codent pour un entier  $k$  qui permettra de retrouver l'indice de l'intégrale dans le bloc. Il faut maintenant faire correspondre cet entier  $k$  à la valeur de l'intégrale. Pour cela deux listes sont nécessaires : la première contiendra à un indice  $i$  le nombre  $k$  et l'autre aura au même indice  $i$  la valeur de l'intégrale.

Puisque la recherche dans le bloc s'effectue sur des entiers contenant les 15 bits de poids faible, on peut les représenter sur 2 octets. Ainsi, les blocs seront dimensionnés afin d'accueillir au maximum 32 768 éléments, et la recherche de la clé aura lieu dans un tableau de 64Ko au maximum. La totalité de ce tableau peut tenir dans le cache L2 (256Ko le plus souvent), et une grande partie peut tenir dans le cache L1 (32Ko). Si la liste est triée, on peut trouver le nombre  $k$  très rapidement par dichotomie (voir Figure 12), en 14 sauts au maximum. Les blocs sont donc triés en parallèle et en temps linéaire après le calcul de toutes les intégrales via l'algorithme RADIX SORT [3].

CONCLUSION. Cette façon de stocker les intégrales de façon creuse permet un accès rapide dû au cache<sup>50</sup> et une insertion en parallèle<sup>51</sup>.

### 3.3.2 Construction de la matrice hamiltonienne

Une autre point chaud de l'algorithme CIPSI est la construction de la matrice hamiltonienne. Dans notre cas nous utilisons un algorithme de type *Determinant driven*. Cela consiste à regarder pour chaque paire de déterminants contenus dans la fonction d'onde sa contribution à la matrice hamiltonienne. Si les déterminants forment une base orthonormale, on peut utiliser les règles de SLATER-CONDON [40, 115] pour obtenir ces contributions (CODE 5). Il nous faut donc une implémentation de ces règles la plus efficace possible.

Nous commencerons par expliciter ces règles, puis nous parlerons plus en détail de leur implémentation.

50. À la condition ne pas utiliser trop souvent des intégrales situées dans des blocs lointains, bien entendu.

51. Chaque bloc peut être calculé indépendamment.

MATHÉMATIQUES. Soit :

- Un déterminant de Slater

$$D = \begin{vmatrix} \phi_1(\mathbf{r}_1) & \dots & \phi_{N_\alpha}(\mathbf{r}_1) \\ \vdots & \ddots & \vdots \\ \phi_1(\mathbf{r}_{N_\alpha}) & \dots & \phi_{N_\alpha}(\mathbf{r}_{N_\alpha}) \end{vmatrix} \quad (20)$$

et  $D_m^p$  un déterminant ne différant de  $D$  que par la seule orbitale  $p$ , et  $D_{mn}^{pq}$  un déterminant différant par deux orbitales  $p$  et  $q$ .

- $\hat{O}_1$  et  $\hat{O}_2$  des opérateurs mono- (par exemple l'énergie cinétique) et bi-électroniques (par exemple l'énergie potentielle).

Alors les règles de SLATER-CONDON s'écrivent :

$$\langle D | \hat{O}_1 | D \rangle = \sum_{i \in D} \langle \phi_i | \hat{O}_1 | \phi_i \rangle \quad (21a)$$

$$\langle D | \hat{O}_2 | D \rangle = \frac{1}{2} \sum_{(i,j) \in D} \langle \phi_i \phi_j | \hat{O}_2 | \phi_i \phi_j \rangle - \langle \phi_i \phi_j | \hat{O}_2 | \phi_j \phi_i \rangle \quad (21b)$$

$$\langle D | \hat{O}_1 | D_m^p \rangle = \langle \phi_m | \hat{O}_1 | \phi_p \rangle \quad (21c)$$

$$\langle D | \hat{O}_2 | D_m^p \rangle = \sum_{k \in D} \langle \phi_m \phi_k | \hat{O}_2 | \phi_p \phi_k \rangle - \langle \phi_m \phi_k | \hat{O}_2 | \phi_k \phi_p \rangle \quad (21d)$$

$$\langle D | \hat{O}_1 | D_{mn}^{pq} \rangle = 0 \quad (21e)$$

$$\langle D | \hat{O}_2 | D_{mn}^{pq} \rangle = \langle \phi_m \phi_n | \hat{O}_2 | \phi_p \phi_q \rangle - \langle \phi_m \phi_n | \hat{O}_2 | \phi_q \phi_p \rangle \quad (21f)$$

Je ne rentrerai pas ici dans l'origine physique et l'implémentation complète de ces règles<sup>52</sup>, je me permettrai cependant de faire un rapide commentaire motivé par la beauté de cette implémentation. Les algorithmes dits « guidés par les déterminants » (*determinant driven*), comme celui proposé ici, n'ont pas bonne presse ; on leur reproche leur inefficacité et on leur préfère des algorithmes « guidés par les intégrales » (*integral driven*)<sup>53</sup>. Cette critique était pertinente il y a encore quelques années, mais nous allons voir que, grâce à de nouvelles instructions, il est maintenant possible d'avoir une implémentation très efficace. C'est un exemple très intéressant de comment une révolution au niveau de l'architecture des processeurs, peut impacter les aspects plus théoriques des approches.

Le calcul du degré d'excitation – c'est à dire le nombre d'orbitales permutées entre deux déterminants – est le point central de cet algorithme, du fait de l'existence des différentes règles en fonction de ce critère. Nous verrons comment ce calcul est implémenté et la représentation des déterminants qui y est associée.

52. Leur implémentation détaillée peut être trouvée dans la référence [108].

53. On calcule pour chaque intégrale sa contribution à l'hamiltonien.

```

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0000000011111111 : D1
0010010010101111 : D2
0010010001010000 : D1 xor D2

```

FIGURE 13 – Bitmask xor operation : Positions of holes and particles

**REPRÉSENTATION DES DÉTERMINANTS** Les déterminants sont définis comme un produit antisymétrisé d'orbitales moléculaires. Ainsi on peut simplement écrire les déterminants comme une suite de nombres booléens reflétant l'occupation de l'orbitale considérée : 0 pour inoccupée, 1 pour occupée. Dans cette représentation, les lignes et colonnes des déterminants sont toujours ordonnées par indice croissant des orbitales moléculaires. Ainsi **0000011111** correspond à un déterminant *Hartree-Fock*, comme le déterminant  $D_1$  dans la Figure 13, le déterminant  $D_2$  quant à lui correspond à une double excitation à partir du déterminant *Hartree-Fock* – il y a deux permutations dans cette paire de déterminants.

**CALCUL DU DEGRÉ D'EXCITATION.** Si l'on souhaite calculer le degré d'excitation de façon naïve, on doit comparer un à un chaque bit de la paire de déterminants considérée ( $2 \times N_{\text{MO}}$  cycles CPU), puis diviser ce nombre par deux. Vu qu'il n'est pas rare d'utiliser quelques millions de déterminants, ce calcul devient vite un goulot d'étranglement. Mais depuis 2008 et le jeu d'instruction SSE4.2, il est possible de faire ce calcul de degré d'excitation 64 fois plus vite :

**XOR** On commence par effectuer le *ou exclusif* (1 cycle CPU) entre deux déterminants, ce qui donne l'emplacement des trous et des particules créés. La Figure 13 illustre ce propos ;

**POPCOUNT** Puis on compte le nombre de bits *up* de cette liste. C'est ce calcul qui est maintenant effectué directement en *hardware*<sup>54</sup> [101] permettant de l'effectuer en 1 cycle CPU en moyenne ;

**BIT SHIFT** Enfin, on divise ce nombre par deux afin d'avoir le degré d'excitation. La division d'un entier positif par deux peut se faire en 1 cycle CPU, il suffit pour cela de décaler tous les bits vers la droite.

La Figure 14, illustre que dans le Quantum Package qui est le code implémentant le CIPSI, c'est l'instruction `popcount` qui est utilisée la majorité du temps CPU. Pour être tout à fait franc le calcul du degré d'excitation n'est principalement pas utilisé pour le calcul de la matrice hamiltonienne mais plutôt dans l'étape de sélection que nous décrirons dans la section suivante.

54. Certaines personnes soupçonnent la NSA d'avoir fait pression pour avoir cette instruction [45].

```

:      if (Nint==1) then ! Determinants/filter_connected
:      !DIR$ LOOP COUNT (1000) ! Determinants/filter_connected
:      do i=1,size ! Determinants/filter_connected
:      degree_x2 = popcnt(xor( key1(1,1,i), key2(1,1))) + &
23.24 : 447fc1: popcnt %rsi,%r14
:      popcnt(xor( key1(1,2,i), key2(1,2))) ! Determinants/filter_connected
0.72 : 447fc6: popcnt %r13,%r15
:      integer :: degree_x2 ! Determinants/filter_connected
:      l=1 ! Determinants/filter_connected
:      if (Nint==1) then ! Determinants/filter_connected
:      !DIR$ LOOP COUNT (1000) ! Determinants/filter_connected
:      do i=1,size ! Determinants/filter_connected
:      degree_x2 = popcnt(xor( key1(1,1,i), key2(1,1))) + &
0.44 : 447fcb: add %r15d,%r14d
:      popcnt(xor( key1(1,2,i), key2(1,2))) ! Determinants/filter_connected
:      if (degree_x2 > 4) then ! Determinants/filter_connected
24.05 : 447fce: cmp $0x4,%r14d
0.00 : 447fd2: jg 447fe0 <filter_connected_i_h_psi0+0x2f0>
:      cycle ! Determinants/filter_connected
:      else if (degree_x2 .ne. 0) then ! Determinants/filter_connected
1.41 : 447fd4: test %r14d,%r14d
0.00 : 447fd7: je 447fe0 <filter_connected_i_h_psi0+0x2f0>
:      idx(l) = i ! Determinants/filter_connected
0.59 : 447fd9: mov %r9d, (%r8,%rcx,4)
:      l = l+1 ! Determinants/filter_connected
0.89 : 447fdd: inc %rcx
:      integer :: i,l,m ! Determinants/filter_connected
:      integer :: degree_x2 ! Determinants/filter_connected
:      l=1 ! Determinants/filter_connected
:      if (Nint==1) then ! Determinants/filter_connected
:      !DIR$ LOOP COUNT (1000) ! Determinants/filter_connected
:      do i=1,size ! Determinants/filter_connected
23.69 : 447fe0: inc %r9
0.00 : 447fe3: add %rbx,%r10
0.00 : 447fe6: cmp %rdx,%r9
0.00 : 447fe9: jle 447fb3 <filter_connected_i_h_psi0+0x2c3>

```

FIGURE 14 – popcount use in Quantum Package

CONCLUSION. Il est maintenant possible de calculer le degré d'excitation entre deux déterminants en quelques cycles CPU, ce qui donne un souffle nouveau aux algorithmes *determinant driven*.

### 3.3.3 Problématique de l'unicité

Le dernier point chaud de l'algorithme CIPSI est la gestion de l'unicité de l'ensemble des déterminants générés au cours du processus. Il s'agit plus d'un problème algorithmique qu'informatique. Comme nous l'avons vu dans le chapitre 2, l'application de  $\mathcal{H}$  produit un ensemble de déterminants. Une fois cette liste de déterminants unifiée, on obtient l'espace  $\mathcal{P}$ . Dans cet espace on sélectionne les déterminants ayant la plus grande contribution énergétique et on les ajoute à l'espace  $\mathcal{V}$ , puis on itère.

L'un des problèmes est l'unification de cette liste intermédiaire. En effet :

- Cet espace est beaucoup trop grand pour être stocké. Ainsi on ne peut pas l'unifier *a posteriori* ;
- À l'inverse, on pourrait sélectionner un déterminant et vérifier si on ne l'a pas déjà ajouté à la fonction d'onde. Le problème est que du fait de la parallélisation, les *threads* n'ont accès qu'à leur

propre plage de déterminants dans  $\mathcal{P}$ . Ainsi, une *thread* pourrait choisir de rajouter un déterminant unique pour elle, mais pas pour l'ensemble des *threads*.

Néanmoins il est possible de contourner le problème de la deuxième stratégie. L'astuce est de remarquer que chaque *thread* a accès à l'ensemble des déterminants de l'espace variationnel. Si un déterminant perturbateur est une simple ou une double excitation par rapport à un déterminant de l'espace variationnel, c'est que ce déterminant peut le générer. Ainsi grâce à l'utilisation du `popcount` ce test peut être fait sur tous les déterminants de l'espace variationnel très rapidement.

ALGORITHME. À une *thread* donnée est associée un déterminant sur lequel appliquer  $\mathcal{H}$ , et son indice dans la liste des déterminants de l'espace variationnel. Pour chaque déterminant perturbatif créé, on calcule son degré d'excitation par rapport à l'ensemble de déterminants d'indice inférieur. Si le degré d'excitation n'est pas supérieur à 2 cela signifie que ce déterminant peut être généré par le déterminant d'indice inférieur, donc on ne le considère pas maintenant pour la sélection. Sinon, on sélectionne ou non le déterminant en fonction de sa contribution perturbative à l'énergie. Cette façon de faire permet une parallélisation efficace de la sélection de déterminants avec un équilibrage de charge dynamique : lorsqu'une *thread* a terminé la sélection par rapport à un déterminant de l'espace variationnel, elle prend le déterminant suivant de l'espace variationnel qui n'est pas en train d'être traité par une autre *thread*.

### 3.3.4 Parallélisation utilisant OpenMP

Dans `Quantum Package`, l'étape de sélection et du calcul de la contribution perturbative à l'énergie ( $E_{PT2}$ ) sont parallélisées avec `OPENMP` pour le moment. Dans un futur proche, des ressources humaines seront apportées afin d'utiliser la bibliothèque `OMQ`, ce qui permettra d'utiliser plusieurs nœuds de calcul. Néanmoins, l'efficacité parallèle est très bonne. La [Figure 15](#) montre un exemple où nous avons fait tourner deux calculs utilisant chacun 180 cœurs simultanément avec la machine UV-PROD de CALMIP.

### 3.3.5 Conclusion

Nous ne devons pas donc vivre dans notre tour d'airain théorique, déconnectés des évolutions purement matérielles. Le jeu d'instructions SSE4.2, cette petite révolution technologique, a permis de faire revivre un algorithme qui était tombé en désuétude. Ainsi, le monde théorique

```
top - 14:04:14 up 14 days, 22:43, 1 user, load average: 329.54, 307.65, 325.17
Tasks: 2690 total, 3 running, 2686 sleeping, 0 stopped, 1 zombie
Cpu(s): 95.5%us, 3.1%sy, 0.0%ni, 1.4%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Mem: 3101742M total, 540847M used, 2560895M free, 293M buffers
Swap: 8191M total, 0M used, 8191M free, 425631M cached
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	P	COMMAND
234124	scemama	20	0	18.4g	9.8g	7196	R	18032	0.3	167678:08	194	full_ci
161321	scemama	20	0	13.1g	2.7g	6956	R	18004	0.1	280:36.59	2	full_ci
161311	scemama	20	0	11404	3724	932	R	9	0.0	0:13.95	1	top
1206	root	20	0	0	0	0	S	1	0.0	1:36.72	319	kworker/319:1
9	root	20	0	0	0	0	S	0	0.0	0:01.02	1	ksoftirqd/1

FIGURE 15 – Quantum Package using OpenMP

des algorithmes a un recouvrement non nul avec celui, plus pragmatique, des aspects matériels.

### 3.4 APPLICATION : QMC ET QMC=CHEM

QMC=Chem est un code de calcul Monte Carlo quantique ayant vocation à tourner sur un nombre de processeurs aussi grand que possible tout en ayant une accélération (*scaling*) quasi-linéaire avec le nombre de processeurs. Premièrement, nous parlerons donc des optimisations du processus monocœur permettant de tirer partie au mieux des processeurs actuels, et dans une deuxième partie comment la parallélisation a été implémentée dans le but de gérer un grand nombre de connexions.

#### 3.4.1 Optimisation du processus monocœur

Nous allons commencer par aborder les points chauds de l'algorithme *Fixed-Node Diffusion Monte Carlo* (FN-DMC). À chaque pas Monte Carlo (des millions, voire des milliards de pas) il est nécessaire de calculer la valeur de la fonction d'onde (Équation 22b) ainsi que de ses dérivées (Équation 22c et Équation 22d) pour chaque ensemble de positions pour les particules :

$$D_k = \begin{vmatrix} \phi_{k_1}(\mathbf{r}_1) & \dots & \phi_{k_{N_\alpha}}(\mathbf{r}_1) \\ \vdots & \ddots & \vdots \\ \phi_{k_1}(\mathbf{r}_{N_\alpha}) & \dots & \phi_{k_{N_\alpha}}(\mathbf{r}_{N_\alpha}) \end{vmatrix} \quad (22a)$$

$$\Psi_T = \sum_k^{N_\alpha} c_k D_k(\mathbf{r}_1, \dots, \mathbf{r}_{N_\alpha}) \quad (22b)$$

$$\frac{\nabla_i \Psi_T}{\Psi_T} = \frac{1}{\Psi_T(\mathbf{r}_1, \dots, \mathbf{r}_N)} \frac{\partial}{\partial \mathbf{r}_i} \Psi_T(\mathbf{r}_1, \dots, \mathbf{r}_N) \quad (22c)$$

$$\frac{\nabla_i^2 \Psi_T}{\Psi_T} = \frac{1}{\Psi_T(\mathbf{r}_1, \dots, \mathbf{r}_N)} \nabla_i^2 \Psi_T(\mathbf{r}_1, \dots, \mathbf{r}_N) \quad (22d)$$

L'effort numérique pour calculer ces quantités de manière directe est donc linéaire avec le nombre de déterminants.

Considérant le nombre de déterminants que nous utilisons<sup>55</sup> nous ne pouvons pas nous contenter de cette croissance (*scaling*) linéaire. Nous avons développé une méthode permettant d'avoir une croissance du coût de calcul en fonction du nombre de déterminants en  $\sqrt{N_{\text{det}}}$  [106]. Dans un formalisme *spin-free* comme celui utilisé dans les approches Monte Carlo quantique, les déterminants de Slater sont écrits comme le produit de deux déterminants : un déterminant  $\alpha$  et un déterminant  $\beta$ .

$$\Psi_T = \sum_k^{N_{\text{det}}} c_k D_k^\alpha(\mathbf{r}_1, \dots, \mathbf{r}_{N_\alpha}) D_k^\beta(\mathbf{r}_{N_\alpha+1}, \dots, \mathbf{r}_{N_{\text{elec}}}) \quad (23)$$

L'astuce est de remarquer que de nombreux déterminants ( $D^\alpha$  et  $D^\beta$ ) apparaissent dans différents produits. Ainsi si l'on ne stocke pas le produit en lui-même mais que l'on stocke plutôt la liste unique des déterminants  $\alpha$  et  $\beta$ , on peut diminuer drastiquement le nombre de déterminants à calculer. Et le *scaling* peut devenir en  $\mathcal{O}(N_\alpha \sim \sqrt{N_{\text{det}}})$ .

Pour mettre en œuvre cette idée, nous réécrivons l'expression de la fonction d'onde donnée en Équation 23 sous une forme bilinéaire :

$$\Psi_T = \sum_i^{N_{\text{det}\alpha}} \sum_j^{N_{\text{det}\beta}} c_{ij} D_i^\alpha(\mathbf{r}_1, \dots, \mathbf{r}_{N_\alpha}) D_j^\beta(\mathbf{r}_{N_\alpha+1}, \dots, \mathbf{r}_{N_{\text{elec}}}) = \mathbf{D}_\alpha^\dagger \mathbf{C} \mathbf{D}_\beta \quad (24)$$

Nous présenterons dans la suite, les différentes étapes du calcul des quantités nécessaires au déroulement d'un calcul QMC :

1. La première étape est la transformation de la représentation des déterminants dans un format bilinéaire. Cette étape peut se faire lors d'une étape préliminaire qui n'est faite qu'une seule fois (*pre-processing*)
2. Puis nous nous intéresserons à l'évaluation des déterminants de Slater et des matrices de Slater inverses associées.
3. Dans un troisième temps, les techniques employées afin de calculer efficacement les dérivées seront explicitées.
4. Finalement, l'évaluation de la fonction d'onde aura lieu.

#### 3.4.1.1 Format de stockage de la fonction d'onde

Afin de passer de la forme standard des produits de déterminants  $\alpha$  et  $\beta$  (Équation 23) à la forme bilinéaire (Équation 24), il faut dans un premier temps créer la liste unique des  $N_{\text{det}\alpha}$  et  $N_{\text{det}\beta}$  déterminants  $\alpha$  et

55. De l'ordre de 1000 à 100 000 déterminants, voire plus.



```

do k=1,det_num
  i = C_rows(k)
  j = C_columns(k)

  print*, "Value of coef", C(k)
  print*, "for the alpha determinant i", Da(i)
  print*, "and the beta determinant j", Db(j)
end do

```

CODE 6 – How to get determinants

$\beta$ , que l'on appellera  $D_\alpha$  et  $D_\beta$ ); cette étape ne pose pas de difficultés particulières.

Puis nous devons créer la matrice des coefficients. Cette matrice sera toujours très creuse, il convient donc de la stocker dans un format adapté. On choisit le *coordinate format* qui est composé de trois listes (`C`, `C_rows`, `C_columns`) contenant respectivement l'indice du déterminant  $\alpha$ , l'indice du déterminant  $\beta$  et la valeur du coefficient associé à ces deux déterminants. On parcourra la liste des déterminants comme montré CODE 6.

Les éléments des listes associées à la matrice `C` sont ordonnés de façon à ce que lire la liste de façon séquentielle corresponde à lire la matrice colonne par colonne<sup>56</sup> dans le but de maximiser les effets bénéfiques liés au cache via les *prefetchers*.

#### 3.4.1.2 Calcul des déterminants de Slater et des matrices inverses de Slater

Il est maintenant nécessaire de calculer les déterminants, ainsi que la matrice de Slater inverse. En effet cette matrice inverse est utilisée pour calculer efficacement les dérivées des déterminants. Au lieu de recalculer pour chaque déterminant sa matrice inverse – ce qui coûte de l'ordre de  $N_\alpha^3$  flops –, on peut la mettre à jour rapidement<sup>57</sup>. La seule condition est de connaître la matrice inverse d'un déterminant proche du déterminant qui nous intéresse.

L'algorithme traditionnel pour calculer un déterminant et l'inverse d'une matrice est la décomposition LU qui permet de réécrire la matrice de Slater ( $S$ ) comme le produit d'une matrice de permutation ( $P$ ) et de deux matrices ( $L$  et  $U$ ) respectivement triangulaire inférieure et supérieure. Soit :

$$S = PLU \quad (25)$$

56. Les trois listes sont ordonnées par  $N_{\text{det}\alpha} \times (j - 1) + i$ .

57. En  $\mathcal{O}(N_\alpha^2)$  flops.



Ce qui permet de calculer ensuite trivialement le déterminant de la matrice :

$$D = \det(S) = \det(P) \det(L) \det(U) = (-1)^s \left( \prod_{i=1}^n l_{ii} \right) \left( \prod_{i=1}^n u_{ii} \right) \quad (26)$$

Où  $s$  est le nombre de permutations présentes dans la matrice  $P$ . Cette décomposition LU est en  $\mathcal{O}(N^3)$ . Elle est implémentée dans la bibliothèque `lapack` sous le nom de `dgetrf`.

**REMARQUE SUR LES COMPLEXITÉS.** Il est intéressant de remarquer que l'algorithme naïf de calcul d'un déterminant est en  $\mathcal{O}(N!)$ , et donc que si l'on a moins de 6 électrons  $\alpha$  ou  $\beta$ , l'algorithme naïf est plus rapide que la décomposition LU.<sup>58</sup> Ainsi nous avons, grâce à un script, écrit explicitement toutes les instructions nécessaires au calcul des déterminants de matrices de tailles  $1 \times 1$  à  $5 \times 5$  avec l'algorithme naïf. La première matrice inverse se calcule quand à elle grâce à la routine `lapack dgetri` en  $\mathcal{O}(N^3)$ , puis les déterminants sont mis à jour grâce à l'algorithme de SHERMAN-MORRISON en  $\mathcal{O}(N^2)$ . Comme dans le cas de la décomposition LU, on peut remarquer que pour des matrices  $2 \times 2$  et  $3 \times 3$ , l'algorithme naïf est plus rapide que SHERMAN-MORISON, c'est donc cet algorithme que nous utilisons. De plus, comme nous l'avons vu précédemment, si nous écrivons une implémentation de l'algorithme de SHERMAN-MORISON générale, le compilateur ne pourra pas vectoriser les boucles de façon optimale car les matrices sont toujours petites<sup>59</sup>. En effet, ne connaissant pas les nombres de tours des boucles à la compilation, tout devrait être fait pendant l'exécution produisant un surcoût non négligeable. Pour éviter cela, nous utilisons des routines spécifiques avec toutes les bornes écrites en dur pour les matrices de  $4 \times 4$  à  $20 \times 20$ <sup>60</sup>; ceci permet aux boucles les plus chaudes d'être vectorisées à 100% dès la compilation, ce qui a été vérifié grâce à l'utilitaire `maqao` [48].

58.  $5! = 120 < 5^3 = 125$ .

59. Par petite, j'entends au maximum  $\leq 20 \times 20$ .

60. Ces routines sont facilement créées à l'aide de *templates*.

### 3.4.1.3 Calcul des dérivées (gradients et laplacien)

Il nous faut maintenant calculer les dérivées. L'expression bilinéaire pour  $\Psi_T$ , donne les dérivées suivantes :

$$\nabla_{x,i}\Psi = (\nabla_{x,i}\mathbf{D}_\alpha)^\dagger \mathbf{C}\mathbf{D}_\beta + \mathbf{D}_\alpha^\dagger \mathbf{C}(\nabla_{x,i}\mathbf{D}_\beta) \quad (27a)$$

$$\begin{aligned} \Delta_i\Psi = & (\Delta_i\mathbf{D}_\alpha)^\dagger \mathbf{C}\mathbf{D}_\beta + \mathbf{D}_\alpha^\dagger \mathbf{C}(\Delta_i\mathbf{D}_\beta) \\ & + 2(\nabla_{x,i}\mathbf{D}_\alpha)^\dagger \mathbf{C}(\nabla_{x,i}\mathbf{D}_\beta) \\ & + 2(\nabla_{y,i}\mathbf{D}_\alpha)^\dagger \mathbf{C}(\nabla_{y,i}\mathbf{D}_\beta) \\ & + 2(\nabla_{z,i}\mathbf{D}_\alpha)^\dagger \mathbf{C}(\nabla_{z,i}\mathbf{D}_\beta) \end{aligned} \quad (27b)$$

Il est intéressant de remarquer dans l'expression du laplacien (Équation 27b) que les termes  $\nabla_i\mathbf{D}_\alpha$  sont nuls si  $i$  est un électron  $\beta$ , et de façon similaire pour  $\nabla_i\mathbf{D}_\beta$ . Ainsi tous les termes croisés contenant à la fois les gradients  $\nabla_i\mathbf{D}_\alpha$  et  $\nabla_i\mathbf{D}_\beta$  sont toujours nuls. Les quatre quantités  $[\nabla_x, \nabla_y, \nabla_z, \Delta]$  peuvent donc être calculées par la même formule générale :

$$\tilde{\nabla}_i\Psi = (\tilde{\nabla}_i\mathbf{D}_\alpha)^\dagger \mathbf{C}\mathbf{D}_\beta + \mathbf{D}_\alpha^\dagger \mathbf{C}(\tilde{\nabla}_i\mathbf{D}_\beta) \quad (28)$$

Où  $\tilde{\nabla}_i$  est un vecteur regroupant les quatre quantités. Cette opération peut être effectuée en parallèle grâce au jeu d'instruction **SIMD**, permettant une vectorisation parfaite.

- $\tilde{\nabla}\mathbf{D}_j^\alpha$  est calculé grâce à  $\tilde{\nabla}\Phi$  et la matrice de Slater inverse calculée précédemment :

$$\tilde{\nabla}_i\mathbf{D}_j^\alpha = \sum_k S_{ik}^{\alpha-1} \tilde{\nabla}\Phi_{ki}^\alpha \quad (29)$$

- Il nous reste à calculer  $(\nabla\mathbf{D}_\alpha)^\dagger \cdot (\mathbf{C}\mathbf{D}_\beta)$  et  $(\mathbf{D}_\alpha^\dagger \mathbf{C}) \cdot (\nabla\mathbf{D}_\beta)$

### 3.4.1.4 Calcul de $\Psi$

On peut remarquer que les produits de matrices  $\mathbf{D}_\alpha^\dagger \mathbf{C}$  et  $\mathbf{C}\mathbf{D}_\beta$  sont présents à la fois dans l'Équation 24 utilisée pour calculer  $\Psi$ , et dans l'Équation 28 utilisée pour  $\tilde{\nabla}\Psi$ . On peut donc réutiliser des données. De plus, au vu des produits que l'on doit effectuer (vecteur dense par matrice creuse) on aura forcément une faible intensité arithmétique. Cette partie sera donc limitée par l'accès à la mémoire, rendant l'optimisation du cache obligatoire si l'on veut avoir de bonnes performances. On peut effectuer le calcul comme suit :

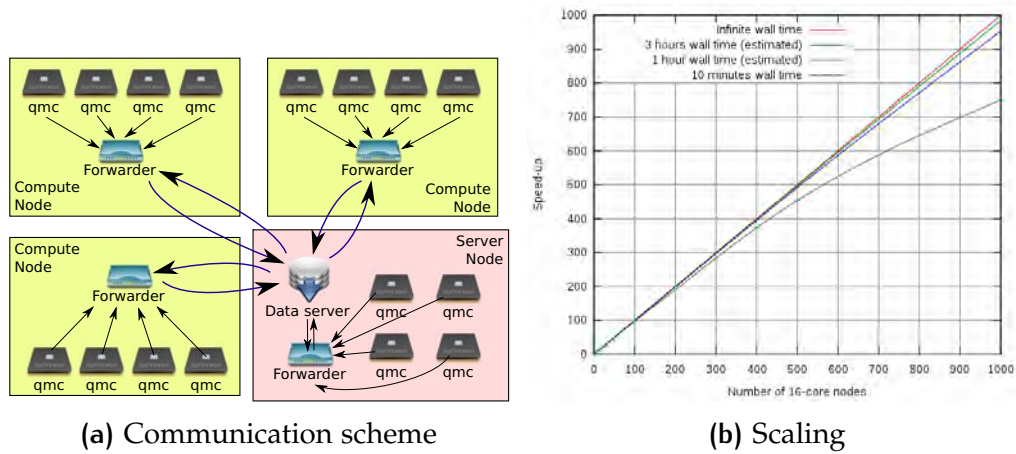


FIGURE 16 – QMC=Chem parallelisation overview

```

do k=1,det_num
  i = C_rows(k)
  j = C_columns(k)
  Da_C(j) = Da_C(j) + C(k)*Da(i)
  C_Db(i) = C_Db(i) + C(k)*Db(j)
end do

```

De cette façon les trois listes dépendant de l'indice  $k$  sont récupérées de la mémoire une seule fois. De plus, comme on accède de façon prédictive à ces listes, les *prefetchers* permettent de diminuer drastiquement la latence et de travailler au maximum de la bande passante. En outre, du fait de notre tri des indices l'indice  $j$  a de fortes chances d'être constant d'une itération à l'autre, ce qui permet de laisser  $C(j)$  et  $Db(j)$  dans les registres CPU au nom de la localité temporelle. Finalement les listes  $Da$  et  $Db$  ont de fortes chances d'être présentes dans les caches, leur taille étant faible<sup>61</sup>. Asymptotiquement la complexité de ce calcul est en  $\mathcal{O}(N_{\text{det}})$  mais le préfacteur étant tellement petit, cette étape n'est pas limitante même pour 1 000 000 de déterminants.

### 3.4.2 Parallélisation de type client-serveur

Nous avons vu dans les parties précédentes que l'algorithme utilisé par QMC=Chem est *embarrassingly parallel*. Il consiste à effectuer une multitude de tâches indépendantes (*workers*), de récupérer les valeurs des quantités à évaluer pour les moyenner. Chaque tâche indépendante a juste besoin de fichiers d'entrée et de points de départ pour sa simulation Monte Carlo.

- Il n'existe pas de communication bloquante. Ainsi nous voulons utiliser une technologie permettant les communications asynchrones.

61. De l'ordre 25 Ko pour une fonction d'onde à un million de déterminants.

- Nos ressources sont élastiques. Nous voulons pouvoir ajouter des *workers*, où en enlever
- Ce programme à vocation à tourner sur des milliers de cœurs, il doit donc être tolérant aux pannes
- Il doit être possible d'utiliser une infrastructure très hétérogène

MPI n'est pas le mieux adapté pour cette tâche. Nous avons donc choisi la bibliothèque ØMQ et une architecture dérivée du modèle Client-Serveur. Elle est composée de trois parties comme montré Figure 16a :

**TRAVAILLEUR (*worker*)** Il s'agit d'un exécutable Fortran monocœur qui réalise une simulation Monte Carlo. Il reçoit les conditions initiales de son expéditeur et lui envoie ses résultats.

**EXPÉDITEUR (*forwarder*)** Chaque nœud de calcul a un expéditeur. Il s'occupe de lancer les exécutables Fortran monocœur ou de les tuer. Il assure également la collecte des données produites par ses travailleurs, et les transmet au serveur de données. Il peut aussi demander au serveur de lui donner des points de départ afin de les transmettre à ses travailleurs.

**SERVEUR DE DONNÉES (*dataserver*)** Le serveur de données s'occupe de la gestion des expéditeurs. Il peut connaître ceux qui tournent et les arrêter quand besoin est. À n'importe quel moment, un nouvel expéditeur peut s'y connecter afin de demander des points de départ et les fichiers d'input. De plus il sauvegarde les données reçues sur disque.

Le serveur de données et les expéditeurs sont écrits en OCaml. En effet la rigidité des langages fortement typés permet dès la compilation un nombre important de vérifications, limitant l'introduction de bugs. Les travailleurs sont quant à eux écrits en Fortran au nom de l'efficacité maximale.

#### 3.4.2.1 Performance parallèle atteinte

Grâce à cette organisation, on peut atteindre un *speed-up* linéaire avec le nombre de cœurs, comme illustré sur Figure 16b. Pour des temps de calcul assez courts, on voit qu'il n'est pas linéaire. En effet, il existe une partie de notre code qui réduit l'efficacité parallèle : il s'agit du transfert des fichiers d'entrée. C'est une partie vraiment extrêmement petite du temps d'exécution de notre code, mais en vertu de la loi d'AMDAHL plus le nombre de processeurs augmente, plus cela réduit le *speed-up*. Heureusement, pour des temps de calcul plus importants cela est négligeable.



# 4 | DE LA RÉDUCTION DE LA COMPLEXITÉ

4.1	Pourquoi la complexité est un <i>casus belli</i> . . . . .	83
4.1.1	Conditions nécessaires pour un code simple . . . . .	85
4.2	Simplification pour le développeur . . . . .	87
4.2.1	Réduction du temps de Compilation : Ninja . . . . .	88
4.2.2	Réduction de la complexité du cœur calculatoire : IRP . . . . .	93
4.2.3	Système de contrôle de versions : Exemple de Git . . . . .	98
4.2.4	Outils d'aide pour la gestion de projet collaboratif . . . . .	100
4.2.5	Conclusion . . . . .	104
4.3	Simplification pour l'utilisateur . . . . .	104
4.3.1	Tous les fichiers de bases atomiques à disposition . . . . .	104
4.3.2	Du stockage des données . . . . .	109
4.3.3	Interface graphique . . . . .	110
4.4	Note sur les fichiers d'entrée et de sortie . . . . .	113
4.4.1	Le poids de l'histoire . . . . .	113
4.4.2	Du fichier d'entrée . . . . .	114
4.4.3	Fichier de sortie . . . . .	121
4.5	Conclusion : Recette pour résoudre la complexité . . . . .	121

## 4.1 POURQUOI LA COMPLEXITÉ EST UN *casus belli*

Une vaste littérature (BANKER *et al.* [15], WOLFGANG [126], KRAUT & STREETER [84]) est disponible au sujet de la complexité inhérente à la croissance des codes de calcul et de la difficulté de leur maintenance. La maintenance est typiquement décrite comme toutes les modifications effectuées entre deux versions stables. Le coût de maintenance est assez variable, mais on peut estimer [83] que plus de 50% du temps de développement est perdu pour des raisons de maintenance et non pour l'implémentation de nouvelles fonctionnalités : il s'agit de corrections de bugs dans les nouvelles méthodes implémentées, de corrections de régressions (méthodes qui marchaient mais qui ne fonctionnent plus), de problèmes liés à l'installation, de compatibilité avec d'anciennes versions, *etc.* Notons que la croissance de la complexité d'un code de calcul n'est malheureusement pas linéaire. De plus, on doit régulièrement

passer de codes « maison », à des codes devant être installés sur de nombreuses architectures et utilisables par le plus grand nombre, cette transition s'avérant alors délicate. Illustrons quelques problèmes liés à une mauvaise gestion de cette complexité.

**Pour le développeur.** Un développeur dans un monde idéal connaît *toutes* les routines du programme, leurs paramètres, et sait comment les utiliser. Il évite de ce fait toute redondance<sup>1</sup> et les effets de bord indésirables dus à la modification d'une routine<sup>2</sup>. La documentation est mise à jour – à la fois celle à destination des futurs développeurs et celle pour les utilisateurs – à chaque modification. Les noms des variables sont explicites et uniques. Les structures de données sont à la fois lisibles pour l'humain et optimisées pour l'ordinateur. Il n'utilise pas de variables globales mutables pouvant provoquer des effets de bord incontrôlés. Enfin, il n'oublie pas de prévenir tous ses collaborateurs en cas de changement pouvant les impacter<sup>3</sup>, et bien sûr il s'occupe de gérer toutes les fusions entre les différentes versions du code afin d'éviter toute divergence problématique. Finalement, le code se compile rapidement, et le débogage de ce processus de compilation est facile.

Malheureusement dans la plupart des codes scientifiques académiques conséquents, ceci est une vue de l'esprit. L'arbre de dépendances du programme est bien trop compliqué rendant la prévision de la propagation des effets de bord très délicate. Aucun développeur ne connaît exactement dans l'ensemble du code toutes les conséquences engendrées par une modification. Ainsi, puisque l'ensemble des chemins que peut prendre le code avant d'arriver à un certain point et après ce point ne sont pas toujours connus, les développeurs :

- soit re-calculent des valeurs qui ont déjà été calculées dans certains chemins du passé afin de garantir que ces valeurs sont bien calculées avant d'être utilisées ;
- soit calculent certaines valeurs qui doivent absolument être calculées dans certains chemins du futur, même si certains chemins n'en ont pas besoin ;
- soit re-calculent une quantité qui a déjà été calculée ailleurs pour des raisons d'interface. En effet, passer une valeur d'un point à l'autre du code nécessite parfois de rajouter un argument à toute une cascade de fonctions ou routines, et cela peut avoir un impact sur d'autres régions du code qui sont en dehors du chemin que l'on considère.

---

1. C'est-à-dire des variables calculées plusieurs fois.

2. Une modification peut, par cascade, avoir des répercussions non voulues sur d'autres parties du code.

3. Changement du nombre d'arguments d'une routine par exemple.

Le nombre de routines étant trop important et leur utilisation très spécifique, il est malheureusement souvent plus aisé de recoder une routine plutôt que de la comprendre. Ceci a pour conséquence la création d'un cycle pervers d'augmentation du nombre de routines, rendant le débogage assez ardu. Enfin, la documentation à destination de l'utilisateur n'est que fortuitement mise à jour, au contraire de la documentation située dans le code source des routines. En effet, les développeurs ont davantage besoin de cette documentation.

Après avoir passé l'étape de modification des routines de calcul, il reste encore des problèmes. Premièrement, il faut rajouter des routines pour la lecture/écriture des nouveaux paramètres et cela n'est que rarement une partie de plaisir. Ensuite, la compilation, souvent le parent pauvre du développement, prend toujours trop de temps.

Puis, une fois que le code passe l'étape de compilation, le développeur souhaite faire partager aux autres développeurs les modifications, mais malheureusement les codes ont tellement divergé que la fusion des versions est difficile. Enfin, le développeur envoie son code aux différents utilisateurs, et il se rend compte que dans la précipitation – ou par fatigue – il a oublié de passer les tests, ou de rajouter un fichier sa communauté se retrouve maintenant avec une version inutilisable.

Tous ces points ne sont pas une fatalité. Je montrerai dans la suite, différents outils et protocoles permettant de résoudre la plupart des problèmes cités. En effet, une grande partie ma thèse fut consacrée à un grand travail de prospection afin de trouver des outils et des processus de développement permettant un développement efficace. Pour faire de la bonne recherche, il faut avoir de bons outils, bien que le développement de ces outils puisse paraître secondaire.

**Pour l'utilisateur** L'herbe n'est pas plus verte du côté de l'utilisateur. À chaque nouvelle version, il doit repasser par le calvaire de l'installation. Puis il faut relire la documentation afin d'apprendre les nouveaux mots clés. Une fois le programme installé, le calcul lancé, les programmes de post-traitement doivent être souvent changés afin de prendre en compte la nouvelle grammaire du fichier de sortie.

#### 4.1.1 Conditions nécessaires pour un code simple

De nombreux codes scientifiques sont devenus trop imposants et ne peuvent plus se permettre de changer drastiquement afin d'essayer de réduire cette complexité. Pour ma part j'ai eu la chance d'éviter cet écueil ; en effet j'ai travaillé durant ma thèse sur le jeune code *Quantum Package* développé deux ans plus tôt par Emmanuel GINER, Anthony SCEMAMA et Michel CAFFAREL. Grâce à cette jeunesse<sup>4</sup>, j'ai implémenté un grand

---

4. Je tiens ici à remercier l'ouverture d'esprit de M. SCEMAMA.



nombre d'utilitaires, de nouvelles méthodes afin d'accélérer les calculs, retravaillé la structure du code et mis en place des *workflows* permettant de réduire cette complexité. Ceci dans le but d'être à la fois agréable pour les utilisateurs et les développeurs. Pour cela il est nécessaire de remplir quelques conditions que j'explicitai dans les paragraphes suivants.

#### 4.1.1.1 *Prise en main aisée*

Un code doit avoir une prise en main aisée et l'une des premières difficultés est son installation. Une réécriture totale des scripts d'installation a donc été faite.

De plus, un frein à une bonne prise en main est la gestion des Entrées/-Sorties (*Input/Output* en anglais) (I/Os). En effet, pour les développeurs ces routines sont rébarbatives à écrire, et pour les utilisateurs la récupération et la création de paramètres via ces fichiers d'entrée n'est pas facile en général. Ainsi, nous avons décidé d'abandonner le concept de *fichier d'entrée* et de *fichier de sortie*. En effet, les formats de fichiers d'entrée et de sortie peuvent changer d'une version à une autre du programme. Maintenant, les données d'entrée/sortie sont stockées dans une base de données (*Easy Fortran I/O library generator* (EZFIO)) et des scripts permettent de fabriquer à la demande des fichiers d'entrée et sortie temporaires adaptés à la version courante du code. La manipulation des données d'entrée/sortie par une *Application Program Interface* (API) permet d'une part de donner aux utilisateurs une interface proche de celle dont ils ont l'habitude, et d'autre part de créer aisément d'autres interfaces. Nous avons, par exemple, développé un script stockant de nombreux ensembles de fonctions de bases atomiques afin d'éviter à l'utilisateur d'aller les chercher lui-même sur internet. De plus, une interface graphique et une interface web permettant le contrôle du *Quantum Package* ont été développées comme *proof of concept*.

Quant à l'utilisateur, une prise en main aisée signifie une grande modularité. Il doit pouvoir utiliser simplement des variables ou des routines calculées dans différentes parties du code. Ceci est rendu possible grâce au paradigme de programmation de référence implicite au paramètres – *Implicit Reference to Parameters* (IRP).

#### 4.1.1.2 *Diminution du temps de restitution*

L'auteur de logiciel et les utilisateurs partagent une caractéristique commune : leur haine de l'attente. Pour l'auteur il s'agit d'attendre la fin de la compilation, et pour l'utilisateur la fin du calcul. Ainsi un grand travail a été fait pour rationaliser le processus de compilation afin de réduire au maximum le temps de compilation, de s'assurer de son exactitude et de la non redondance. De même, l'intégration des deux nou-

velles théories (utilisation des bases de Slater et de pseudo-potentiels), qui ont été précédemment présentées, s'inscrivaient dans ce schéma.

#### 4.1.1.3 Économie et systématisation

Les développeurs passent beaucoup de temps à effectuer des tâches répétitives, qui sont somme toute automatisables. La création de la documentation pour l'utilisateur en est un exemple, en effet cette documentation est pour la plupart déjà présente dans le code source, pourquoi ne pas l'utiliser ? Nous allons voir que l'IRPF90 permet cela également.

L'IRPF90 permet aussi la création automatique de routines – très pratique pour la création de boucles optimisées pour la vectorisation comme nous l'avons vu dans un précédent chapitre – afin de respecter ce beau principe qu'est le *Don't repeat yourself* (DRY).

De plus l'utilisation d'outils collaboratifs ainsi que de procédures simples et systématiques permet de simplifier<sup>5</sup> le développement communautaire. Ainsi durant ce processus des tests de non-régression sont automatiquement lancés afin de s'assurer que le code n'a pas perdu de fonctionnalité.

## 4.2 SIMPLIFICATION POUR LE DÉVELOPPEUR

Nous arrivons maintenant dans le fabuleux monde du *management* de code, où les termes sont anglais et où les traductions sonnent faux.

La création d'un code est un processus chaotique qu'on espère convergent. Au départ, le code est simple. Un répertoire contenant quelques fichiers de code source et un `Makefile` compréhensible. Puis se rajoutent de nouveaux modules ; de la méta-programmation ; des scripts d'installation et de configuration ; des collaborateurs ; des erreurs ; différentes versions avec différentes fonctionnalités ; des bugs ; des fonctions qui ne marchent plus ; des sous-routines en double, triple, quadruple, et avec un nombre de paramètres ubuesque ; une compilation durant toujours trop longtemps et qui ne fonctionne plus en parallèle à cause de bugs dans le fichier de configuration ; une documentation absente ; *etc.*

Pourtant rien de ceci n'est une fatalité. Nous proposons un passage en revue de différents outils permettant d'échapper à ces écueils et de rester heureux comme SISYPHE [31]. Une application à un code à source ouverte et collaboratif est illustrée<sup>6</sup>.

---

5. Voire de le rendre agréable.

6. Pour un autre point de vue sur la question, avec un regard plus orienté Python, voir [82].

FIGURE 17 – *Relevant xkcd for the compilation time* [128]

#### 4.2.1 Réduction du temps de Compilation : Ninja

##### 4.2.1.1 *Le premier d'entre tous : Make*

Il n'est pas malhonnête de dire que l'outil le plus utilisé pour la gestion de la création des binaires de façon automatique est l'utilitaire `Make`<sup>7</sup>. Il date de 1976 et on le doit au fameux Stuart FELDMAN ; il est maintenant distribué de façon standard dans de nombreuses distributions. Son développement est toujours actif, ainsi la version la plus récente, à l'heure où j'écris ce manuscrit, est actuellement la version 4.1 datant de novembre 2014.

`Make` est principalement utilisé durant le processus de compilation. Son rôle est d'automatiser la création de fichiers dépendant eux-mêmes d'autres fichiers qui sont générés à partir de règles simples. Pour chaque fichier que l'on souhaite créer (ce fichier est un nœud de l'arbre de dépendances et est appelé *cible*, *target*, dans le jargon de `Make`) on définit les fichiers dont ce nœud a besoin (ses fils) et les commandes utilisant ces fils afin de créer le nœud suscité. Puis on fait de même pour chaque fils. Ainsi dans le `CODE 8`, afin de créer le binaire `hello_world`, il faut d'abord créer son fils `hello_world.o`. Ce fils a lui-même besoin de `hello_world.f90` qui est présent sur disque. Ce fichier de configuration est appelé *makefile*.

Pour chaque cible, il est ainsi possible de créer récursivement toute sa généalogie ; c'est-à-dire tous les nœuds intermédiaires dont la cible a besoin et la séquence permettant de les fabriquer. Cet ordre d'exécution est calculé automatiquement par `Make`. L'utilisateur n'a pas à le définir manuellement. La gestion de la cohérence de l'arbre de dépendances est automatique et repose sur les dates de dernière modification des fichiers : si un fils d'un nœud est plus récent que le nœud lui-même, le nœud est invalidé et doit être reconstruit.

---

7. Dans sa version GNU.

On gagne donc en simplification d'utilisation – le développeur ne doit indiquer que les fils du nœud et rien d'autre –, en efficacité – `Make` peut construire automatiquement les fils en parallèle – et en fiabilité – si le *makefile* est bien construit alors on est sûr que les nœuds seront toujours construits et mis à jour – ce que l'on perd en contrôle.

```
target: children
#This space needs to be a <TAB>
    command
```

CODE 7 – General syntax of a simple Makefile

```
hello_world.o: hello_world.f90
    gfortran -c hello_world.f90 -o hello_world.o

hello_world: hello_world.o
    gfortran hello_world.o -o hello_world
```

CODE 8 – Syntax of a simple Makefile

```
O_FILES := $(patsubst %.c,%.o,$(notdir $(C_FILES)))
vpath %.c $(sort $(dir $(C_FILES)))
$(LIB): $(O_FILES)
```

CODE 9 – Syntax of a real Makefile. Courtesy of [41]

CRÉATION DE LA LISTE DES FILS. `Make` a au départ une syntaxe assez simple (voir CODE 7). Au cours du temps, `Make` a évolué pour satisfaire de plus en plus d'utilisateurs et de nombreuses fonctionnalités se sont greffées les unes sur les autres<sup>8</sup>. Par exemple, il existe un grand nombre d'options permettant de créer la liste des fils : on peut utiliser des wildcards<sup>9</sup> – permettant de créer des listes généralistes –, ou bien encore des filtres – permettant des opérations sur ces listes –, etc. Ce qui à terme donne des règles telles que celles montrées dans le CODE 9.

**Inconvénients.** Bien que toutes ces options soient agréables sur papier, leur utilisation relève généralement plus du tâtonnement que d'une utilisation éclairée. En effet, les règles deviennent suffisamment complexes pour que leur clarté s'estompe ; rendant leurs modifications délicates. De plus, parcourir cet arbre de dépendances de nœud en nœud en suivant des règles écrites dans un langage pouvant devenir sibyllin

8. Le manuel fait maintenant 500 pages !

9. Le fameux `*`.

alourdit considérablement le temps de débogage pour l'humain. Enfin, cette intelligence du langage augmente son temps d'exécution <sup>10</sup>.

*Make* n'est pas un mauvais outil <sup>11</sup>, mais il a les défauts de ses qualités : son grand nombre d'options rend possible la création de patterns totalement contre-productifs au niveau de l'efficacité <sup>12</sup>, et la création de lignes totalement cryptiques dignes des plus belles expressions régulières rend très difficile sa maintenance.

De nouveaux outils ont été développés afin de résoudre les problèmes suscités ; deux approches antagonistes ont été suivies :

- Développer des outils de haut niveau avec une syntaxe plus « cohérente » que *Make*, ceci afin de rendre l'écriture des fichiers de configuration plus facile (on peut citer l'utilitaire *Generate Your Project* [34] par exemple) ;
- Ou à l'inverse des utilitaires de bas niveau, délaissant une certaine flexibilité afin d'accélérer la création de l'arbre de dépendances et son exploration.

Notre groupe a choisi la rapidité d'exécution. Il nous a semblé plus logique d'utiliser un utilitaire de bas niveau, même si la création du fichier de configuration peut être un moins « amusante », afin de s'assurer des plus petits temps de compilation. Pour cela, nous avons utilisé l'utilitaire *Ninja*.

#### 4.2.1.2 *Ninja*

```
rule compile
  command = gfortran -c $in -o $out

rule link
  command = gfortran $in -o $out

build hello_world.o: compile hello_world.f90
build hello_world: link hello_world.o
```

CODE 10 – Syntax of a simple *build.ninja* file.

*Ninja* [90] a donc le même objectif que *Make*, mais il a le parti pris d'être le plus rapide possible. Il a été développé dans le cadre du projet de navigateur web *open source* Chromium. L'histoire veut que le développeur ait commencé à développer cet outil – après beaucoup d'énervement et de frustration – lors des 90 secondes nécessaires à *Make* pour

10. Il peut être intéressant pour le lecteur de lancer *Make* avec l'option '-d' afin de voir tous les tests qui sont effectués afin de calculer l'arbre de dépendances.

11. Pour un avis un peu plus tranché sur la question, voir [41].

12. Comme l'utilisation récursive de *Make* par exemple [94].

lui signaler qu'il avait oublié un point virgule lors de sa dernière modification du code source. Bien que n'oubliant jamais de point virgule, ceci était aussi mon ressenti lors de mon travail dans le *Quantum Package* : une création puis résolution de l'arbre de dépendances après modification d'un seul fichier prenant onze secondes environ ; ceci causé principalement par la recompilation de fichiers inutiles du fait de bugs dans les *makefiles*.

Ce souci de *Ninja* pour la performance a pour corollaire principal une réduction drastique des fonctionnalités<sup>13</sup>. En effet la rapidité de *Ninja* tient dans sa contrainte concernant la liste des dépendances : elle doit être écrite *explicitement*. Il n'existe pas de *wildcard*, ou toute autre technique afin d'automatiser la création de la liste des fils. Tous les fichiers doivent être écrits en dur. Ceci permet à *Ninja* de ne faire aucun compromis sur la vitesse d'exécution.

**Avantages.** L'un des défauts de *Ninja* est aussi son plus gros avantage : il est très rébarbatif d'écrire à la main ces fichiers de configuration. Ainsi, il est recommandé de générer ces fichiers en utilisant des scripts. Le choix du langage utilisé pour cette génération est quant à lui libre<sup>14</sup>. Ceci permet à la fois une facilité de développement – du fait de la familiarité du langage choisie pour générer le fichier *Ninja* – et une flexibilité accrue par rapport à *Make* ; il est par exemple possible de générer différents fichiers pour différentes utilisations. Ainsi pour le *Quantum Package*, un mode « développeur » est présent permettant de compiler chaque binaire indépendamment, ainsi qu'un mode « production » ne permettant de compiler que l'ensemble des binaires, mais en un temps record. De plus, du fait des dépendances totalement explicites il est beaucoup plus facile de débbugger le processus de compilation même si le fichier de configuration est relativement plus imposant qu'un *makefile*. Pour cela, il suffit de suivre à la main la création d'un fichier, ou encore de regarder l'image de l'arbre de dépendances que *Ninja* peut créer de façon automatique<sup>15</sup>. On peut littéralement suivre un fichier, tandis que dans le cas *Make* c'est beaucoup plus compliqué du fait des *wildcards* par exemple. Enfin, ce type de fichier de configuration est plus rigide et permet à *Ninja* d'effectuer des tests de validité *a priori*, ceci laissant moins de place pour l'apparition de bugs.

**CAS D'UTILISATION.** Le *Quantum Package* se compose de différents modules ayant des dépendances entre eux et où l'on doit d'abord passer l'utilitaire *IRPF90* pour générer les fichiers *Fortran* adéquats avant de pouvoir les compiler. Le script de génération du fichier de configuration

13. Soit une illustration du principe cher à Kelly JOHNSON KISS : *Keep It Simple Stupid*.

14. Dans le cas du *Quantum Package*, le langage *Python* est utilisé.

15. Via la commande `$ ninja -t graph mytarget | dot -Tpng -o graph.png`.

	<i>ex nihilo</i>	No modif.	One modif.
Make	313	5	11
Ninja	148	0	4
Ninja cached	30	0	4

TABLE 12 – Ninja vs Make benchmark. Time in seconds.

de Ninja fait un petit millier de lignes et le fichier généré quelque milliers de lignes. C’est au cours de l’écriture de ce script et du volume qu’il représentait que j’ai réalisé que notre *Makefile* était rempli de bugs et qu’il est vraiment difficile d’écrire un *Makefile* généraliste *correct* prenant en compte toutes les possibilités.

Le but principal étant quand même la rapidité, il est observé un gain de temps substantiel moyen d’un facteur deux (Tableau 12) pour le même travail à effectuer.

CONCLUSION. S’il faut retenir une chose de cette partie, et cela dépasse l’utilisation de Ninja, c’est que pour la création de fichiers de type *makefile* il est préférable d’explicitier toutes les dépendances et donc de créer un script permettant de générer ces dépendances plutôt que d’utiliser des listes implicites. Ceci accélérera l’exécution de la compilation, simplifiera la maintenance et permettra de plus facilement corriger les bugs. Le temps passé à la création du script de génération sera largement compensé.

#### 4.2.1.3 Cachons la compilation

Il arrive assez régulièrement d’avoir à recompiler des fichiers à l’identique. Ce cas de figure arrive typiquement après un `$ make clean`<sup>16</sup> ou bien encore lorsque l’on souhaite revenir à la version optimisée du code après avoir changé les options de compilation pour faciliter le débogage. Nous avons développé un petit script en Python ( $\approx 100$  lignes [9]) sans grande prétention, qui permet de mettre en cache les fichiers déjà compilés et ainsi éviter toute recompilation superflue. En effet, si ni le fichier ni les options de compilation n’ont changé et que la compilation a déjà été effectuée, il n’est pas nécessaire de la refaire ; on peut simplement récupérer le résultat de la précédente compilation. Cette idée est somme toute assez naturelle, ainsi le projet Chromium utilise le même genre de processus.

INFORMATIQUE. Ce script prend en argument une commande de compilation. Par exemple :

16. Ou maintenant un `$ ninja -t clean`.

```
ifort -g -openmp -c kin_ao_ints.irp.module.F90 \
      -o kin_ao_ints.irp.module.o \
      -I ezfio_files
```

Et il en calcule une *clé MD5* correspondant à la paire :

(code source du fichier, options de compilation)

et associe cette paire au fichier résultant de la compilation. Si la clé existe, le script retourne le fichier compilé qui a été précédemment sauvegardé, sinon il compile le fichier en exécutant la commande fournie en argument et sauvegarde le fichier<sup>17</sup>. Ce petit script peut être adapté assez facilement à la plupart des langages compilés. Il suffit pour cela de changer les expressions régulières permettant de capturer le nom des fichiers d'entrée et de sortie ainsi que les options de compilation.

Ce script nous a permis de réduire d'un facteur trois le temps d'exécution total de l'installation de notre code dans le cas où tous les fichiers ont déjà été mis en cache (Tableau 12). Le temps d'exécution n'est du qu'au passage de l'outil *IRPF90* – que nous allons présenter dans la prochaine partie – et à la compilation des modules qui ne peuvent être mis en cache dans le cas du *Fortran*.

#### 4.2.2 Réduction de la complexité du cœur calculatoire : IRP

Nous arrivons ici dans l'une des sections m'ayant posé le plus de difficultés lors de la rédaction. En effet ce point me paraît si essentiel et la solution si élégante que j'aimerais vraiment que tout ce qui suit soit limpide. Nous parlerons ici de *IRP* et plus précisément de *IRPF90*, l'implémentation effectuée par Anthony SCEMAMA pour le *Fortran90*. Dans une première partie, j'utiliserai une analogie avec *Make* ; puis dans la conclusion je ferai une comparaison avec le paradigme de la programmation orienté objet.

Pour la compilation de binaires, il ne viendrait à l'esprit de personne de se passer d'utilitaires comme *Make* ou *Ninja*. L'arbre de dépendances de tous ces binaires est long, complexe, et apporte peu d'information au développeur. Peu de personnes se demandent paniquées lors de la compilation d'un programme : « Mon dieu ! Ce fichier source a-t-il été compilé avant ou après celui-là ? ! Je déteste perdre le contrôle et ne pas savoir ! ». En effet, même si nous n'avons pas le contrôle sur l'ordre d'exécution de la compilation nous sommes assurés d'avoir le résultat final, et mieux, le plus rapidement possible. De plus, l'écriture du *makefile* est assez aisée pour le développeur. Il n'a pas à gérer tout l'arbre de dépen-

17. Il est à noter que dans le cas particulier du *Fortran*, les fichiers de module *.mod* sont problématiques du fait qu'il n'existe aucun standard concernant les noms de ces fichiers. De ce fait, si le fichier d'entrée produit un module notre script ne fait qu'exécuter la commande passée en arguments.



*"Instead of telling the machine what to do, we can express what we want"*

FIGURE 18 – Zen of IRP by Anthony SCEMAMA

dances, il a juste besoin de connaître la liste de tous les fils du nœud à compiler et la règle permettant de produire le fichier correspondant au nœud à partir de ses fils.

IRPF90 n'est rien d'autre que la généralisation de `Make` pour certaines variables d'un programme. En effet, on peut considérer un programme scientifique comme le calcul d'une succession de variables intermédiaires afin d'arriver à la création d'une variable terminale ; qui dans un code de chimie quantique est par exemple une énergie. Il est possible de décomposer ces variables en deux grandes catégories :

- Celles que l'on peut considérer comme temporaires. Il s'agit de variables locales qui ont une courte durée de vie, comme par exemple les compteurs de boucles ou des fragments de calculs.
- Les variables nodales. Nodales, car elles sont présentes dans l'arbre de dépendances et il s'agit donc de variables que l'on doit sauvegarder. Ce sont des résultats qui pourront être utilisés par d'autres variables nodales, et ce sont ces variables qui vont être gérées automatiquement par IRPF90.

Une des difficultés dans les paradigmes actuels de programmation est la création puis la récupération de ces variables nodales. En effet, si une variable nodale a déjà été construite alors il ne faut *pas* la reconstruire. Le développeur doit donc d'une part connaître *tout* l'arbre de dépendances du code ; et passer, par cascade, cette variable dans des fonctions afin de la faire migrer dans différentes parties du code. C'est là qu'intervient le modèle IRP, il crée automatiquement l'arbre de dépendances des ces variables nodales. Ainsi, le développeur peut utiliser ces variables nodales n'importe où et n'importe quand dans le code. L'IRP s'assurera que les variables qui ont déjà été construites ne seront pas recalculées, et calculera les variables non encore construites en suivant l'exploration de l'arbre de dépendances.

#### 4.2.2.1 Informatique.

Notons que l'IRPF90 est une extension du `Fortran90`, donc tout programme `Fortran90` est un programme IRPF90 valide.

Une variable nodale est caractérisée par le fait qu'il n'existe qu'une et qu'une seule manière de la construire. La routine qui de construit la variable nodale n'a jamais d'argument, et elle est appelée `Provider` car elle *fournit* la valeur de la variable considérée. Cette routine est décrite dans IRPF90 par un bloc `BEGIN_PROVIDER [...] END_PROVIDER` contenant du code `Fortran` dont le rôle est de construire une valeur *valide* de

la variable, Le `Provider` est automatiquement appelé avant que la variable soit utilisée.

D'un point de vue informatique, les variables nodales sont des variables globales non-mutables évaluées de façon paresseuse (*lazy evaluation*). « Globales », car une fois définies elles sont utilisables dans tout le code ; « non-mutables », car elles ne peuvent être changées (sauf dans certains cas particuliers) ; « évaluées de façon paresseuse », car elles ne sont calculées que si elles sont présentes dans l'arbre de dépendances. Le fait que les `providers` sont globaux est très pratique pour le développeur. Par exemple, si l'on veut la matrice de recouvrement des orbitales atomiques, il suffit de chercher dans la documentation le nom de la variable nodale correspondante, puis de l'utiliser comme si elle était déjà calculée. C'est tout ; il n'y a rien d'autre à faire.

**SYNTAXE.** La principale modification qu'il faut effectuer pour bénéficier de la gestion automatique de l'arbre de dépendances est l'écriture des `providers` (voir Figure 19 pour un exemple) dans des fichiers `irp.f`.

**ALGORITHMIQUE.** L'écriture automatique de code `Fortran` à partir de fichiers `IRPF90` est somme toute assez simple. L'algorithme est le suivant<sup>18</sup> :

1. Pour chaque fichier `irp.f` est généré un module contenant :
  - Les variables contenant les valeurs des variables nodales
  - Un booléen pour chaque variable nodale permettant savoir si la valeur a déjà été calculée
2. Puis pour chaque variable, le `Provider` est écrit.
  - Il vérifie si la variable a déjà été calculée
  - Si ce n'est pas le cas, il appelle les `providers` de tous ses fils, puis il calcule la valeur, et enfin il marque la variable nodale comme *construite*
3. Puis des appels sont insérés dans toutes les *subroutines*, fonctions, `providers` et programmes afin que les `providers` soient appelés avant l'utilisation des variables correspondantes.

---

<sup>18</sup>. Pour une description détaillée des étapes 1 et 2 voir CODE 11.

$$t(u(d1, d2), v(u(d3, d4), w(d5)))$$

$$\text{where } \begin{cases} t(x, y) = x + y + 4 \\ u(x, y) = x + y + 1 \\ v(x, y) = x + y + 2 \\ w(x) = x + 3 \end{cases}$$

(a) Mathematical representation

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

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

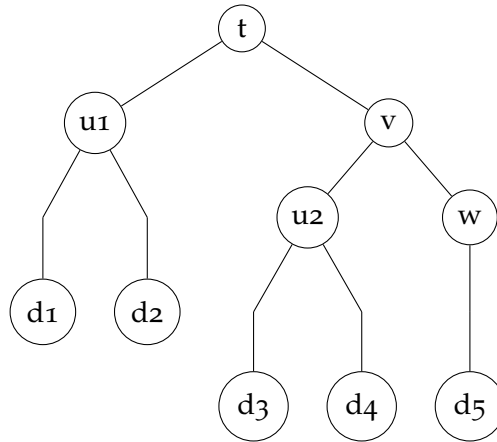
BEGIN_PROVIDER [ integer, v ]
  v = u2+w+2
END_PROVIDER

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

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

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

(c) Nodal variables (uvwt.irp.f)



(b) Graph dependency

```
BEGIN_PROVIDER [ integer, d1 ]
&BEGIN_PROVIDER [ integer, d2 ]
&BEGIN_PROVIDER [ integer, d3 ]
&BEGIN_PROVIDER [ integer, d4 ]
&BEGIN_PROVIDER [ integer, d5 ]

  print *, '>d1, d2, d3, d4, d5?'
  read(*,*) d1, d2, d3, d4, d5

END_PROVIDER
```

(d) Input variables (input.irp.f)

```
program irp_example
  implicit none
  print *, '>t = ', t
end
```

(e) Main (irp\_exemple.irp.f)

```
$ ls
input.irp.f  irp_example.irp.f  uvwt.irp.f

$ irpf90 --init
$ ls
input.irp.f  irp_example.irp.f  IRPF90_man  IRPF90_temp
Makefile uvwt.irp.f

$ make >> /dev/null
$ ./irp_exemple
>d1, d2, d3, d4, d5?
1 5 8 10 7
>t = 42
```

(f) Trace of execution

FIGURE 19 – IRPF90 in action

```

module {{file.name}}_mod
  {% for value in file.l_provider %}
  {{value.type}} :: {{value.name}}
  logical :: {{value.name}}_is_built = .FALSE.
  {% endfor %}
end module {{file.name}}_mod

{% for value in file.l_provider %}
subroutine provide_{{value.name}}

  {% for mod in value.l_mod %}
  use {{mod.name}}_mod
  {% endfor %}

  if (.not.{{value.name}}_is_built) then

    {% for parent in value.l_parents %}
    call provide_{{parent.name}}
    {% endfor %}

    call bld_{{value.name}}
    {{value.name}}_is_built = .TRUE.
  end if
end subroutine provide_{{value.name}}

subroutine bld_{{value.name}}

  {% for mod in value.l_mod %}
  use {{mod.name}}_mod
  {% endfor %}

  {{value.code}}
end subroutine bld_{{value.name}}
{% endfor %}

```

CODE 11 – Generated Fortran code by IRPF90

#### 4.2.2.2 Conclusion

Pour moderniser un vieux code, la tendance est d'aller vers la programmation orientée objet. Pourtant ce paradigme ne me paraît pas optimum pour le *High Performance Computing* (HPC). En effet, les code académiques sont souvent très changeants, rendant la modélisation d'objets assez difficile. De plus, les structures de données associées aux objets ne

favorisent pas l'utilisation optimale de l'accès aux données <sup>19</sup>, ce qui est un frein énorme à l'efficacité. La programmation par IRP semble soigner ces deux défauts. S'il l'on veut utiliser une nouvelle variable, il suffit de créer un nouveau `Provider` ; il n'y a pas de grande question darwiniste sur la dépendance ou non de cette variable d'une classe abstraite. L'IRP semble bien adapté au développement toujours changeant, tandis que l'orienté objet me semble utile principalement pour des codes dont on connaît *a priori* les tenants et les aboutissants. Il est important de noter que l'IRP n'est pas incompatible avec l'orienté objet : on peut créer des objets contenant des `providers` et *vice versa*.

De part sa facilité d'utilisation, les services qu'il rend, et son interopérabilité, j'espère vraiment que l'IRP deviendra un jour populaire. Je pense qu'un portage en C/C++, sera nécessaire pour améliorer sa diffusion. Peut-être un futur projet ...

#### 4.2.3 Système de contrôle de versions : Exemple de Git

Si l'on devait peindre le diptyque des grandes peurs de tout développeur, il y aurait d'un côté le fait de se rendre compte – après un laps de temps toujours trop grand – que son nouveau code ne donne plus les bons résultats ; et de l'autre côté, de ne pouvoir revenir en arrière du fait de l'absence de sauvegarde pertinente.

Ainsi, un système de contrôle de versions a pour but principal d'éviter les cruels choix des noms des fichiers de sauvegarde tels que dans un ordre chronologique :

- `*_New` ;
- `*_New_New` ;
- `*_Old_but_Newer_than_New` ;

Et aussi de rendre ces sauvegardes systématiques. On définit pour cela des dépôts (*repository* dans la langue de M. TURING) contenant un ensemble de fichiers. Git permet de garder ni plus ni moins que l'historique complet des modifications effectuées dans ce dépôt. Il est ainsi possible de naviguer dans cet historique afin de comparer des versions, de rétablir une version antérieure, de restaurer des fichiers supprimés par mégarde, et toute autre chose utile. Laconiquement, rien n'est perdu.

---

19. "You wanted a banana but what you got was a gorilla holding the banana and the entire jungle." Joe ARMSTRONG, créateur du langage ERLANG dans [112].

## 4.2.3.1 Introduction succincte

Voici très rapidement une introduction pratique à Git<sup>20 21</sup>.

- Un dépôt s’initialise avec `$ git --init`. Puis, les fichiers à surveiller s’ajoutent avec la commande `$ git add <file>` ;
- Quand on veut sauvegarder l’ensemble des modifications effectuées il faut créer un *commit* (`$ git commit -a -m "message"`). Il s’agit d’un instantané de tous les fichiers suivis qui sera stocké dans la base de données interne de Git ;
- En cas de coup dur `$ git reset --hard` permet de restaurer l’ensemble du code à l’état du dernier *commit* ; `$ git checkout --force <file>` permet de restaurer un fichier spécifique.
- `$ git diff` permet de voir les différences entre les fichiers actuels et ceux du dernier *commit*, et `$ git log` permet de voir l’histoire de tous les *commits*.
- Souvent il ne s’agit pas de repartir de zéro, mais d’utiliser un dépôt préexistant. `$ git clone <path>` permet de copier le dépôt présent dans le chemin (ce dépôt est maintenant appelé dépôt maître ou *master*) dans le répertoire local.
- Grossièrement, il y a deux grands types d’interactions possibles entre deux dépôts :
  - Le maître a fait des modifications et vous voulez les récupérer ? `$ git pull`.
  - Vous avez développé une superbe nouvelle routine et vous voulez le communiquer au maître ? `$ git push`. Il vous faut bien évidemment avoir les droits sur le dépôt maître.
- Le terme *merge* est utilisé lorsque l’on fusionne deux versions (à l’aide d’un *pull* par exemple). On parle aussi d’intégration. C’est ici que certains problèmes se posent. Si deux versions ont divergé dans des régions qui se recouvrent, leur *merge* ne peut être automatisé. C’est donc à l’utilisateur d’effectuer la résolution des conflits.

Pour finir, un des avantages de Git sur la concurrence est le fait que chaque dépôt contient l’ensemble de l’historique de toutes les modifications. Autrement dit, lors d’un `git clone` toute l’histoire du dépôt est disponible, et non pas une *tabula rasa*. Ainsi, chaque utilisateur possède une sauvegarde locale de tout le dépôt.

20. Un didacticiel à l’usage des non-informaticiens, voir aussi [33]. Pour un exemple de *workflow* on peut se référer à [123].

21. Pour les gourous souhaitant avoir une compréhension profonde de Git (tout ce que vous avez toujours voulu savoir sur les *Blobs*, *Tree object*, *Commit object* et leur gestion) je ne saurais que conseiller la référence [127].

#### 4.2.4 Outils d'aide pour la gestion de projet collaboratif

Nous avons vu comment utiliser **Git** pour faire les modifications sur des dépôts qui vous appartiennent. Maintenant nous allons voir, comment utiliser **Git** de manière collaborative. Avant cela, il est nécessaire de définir deux termes :

- Les Forks <sup>22</sup> sont des copies personnelles d'un dépôt tiers.
- Dans le jargon utilisé par les développeurs utilisant **Git** une expression revient régulièrement, celle de *demande de pull* (*pull request*). Il s'agit de demander à un dépôt de bien vouloir se synchroniser avec vous afin de récupérer les modifications que vous avez faites. Cela permet de synchroniser deux dépôts qui n'ont pas le même propriétaire. Philosophiquement, il ne s'agit pas de *push request* (« je vous demande d'accepter mes modifications »), mais bien de *pull request* (« je vous demande de venir prendre mes modifications »). Ainsi l'ordre des demandes de pull est surprenant : Dépôt ancien  $\Rightarrow$  Dépôt neuf quand le dépôt ancien souhaite se mettre à jour avec le Dépôt neuf.

Lorsque l'on en vient à utiliser **Git** de manière massivement collaborative quelques écueils peuvent apparaître. Il faut pouvoir transmettre son dépôt facilement aux collaborateurs, typiquement à l'aide d'une page web. De plus ces collaborateurs peuvent avoir différentes utilisations du code : certains veulent modifier le code pour leur propre utilisation, d'autres veulent faire partager leurs modifications et les intégrer à la version principale.

Nous allons montrer dans la suite de ce chapitre, une méthode que j'espère simple permettant de simplifier ce processus. Les deux idées fondamentales qui guideront nos pas sont les suivantes :

- Ce *workflow* doit être le plus simple, flexible et systématique possible. En effet, dans notre milieu de la recherche académique, le cursus informatique des développeurs étant pour le moins hétérogène, je conviens très bien que ce genre de considération et de limitation de leur liberté soit assez énervante ;
- Il faut s'assurer que le code disponible au téléchargement reste stable à tout moment du processus de développement.

##### 4.2.4.1 Github

Afin de partager les codes, le site web **Github** est né. Certains l'appellent le réseau social des codes *open source*. Il permet d'héberger gratuitement un dépôt sur internet. Il propose une navigation agréable à travers le code source, permet la création rapide d'une page web, d'un

---

22. Je m'abstiendrai d'utiliser « Fourche » comme traduction.

wiki. Il permet de gérer les pull requests facilement et permet aussi aux utilisateurs de faire remonter des bugs ou des demandes d'améliorations. Enfin, il facilite grandement la création de *forks* et de *clones*. Il est à noter que du fait que `Git` soit décentralisé, si le site `Github` vient à fermer vous ne perdrez aucune de vos données. `Github`, n'est qu'une interface ; certes une interface pratique, mais rien d'autre qu'une interface.

Notre *workflow* est le suivant. L'organisation `LCPQ` possède le dépôt contenant la version stable du `Quantum Package`, il est considéré comme le dépôt référent. Chaque collaborateur en a un *fork*, qui est sa version de travail. Les synchronisations entre les différentes copies du code se font à l'aide de *pull requests*. Seuls les *core developers* peuvent accepter les modifications proposées au dépôt `LCPQ` car les modification pourraient impacter la stabilité de la version principale.

La question qui se pose maintenant est de savoir vérifier la stabilité de notre code. Les versions des collaborateurs peuvent être instables<sup>23</sup> mais il est impensable que cela arrive à la version référente. Nous allons donc voir dans une première partie, comment vérifier cette stabilité, et dans un deuxième temps comment intégrer cette vérification dans notre *workflow*.

#### 4.2.4.2 Tests de non-régression

Il nous faut donc définir des critères de stabilité que chaque version du code destinée à arriver dans le dépôt principal doit vérifier<sup>24</sup>. Les nouvelles versions ne doivent pas donner des résultats différents de l'ancienne version<sup>25</sup>.

Dans le `Quantum Package` nous vérifions plusieurs choses :

- La compilation du code, c'est en effet une condition nécessaire ;
- La cohérence des fichiers d'entrée. Le code doit refuser de se lancer si les paramètres d'entrée sont incohérents. Par exemple, un nombre non-entier de déterminants dans un calcul d'*Interaction de Configuration (IC)* ;
- La validation des résultats obtenus. Pour un jeu de paramètres donnés, des valeurs de référence doivent être produites.

Informatiquement, nous utilisons la bibliothèque `Python unittest` [118] pour gérer les tests de non-régression<sup>26</sup>. Elle permet de vérifier aisément certaines conditions telles que :

le test doit retourner vrai (au sens booléen du terme)

23. C'est-à-dire ne pas compiler ou donner de mauvais résultats.

24. Je dois ma première rencontre avec ce genre de batterie de tests au code `ABINIT` [70].

25. Sauf correction de bug bien évidemment.

26. Ceux validant les fichiers d'entrée et de sortie



la valeur de sortie d'un test doit être équivalente à une valeur de référence à une précision donnée  
le test doit produire une erreur<sup>27</sup>.

Par exemple dans le **CODE 12**, on vérifie que les fonctions `check_disk_access` – testant les paramètres concernant la lecture et/ou l'écriture des fichiers d'intégrales – et `check_mo_guess` – testant le type d'orbitale moléculaire demandée par l'utilisateur –, retournent « vrai ».

```
class InputTest(unittest.TestCase):

    def test_check_disk_access(self):
        self.assertTrue(check_disk_access("methane",
                                           "un-ccemd-ref"))

    def test_check_mo_guess(self):
        self.assertTrue(check_mo_guess("methane",
                                        "maug-cc-pVDZ"))
```

CODE 12 – UnitTest implementation

Ces tests doivent être systématiques, et leurs résultats doivent être visibles à la fois par l'émetteur et le destinataire du pull request pour plus de transparence. De plus, ces tests doivent être effectués dans un environnement vierge, afin de s'assurer de leur fiabilité. Peut-être que ce code marche sur cette machine, avec cette version de Python, et ce compilateur particulier ; mais sur une autre machine, comment en être sûr ? On doit donc s'assurer de l'universalité de la réussite des tests.

#### 4.2.4.3 *Continuous Integration*

C'est ici que Travis CI intervient. Il met gratuitement à disposition de la communauté *open source* des machines virtuelles. Ces machines virtuelles lancent les commandes présentes dans le fichier `.travis.yml` à chaque push et pull request effectués sur le dépôt Git via Github. Il notifie ensuite l'utilisateur de la réussite ou non de ces commandes.

À chaque déclenchement de Travis CI, tous les tests sont effectués. Ainsi le développeur sait le plus tôt possible si sa modification a eu des effets de bord non prévus. Le cas échéant, il peut la corriger dès que faire se peut. Une fois que sa version passe les tests, il peut faire le pull request vers le dépôt référent.

Le CI de Travis CI est l'acronyme de *Continuous Integration*. Cette philosophie de développement essaie d'éviter un écueil du développement collaboratif : l'intégration/la fusion/la synchronisation, de deux

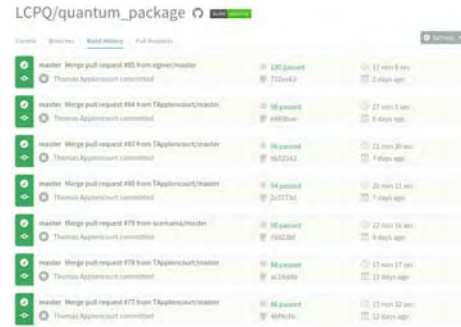
27. Un segment `raise exception` en Python.



(a) Personal fork with lot of errors



(b) The tests pass, you can accept the pull request



(c) In the main repository, by construction all the tests are OK

FIGURE 20 – Merge workflow

versions qui ont beaucoup divergé<sup>28</sup>. Pour cela, il convient de fusionner les différentes versions le plus souvent possibles. C'est-à-dire, si l'on utilise Git, d'effectuer des pull requests le plus fréquemment possible. Par construction, cela minimise les divergences entre les nombreuses versions du code en liberté. À chaque changement du dépôt maître, chaque collaborateur est vivement encouragé à se mettre à jour par rapport à cette version.

#### 4.2.4.4 Où l'on combine tous ces outils

De cela, nous pouvons déduire un protocole garantissant la stabilité d'un code de la façon la plus automatique possible :

1. Un dépôt principal contient la version stable (*LCPQ master*) hébergé sur Github ;
2. Chaque collaborateur en a un duplicata qu'il peut modifier à convenance (*Fork*) ;
3. Quand les collaborateurs veulent faire partager leurs modifications et donc modifier le dépôt maître, ils jouent la partition suivante :
  - a) Premièrement, un *pull request* du *LCPQ master* vers leur *Fork* est effectué. Ceci a pour but de rapatrier dans leur *Fork* les

<sup>28</sup>. Il est laissé au lecteur d'imaginer ce calvaire.

changements ayant potentiellement eu lieu dans le dépôt de référence (*LCPQ master*)

- b) Deuxièmement les conflits possiblement créés par ce *merge* sont résolus.
  - c) Finalement un *pull request* du *Fork* vers le *LCPQ* est lancé. Il permettra de rajouter à la version stable les nouvelles fonctionnalités présentes dans le *Fork*. Cette demande est interceptée par *Travis CI* qui lance les commandes idoines d'installation et de tests de non-régression (*Figure 20b*). Si aucune erreur n'est détectée lors de l'exécution par la machine virtuelle, la demande de *pull* peut être acceptée (du fait des étapes précédentes, cette intégration devrait se faire sans aucun heurt).
4. Par construction, le dépôt maître reste stable tout en étant mis à jour (*Figure 20c*).

Cette façon de faire est très souple. En effet, chaque développeur, ou groupe de développeurs, peut faire ce qu'il souhaite dans son *Fork*. Il n'a pas à changer ses habitudes. La seule chose à faire est d'accepter les *pull requests* si et seulement si les batteries de tests ne renvoient pas d'erreur.

#### 4.2.5 Conclusion

Dans un premier temps nous avons vu les outils permettant la réduction du temps de compilation. L'idée principale est d'écrire des fichiers de dépendances explicites et de générer ces fichiers à l'aide d'un script externe. Puis, en suivant l'analogie du *Makefile* nous avons présenté *IRPF90* qui simplifie grandement le développement en *Fortran90* en permettant au développeur de ne pas avoir à retenir l'ensemble de l'arbre de dépendances des variables qu'il utilise. Enfin, nous avons présenté des outils facilitant le développement collaboratif.

## 4.3 SIMPLIFICATION POUR L'UTILISATEUR

### 4.3.1 Tous les fichiers de bases atomiques à disposition

**HISTORIQUE.** Les calculs de chimie quantique reposent sur l'utilisation de fonctions de base sur lesquelles sont projetés les objets mathématiques considérés<sup>29</sup>. Il n'existe pas de jeu universel de fonctions de base<sup>30</sup>. De plus le format utilisé par ces bases de données ne fait pas consensus, chaque code utilise son propre format. Ainsi il fut assez tôt

29. Hamiltonien, fonction d'onde, déterminant, *etc.*

30. Plus de 500 ensembles de fonctions de base est recensé.

envisagé [56] de regrouper toutes ces informations dans une base de données commune. L'*Environment Molecular Sciences Laboratory, Pacific Northwest National Laboratory, US Department of Energy* (EMSL) créa ainsi un site web permettant un accès aisé à tout cet ensemble de fonctions de base ; ce qui a rendu et rend encore, un formidable service à notre communauté.

Traditionnellement les codes de calcul ont un certain nombre d'ensembles de fonctions de base présentes en interne. L'utilisateur souhaitant utiliser une autre base atomique présente sur le site de l'EMSL peut la rajouter « à la main » dans le fichier d'entrée du programme. Pour cela, il se connecte sur le site web de l'EMSL, choisit la base atomique, puis copie/colle les paramètres dans le fichier d'entrée de son programme. Malheureusement, cette façon de procéder ne peut pas facilement être automatisée rendant le test de nombreux ensembles de fonctions de base (*benchmark*) assez mal aisé.

Ainsi j'ai décidé de demander aux chercheurs de l'EMSL la permission de pouvoir copier leurs données localement<sup>31</sup>. On peut maintenant disposer de l'ensemble de la base de données de l'EMSL directement sur son poste de travail permettant l'utilisation de toutes les fonctions de bases facilement.

#### 4.3.1.1 *Création de la base de données*

En regardant le code source du site web, il fut assez aisé d'isoler les URL permettant de récupérer la liste des noms des fonctions de base, et les paramètres correspondants.

---

31. Ce script a rencontré un succès modeste et a déjà été « forké » par cinq utilisateurs à travers le monde.

```
#!/usr/bin/env python
import requests

url = ('https://bse.pnl.gov:443/bse/portal/user/'
       'anon/js_peid/11535052407933/'
       'action/portlets.BasisSetAction/'
       'template/courier_content/panel/Main/'
       '/eventSubmit_doDownload/true')

params = {'bsurl': ('/files/projects/Basis_Set_Curators/'
                   'Gaussian/emsl-lib/CC-PV8Z.xml'),
          'bsname': 'cc-pV8Z',
          'elts': ' '.join(['H', 'Ne']),
          'format': 'GAMESS-US',
          'minimize': 'True'}

print requests.get(url, params=params).text
```

CODE 13 – Request library. Example of basis set download from the EMSL web site.

RÉCUPÉRATION DES DONNÉES. La bibliothèque `Python requests` [102] permet facilement de s'interfacer avec le protocole HTTP. Synthétiquement, on peut communiquer avec un serveur web grâce à la méthode `GET`. Ainsi une même URL, appelée avec des paramètres différents permet de récupérer des données différentes. Dans notre cas, nous allons passer à l'URL du serveur web de l'EMSL les paramètres identifiant nos fichiers de base atomique<sup>32</sup> afin de pouvoir les récupérer. `requests` rend cette gymnastique transparente. Il suffit de transmettre à `requests` l'URL et un dictionnaire contenant tous les paramètres nécessaires et leurs valeurs.

PARALLÉLISATION. Il convient maintenant de rendre le script parallèle. En effet une requête `GET` doit être effectuée pour chaque ensemble de fonctions de base présent dans la base de données de l'EMSL, et il en existe plusieurs centaines. Ainsi, si l'on effectue ceci de façon séquentielle, il faut de l'ordre d'une vingtaine de minutes.

Nous présenterons rapidement un algorithme général et très simple permettant la parallélisation de tâches indépendantes en `Python` se basant sur l'utilisation de `threads`. Les éléments présentés sont disponibles dans les modules `Threading` et `Queuing` de la bibliothèque standard de `Python`. L'idée est simple, nous allons créer plusieurs `threads`<sup>33</sup> qui s'oc-

32. L'URL de cette base de données, son nom, quels éléments nous souhaitons, le format demandé, etc.

33. Nous les avons définies à la sous-section 3.1.3.

cuperont de télécharger les fichiers de bases atomiques. Ces *threads* prendront donc en entrée les paramètres nécessaires au téléchargement de la base, et en sortie écriront les données récupérées dans une base de données. Mais comment s'assurer la bonne répartition des tâches effectuées entre *threads*? Les tâches seront stockées dans une liste. Chaque *thread* accédera à cette liste globale, récupérera le premier élément de cette liste puis le supprimera, ceci jusqu'à épuisement de la liste. La question de la concomitance des accès mémoire se pose; que se passe-t-il si deux *threads* accèdent au même moment à cette liste? Pour éviter ce genre d'embarras nous allons utiliser une structure de données qui est *thread safe* : les *Queues*. Les queues sont des listes spéciales. D'une part elles ne peuvent être modifiées que par une *thread* à la fois, et deuxièmement elles implémentent des méthodes permettant l'addition et l'extraction rapide d'éléments. Il est à noter aussi que les queues sont bloquantes : si l'on veut extraire une donnée d'une liste vide, le processus attendra que cette queue se remplisse avant de rendre la main.

Ainsi nous aurons une *queue* d'entrée contenant les noms des bases atomiques à télécharger, et une *queue* de sortie contenant les paramètres téléchargés depuis le site web de l'EMSL. La queue de sortie servira de tampon entre les processus parallèles et le processus séquentiel principal. Les *threads* accéderont à la liste d'entrée, effectueront la requête web, et écriront les paramètres de la base de fonctions ainsi téléchargée dans la queue de sortie. Le processus principal s'occupera quant à lui, d'initialiser la queue d'entrée puis de vider la queue de sortie dans la base de données SQLite.

L'implémentation de ce genre de parallélisme ne prend qu'une dizaine de lignes. Ils s'agit donc d'un moyen rapide de paralléliser un petit script sans grande prétention. Au niveau de la performance, en utilisant une vingtaine de *threads*, on peut maintenant télécharger toutes les bases de l'EMSL en quelques minutes seulement.

#### 4.3.1.2 Commande line interface

Un utilitaire en ligne de commandes `EMSL_local.py` [8] a été écrit en Python (voir CODE 14) pour faciliter l'accès à cette base de données et permettre un *scripting* aisé<sup>34</sup>. Ainsi, nous l'avons interfacé avec notre programme principal de calcul Quantum Package. Cela permet, lors de la création de nos fichiers d'entrée d'utiliser toutes les fonctions de base disponibles sur le site de l'EMSL, ce qui est un certain confort. Faire un *benchmark* utilisant plusieurs bases de fonctions est donc devenu très simple.

34. Il utilise le très joli utilitaire *docopt* [49] qui permet de générer automatiquement le *parser* des arguments de la ligne de commande via la documentation.

```

$ ./EMSL_api.py -h
EMSL Api.

Usage:
  EMSL_api.py list_basis [--basis=<basis_name>...]
                        [--atom=<atom_name>...]
                        [--db_path=<db_path>]
                        [--average_mo_number]
  EMSL_api.py list_atoms --basis=<basis_name>
                        [--db_path=<db_path>]
  EMSL_api.py get_basis_data --basis=<basis_name>
                        [--atom=<atom_name>...]
                        [--db_path=<db_path>]
                        [(--save [--path=<path>])]
                        [--check=<program_name>]
                        [--treat_l]

  EMSL_api.py list_formats
  EMSL_api.py create_db --format=<format>
                        [--db_path=<db_path>]
                        [--no-contraction]

  EMSL_api.py (-h | --help)
  EMSL_api.py --version

Options:
  -h --help          Show this screen.
  --version          Show version.
  --no-contraction   Basis functions are not contracted

<db_path> is the path to the SQLite3 file
                containing the Basis sets.
By default is $EMSL_API_ROOT/db/Gaussian_uk.db

Example of use:
  ./EMSL_api.py list_basis --atom Al --atom U
  ./EMSL_api.py list_basis --atom S --basis 'cc-pV*' \
                        --average_mo_number
  ./EMSL_api.py list_atoms --basis ANO-RCC
  ./EMSL_api.py get_basis_data --basis 3-21++G*

```

CODE 14 – EMSL local API

### 4.3.2 Du stockage des données

La prospection scientifique produit un grand nombre de résultats devant être stockés de façon pérenne. De plus leur stockage doit être complété par des meta-données suffisamment explicites pour se remémorer quelques années plus tard à quoi les données correspondent. L'extraction des données doit également être facile pour pouvoir permettre d'effectuer des calculs de post-traitement sur les jeux de données.

Le cas particulier de notre équipe a été l'application du *benchmark* G2, visant à calculer les énergies d'atomisation de 55 molécules. L'énergie d'atomisation est l'énergie nécessaire pour rompre toutes les liaisons chimiques d'une molécule. Il s'agit donc de la différence d'énergie entre l'énergie de la molécule et la somme des énergies des atomes la constituant. La distribution des erreurs par rapport aux valeurs de référence permet de trouver, parmi plusieurs stratégies, celle qui est la plus précise. Ainsi, pour chaque modification du protocole de calcul il est nécessaire de calculer les énergies des 55 molécules, et des différents atomes les constituant.

À l'heure où cette phrase est écrite, nous avons environ 10 000 valeurs d'énergie. Cet ensemble de valeurs correspond à l'ensemble des hypothèses que nous avons testées. Maintenant, nous devons naviguer dans cette océan de données pour récupérer les informations utiles. Vu la quantité de données, l'utilisation de tableurs est illusoire. Heureusement les bases de données relationnelles sont parfaitement adaptées à ce type d'utilisation. Elles sont constituées de tableaux reliés entre eux par des règles d'appartenance. Une syntaxe propre aux bases de données a été créée et permet la récupération aisée de données ne satisfaisant que certaines règles.

Ainsi j'ai créé l'utilitaire Python `G2_API.py` ([6]), utilisant le gestionnaire de base de données SQLite.

Cet utilitaire permet de stocker aussi bien les données de référence que les données issues de différents calculs. Par exemple, les géométries expérimentales de toutes les molécules sont stockées, ainsi que les énergies de référence, la multiplicité de spin, les énergies de point zéro, *etc.* Ainsi, cet utilitaire permet d'une part de construire de façon automatique des fichiers d'entrée pour plusieurs programmes correspondant aux 55 molécules<sup>35</sup>, mais il permet également de stocker les résultats des calculs et de filtrer et post-traiter les résultats (pour un exemple de sortie produite par cet utilitaire, voir la Figure 21.).

Un avantage de cet approche est le côté auto-suffisant et portable. En effet, avec cet utilitaire et le fichier de base de données, n'importe quel collaborateur a accès à toutes les données permettant la reproduction et

35. Ceci est illustré dans l'Appendice B.



```
$ ./schindler.py list_run --method DMC --order_by mad --like_run_id 60
```

Run_id	Method	Basis	Geo	Comments	mad
					kcal/mol
71	DMC	cc-pVDZ	Experiment	1M_Dets_N0_1k_Dets	11.26+/-1.38
114	DMC	vtz-BFD	Experiment	1M_Det	9.26+/-1.24
52	DMC	cc-pVDZ	Experiment	Hartree-Fock	8.48+/-1.04
111	DMC	vdz-BFD	Experiment	HF tau=0.01	8.00(99)
70	DMC	cc-pVDZ	Experiment	1M_Dets_N0_1_Det	7.66+/-1.19
112	DMC	vdz-BFD	Experiment	FCI, ci_threshold=1.e-4	7.39+/-3.68
108	DMC	vdz-BFD	Experiment	Quantum Package	7.18+/-1.00
75	DMC	cc-pVDZ	Experiment	1M_Dets_N0_1k_Dets(0.95Norme)	7.01+/-1.95
77	DMC	cc-pVDZ	Experiment	1M_Dets_N0_0.75Ecorr	6.70+/-2.89
64	DMC	ANO-Roos-aug-DZ	Experiment	Hartree-Fock	6.60+/-2.28
72	DMC	cc-pVDZ	Experiment	1M_Dets_N0_1k_Dets(N0_det_atoms)	5.32+/-1.35
116	DMC	vtz-BFD	Experiment	0.75_106_115	4.27+/-2.06
60	DMC	ANO-GS_3z	Experiment	Toulouse Table 8 U 12	1.25(7)

FIGURE 21 – The API for the G2 in action

la vérification des résultats : les géométries, les méthodes utilisées, et les énergies obtenues<sup>36</sup>.

La visualisation graphique via le site web PLOTLY [117] a aussi été implémentée permettant une visualisation rapide et portable des données.

#### 4.3.3 Interface graphique

Un assez grand nombre d'heures fut passé à la création d'une interface graphique. Plus exactement au choix technologique pour créer cette interface graphique. Les interfaces graphiques sont souvent écrites suivant le modèle MVC (modèle-vue-contrôleur). La vue est l'interface graphique en elle-même, ce que voit l'utilisateur, le modèle quant à lui correspond à l'ensemble des données – dans notre cas –, et le contrôleur fait la liaison entre les deux. Soit trois questions à se poser :

- Quel langage utiliser ?
- Quelle bibliothèque gèrera la partie graphique ?
- Comment faire communiquer entre elles les différentes parties de l'interface graphique ?

Nous avons en plus de ces points une contrainte supplémentaire assez forte : cette interface graphique doit pouvoir être lancée via un supercalculateur. Ce qui n'est pas une sinécure car d'une part les logiciels disponibles ne sont pas forcément des plus récents, et d'autre part nous n'avons pas les droits d'administrateur pour en installer de nouveaux.

36. Bienvenue à l'heure du *supplementary material* 2.0.

LANGAGE. Au vu de mes affinités, il fut décidé de développer le serveur en `Python`. De plus l'interpréteur est disponible sur tous les supercalculateurs<sup>37</sup>.

BIBLIOTHÈQUE GRAPHIQUE. La technologie « standard » permettant la création d'interfaces graphiques en `Python` est TkInter qui repose sur la technologie Tcl/Tk. Malheureusement, cette technologie est assez primitive, rendant le développement assez long et fastidieux si l'on n'est pas maître de toutes ses arcanes ; ce qui était mon cas. De plus, d'un point de vue graphique cette bibliothèque crée des interfaces que le précédent millénaire n'aurait pas renié. Si l'on veut une interface plus *graphique* il faut se rabattre sur la technologie bien nommée Qt [119]<sup>38</sup>. Les deux plus grands *bindings* sont les bibliothèques PyQt [104] et PySide. Une première version de cette interface graphique fut écrite grâce à PyQt, avant que je ne m'aperçoive que la version de Qt sur les supercalculateurs était suffisamment antédiluvienne pour rendre impossible l'exécution du programme et Qt est très difficilement installable – et c'est un euphémisme – sans droits d'administrateur.

**Technologie web** C'est à ce moment que j'ai réalisé qu'il existe une technologie d'interface graphique qui s'installe très rapidement, partout et sans heurts : les navigateurs web. En effet, les navigateurs servent avant tout à afficher de bien belle façon des éléments visuels. De plus Firefox, comme d'autres, propose des versions portables utilisables à peu près n'importe où. Il nous suffit donc de créer un serveur web en `Python` afin de bénéficier de tous ces avantages ; ceci est facilité grâce aux *web frameworks* développés : l'un des plus populaire est Django [47]. Néanmoins, nous avons choisi Flask [105], car il s'agit d'un *microframework* et de ce fait il ne pèse pas bien lourd, a peu de dépendances et sa prise en main est très rapide (CODE 15).

---

37. À partir de la version 2.6 du moins.

38. Prononcer "cute"...

```

from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'Hello World!'

if __name__ == '__main__':
    app.run()

```

(a) Python File

```

$ python hello.py
* Running on http://127.0.0.1:5000/

```

(b) Execution trace

CODE 15 – Flask Hello World

Utiliser des technologies web a quand même un inconvénient : il faut connaître les technologies qui leur sont associées, c'est-à-dire HTML, Javascript, et CSS. Pourtant, il existe une multitude de tutoriels sur internet, et si l'on souhaite faire des choses assez rudimentaires, cela s'avère aisé.

COMMUNICATION. L'échange de données (Fortran  $\Rightarrow$  Python) se faisait à l'aide de fichiers, ceci avait comme principal problème la latence. En effet, sur les systèmes de fichiers distribués tel que Lustre, il peut y avoir un temps de latence assez grand avant qu'un fichier soit effectivement lu. Ainsi dans une deuxième version nous avons utilisé la bibliothèque ØMQ présentée dans une précédente partie, et nos problèmes de latence ont été résolus.

#### 4.3.3.1 Morale

Nous avons ainsi écrit une interface graphique permettant de lancer nos codes de calcul et de visualiser les résultats. Si l'on doit retenir une chose de cette partie c'est l'utilisation d'un serveur web dans le cas où le souci principal est la portabilité de l'interface graphique ; ou bien l'utilisation de TkInter pour les « pythonistas » les plus téméraires. Il est peut être de bon ton de garder un œil sur la toute nouvelle bibliothèque Flexx [81] qui prend ce parti pris des technologies web mais avec un beau *packaging* en Python et permet ainsi de ne pas avoir à apprendre le HTML et autres.

## 4.4 NOTE SUR LES FICHIERS D'ENTRÉE ET DE SORTIE

J'aimerais parler ici d'un point qui me tient particulièrement à cœur, la gestion des paramètres d'I/O, non pas dans leur version massivement parallèle ou de leur impact dans le temps de calcul mais plutôt dans leur conception même. En effet, bien que souvent peu discutée, leur importance me paraît assez essentielle. En effet ils apparaissent régulièrement dans les griefs<sup>39</sup> qu'ont les utilisateurs de programmes scientifiques :

- La création de fichiers d'entrée est souvent assez « artistique » pour ainsi dire. Les mots clés de ces fichiers d'entrée ne sont pas toujours clairs, leur documentation dépassée ou inexistante et la syntaxe même de ces fichiers peut sembler aberrante ;
- De plus, il est trivial, mais toujours bon de rappeler que des paramètres d'entrée corrompus ou invalides ne peuvent pas donner de bons résultats. Et il est assez rageant que le programme s'interrompe à cause de cela, non dès le début du calcul mais après un temps toujours trop long ;
- Et finalement, le fichier de sortie est souvent trop volumineux et possède une grammaire arbitraire, il faut effectuer des `grep` pour récupérer les informations et faire attention si d'une version à l'autre la syntaxe a changé, ce qui rend le maintien des scripts de post-traitement assez rébarbatif.

Pour toutes ces raisons, nous allons essayer dans la suite de ce chapitre, d'explicitier comment on en arrive à créer des programmes avec ce genre de maladresse et comment, peut être, y remédier.

### 4.4.1 Le poids de l'histoire

Traditionnellement et schématiquement, il existe un unique binaire lisant un fichier texte précisant à la fois les types de calculs à effectuer<sup>40</sup> et les paramètres de calcul<sup>41</sup> : c'est le fichier d'entrée. Ce fichier texte est lu via le flux standard<sup>42</sup>. Une fois ce fichier lu et interprété, le programme commence à générer des données qu'il écrit dans la sortie standard qui peut être redirigée si l'on souhaite en garder une trace<sup>43</sup>.

39. Plus ou moins de mauvaise foi.

40. Par exemple un calcul *Hartree-Fock*.

41. On définit un seuil de convergence pour le SCF, par exemple  $1.0 \times 10^{-8}$ .

42. Le fameux `<` en Bash.

43. Le fameux `>` en Bash.

#### 4.4.2 Du fichier d'entrée

Lorsqu'on veut lire des données dans un fichier, on ne sait généralement pas combien de caractères ou de lignes on souhaite récupérer. Cette information est découverte *au cours* de la lecture du fichier. Il est alors très commode d'utiliser des structures de données dont la taille n'est pas fixée *a priori*, comme les listes chaînées. Dans un langage de bas niveau comme le **Fortran**, la notion de liste n'existe pas dans le langage : les types primitifs n'incluent que la notion de tableau. Si l'on veut utiliser des listes chaînées, il faut donc les programmer soi-même ou utiliser une bibliothèque externe. Il en est de même pour tous les autres outils de gestion de chaînes de caractères<sup>44</sup>. Le **Fortran** étant un langage très répandu dans le domaine du calcul mais peu répandu dans d'autres domaines, il existe un choix assez restreint de bibliothèques destinées à ce type de traitements par rapport à ce qu'on peut trouver dans la communauté du langage C par exemple. Cette rigidité du **Fortran** a un corollaire fort : la rigidité des syntaxes pour les fichiers d'entrée des programmes. Ainsi, tous les programmes ont des fichiers d'entrée avec une syntaxe qui leur est propre ; et que tout utilisateur doit apprendre avant de pouvoir utiliser le programme. Ainsi, il faut laisser deux lignes vides à la fin d'un fichier d'entrée de GAUSSIAN [63] sinon le programme crée une erreur de segmentation dont le débogage s'avère pour le moins difficile pour un béotien. En outre, un fichier de données lisible par un être humain ne permet pas en général d'accès direct simple. Il doit être lu en séquence : il faut donc lire l'intégralité du fichier pour le transformer en une structure de données utilisable par le programme. De plus, paradoxalement, c'est à l'utilisateur de créer le fichier d'entrée *ex nihilo*. Je dis paradoxalement, car si l'on arrive à *parser* un fichier, on peut mécaniquement, en créer un *template*. Enfin, la rigidité du langage de bas niveau utilisé a pour corollaire qu'il est difficile de valider un fichier d'entrée. Il est peut-être envisageable de vérifier que les données satisfont quelques règles simples, mais rendre ces tests systématiques est illusoire, les routines nécessaires devenant vite ubuesques dans ces langages. Ainsi le programme peut crasher à un moment où il utilisera une variable d'entrée initialisée avec des valeurs non valides ; avoir un programme qui plante après quelques heures de calcul du fait qu'un des chemins d'un des fichiers pointe vers un répertoire qui n'existe pas est pour le moins frustrant...

De cet inventaire à la PRÉVERT de mes griefs au sujet des fichiers ressort quelques points :

---

44. Découpage des lignes en liste de mots, recherche de mots-clés dans une chaîne de caractère, *etc.*

- L'idée même d'avoir un fichier d'entrée créé *from scratch* par l'utilisateur comme unique composant de la gestion des paramètres d'entrée est à revoir ;
- L'utilisation d'un langage de bas niveau pour la gestion de ces fichiers n'est peut être pas le choix le plus pertinent ;
- La transcription des fichiers d'entrée définis par l'utilisateur en variables utilisables par le programme étant essentielle et somme toute assez mécanique, il faut l'automatiser au maximum.

#### 4.4.2.1 *Cahier des charges*

Dans l'idéal, l'utilisateur doit avoir à sa disposition un *patron* de fichier d'entrée (*template*) contenant tous les mots clés possibles rangés par catégorie, les valeurs par défaut associées ainsi que leur documentation. Ce patron serait généré automatiquement par un programme tiers et la documentation récupérée directement du code source. Bien sûr, l'utilisateur pourrait modifier la valeur des variables du patron à sa guise.

Ce programme de génération du patron, écrit dans un langage de haut niveau, s'assurerait aussi de la cohérence des données présentes dans ce patron ainsi que de leur écriture dans des fichiers atomiques ayant des noms explicites. Par fichier atomique j'entends qu'à *un* paramètre est associé *un* fichier ; ceci afin de faciliter leur manipulation par des programmes tiers. Toujours dans cette même optique, ce programme génère automatiquement les routines nécessaires à leur manipulation ; le tout dans plusieurs langages.

Finalement l'utilisateur choisit un binaire à lancer<sup>45</sup>. Le binaire lira directement et sans vérification les fichiers atomiques correspondant aux options adéquates grâce aux routines précédemment créées. En effet, les vérifications seraient superflues, car déjà effectuées par le programme de création du fichier d'entrée à partir d'un *patron*.

Cela permet de résoudre plusieurs soucis :

- L'utilisateur n'a pas à se souvenir de tous les mots clés et à retenir la syntaxe propre au fichier d'entrée ;
- Le développeur ne gère la documentation qu'à un seul endroit ;
- Il existe une vérification *a priori* des paramètres ;
- Le fait d'associer un paramètre à un fichier permet alors l'accès direct à la donnée pour le code de calcul ;
- L'utilisateur peut facilement écrire des scripts pour manipuler les paramètres d'entrée via l'API fournie par le programme.

Ces propositions ainsi que leurs implémentations ne sont point illusoires. Dans la suite, nous parlerons plus en détail de l'aspect technique de leur mise en œuvre.

---

45. À un binaire correspond un type de calcul bien précis.

4.4.2.2 *Fichier atomique : EZFIO*

Afin de pouvoir implémenter notre stratégie développée plus haut, il nous faut des routines permettant la manipulation de fichiers atomiques. Ces routines seraient utilisées à la fois par le générateur de patron et par le code de calcul.

```
molecule
  num_atoms    integer
  mass         real      (molecule_num_atoms)
  coord        real      (3,molecule_num_atoms)
```

CODE 16 – EZFIO configuration file

Ainsi, on crée les trois fichiers (`num_atoms`, `mass` et `coord`) qui vont être regroupés dans un répertoire `molecule`<sup>46</sup>. On peut remarquer que des variables peuvent être utilisées pour en dimensionner d'autre.

Grâce à ce simple fichier de configuration (appelé de façon originale `ezfio_config`), des routines permettant la lecture et l'écriture de ces variables sont générées dans de multiples langages (`Fortran`, `Python`, `OCaml` et `Bash`), voir par exemple le CODE 17 pour une illustration de leur utilisation en `Python` et `Fortran`. La récupération des données et leur modification est maintenant somme toute assez triviale, ceci est d'une grande aide à la fois pour le code `Fortran` – où l'on ne doit plus écrire les routines de lecture/écriture – et pour notre futur programme de génération de patron ; en effet, la liste de toutes les catégories et leurs variables associées est facilement récupérable.

```
from EZFIO import EZFIO
ezfio.set_file(filename)
ezfio.molecule_num_atoms = 10
num_atoms = ezfio.get_molecule_num_atoms()
```

(a) Python

```
! Link libezfio.a when compiling
ezfio_set_file(filename)
call ezfio_set_molecule_num_atoms(10)
call ezfio_get_molecule_num_atoms(num_atoms)
```

(b) Fortran

CODE 17 – Example of EZFIO usage

46. Il convient d'insister sur un point qui est déroutant lors des premières utilisations des répertoires générés par l'EZFIO : il n'existe plus de fichier d'entrée regroupant l'ensemble des paramètres ; à la place existent une myriade de fichiers atomiques contenant chacun une et une seule valeur. Ces fichiers sont regroupés dans des répertoires par catégorie et ces répertoires sont eux-mêmes regroupés dans un répertoire principal. Ce répertoire est appelé par abus de langage « le répertoire EZFIO ».

Malheureusement, l'EZFIO ne connaît pas le contexte dans lequel les données seront utilisées. De ce fait, il ne peut pas vérifier la validité des données ; à l'exception de la cohérence entre les dimensions des tableaux et les valeurs des variables les dimensionnant. Par exemple, il ne peut pas vérifier qu'une variable correspondant à un nombre d'atomes doit être positive. Ainsi, il nous faut maintenant trouver un moyen de s'assurer de la cohérence des données manipulées par l'EZFIO. Cette cohérence sera gérée par le même programme qui s'occupera de la génération de patron et sera écrit dans un langage fortement typé : OCaml. Nous allons détailler ce point, dans la prochaine partie.

#### 4.4.2.3 Cohérence des paramètres : langage fortement typé

Dans Quantum Package l'utilisateur peut définir un nombre maximum de déterminants comme paramètre de la simulation. Ce nombre maximum de déterminants possède un type spécial en OCaml : `Det_number_max`. Les valeurs de ce type sont assujetties à une règle simple : elles ne peuvent être négatives. De plus, il est impossible d'assigner un nombre maximum d'atomes (de type `Atom_number_max`) à une variable de type `Det_number_max` ; bien que pourtant les deux valeurs soient des entiers positifs. En effet, les conversions implicites de type sont interdites dans un langage fortement typé<sup>47</sup>. Le typage fort limite donc énormément le champ des erreurs de programmation possible et facilite la gestion de la cohérence.

Dans Quantum Package, cette gestion haut niveau des paramètres d'entrée est gérée par le programme `qp_edit`. Ce programme sert d'intermédiaire entre l'utilisateur et le répertoire EZFIO. Lors de la modification d'un paramètre via ce programme, si une nouvelle valeur ne remplit pas les critères du type associé une erreur sera levée, sinon la modification sera acceptée.

#### 4.4.2.4 Fusion des deux technologies

Afin de résoudre les problématiques liées au fichier d'entrée, nous avons introduit deux technologies :

1. L'utilitaire EZFIO permettant d'accéder facilement aux paramètres d'entrée. C'est un utilitaire de « bas niveau ». Les types des variables sont simples<sup>48</sup>
2. Les routines OCaml permettant la gestion à plus haut niveau des variables grâce à son typage fort.

Il faut maintenant lier les deux technologies. Pour cela, il suffit de définir un dictionnaire permettant de faire la liaison entre le type de

47. C'est même la définition d'un langage fortement typé.

48. i.e *integer, double precision, logical, ...*



```
[Det_number_max]
doc: Max number of determinants in the wave function
type: Det_number_max
default: 1.e+10
```

CODE 18 – Example of a high-level configfile

« bas niveau » de l'EZFIO et le type de « haut niveau » de l'OCaml. Si l'on reprend l'exemple du type OCaml `Det_number_max`, il est associé à un type `Integer` en Fortran. Ce dictionnaire est généré en même temps que la création des types de haut niveaux en OCaml. Le développeur n'a maintenant plus qu'à générer un seul fichier de configuration afin gérer toute la lecture et validation des paramètres d'entrée<sup>49</sup>. Ce fichier est ensuite traduit automatiquement par un programme Python<sup>50</sup> en :

- fichiers de configurations EZFIO et donc des routines permettant la modification des variables ;
- et en routines OCaml permettant leur validation.

Il nous reste maintenant le dernier point : la création d'un patron de fichiers d'entrée.

49. Par convenance ce programme génère aussi les providers d'IRPF90 si besoin est.

50. Il s'agit d'un programme qui crée d'autres programmes dans un langage différent. On parle alors de *meta-programation*.

4.4.2.5 *Patron de fichier d'entrée*

```

Determinants
=====

Max number of determinants in the wave function ::

    n_det_max = 10000

Force the selected wave function to be an eigenfunction of S^2
If true, input the expected value of S^2 ::

    expected_s2 = 0.

Number of requested states, and number of states used for the
Davidson diagonalization ::

    n_states      = 1
    n_states_diag = 1

Number of determinants ::

    n_det = 2

Determinants ::

-0.940598346297
+++++-----
+++++-----

-0.135349396755
+++++-----
+++++-----

```

CODE 19 – Example of template

Les utilisateurs peuvent maintenant modifier et valider les paramètres d'entrée atomiques. Ceci est très pratique pour une utilisation de type *script*. Néanmoins l'utilisateur souhaite en général avoir un aperçu de tous les paramètres d'entrée possibles, leurs valeurs, ainsi que leur documentation. C'est ceci que nous dénommons : un *patron* de fichier d'entrée. Ce patron est généré par le programme OCaml `qp_edit`. Ce programme est généré automatiquement grâce au fichier de configuration de haut niveau décrit plus haut. Il lit les valeurs des paramètres EZFIO

et s'assure de leur cohérence : si des valeurs n'existent pas, il écrit les valeurs par défaut définies dans le fichier de configuration des paramètres d'entrée. Grâce à ces informations, le patron est fabriqué puis affiché à l'écran dans un éditeur de texte<sup>51</sup>. L'utilisateur effectue dès lors les modifications voulues. À la fermeture du fichier, `qp_edit` vérifie la cohérence des données puis enregistre toutes les valeurs dans le répertoire `EZFIO` si aucune erreur n'est détectée. Le `CODE 19` montre une partie du patron de fichier d'entrée ainsi généré.

L'utilisateur a donc un moyen simple et sûr pour définir les valeurs des paramètres d'entrée qu'il souhaite utiliser. Le développeur quant à lui sait que toutes ces données sont toujours présentes et valides dans le répertoire `EZFIO`, ce qui réduit considérablement l'effort de programmation nécessaire en `Fortran`.

#### 4.4.2.6 Exécutable

Une dernière question se pose, celle de la séparation des exécutables. Traditionnellement il n'existe qu'un gros exécutable. Les mots clés à l'intérieur du fichier d'entrée permettent de choisir le type de calcul. Dans le `Quantum Package` il existe un exécutable par type de calcul afin de permettre facilement l'utilisation de scripts et la modularité du code. De plus à terme, il est prévu d'utiliser `Ninja` aussi pour exécuter les calculs. En effet, réaliser un calcul est semblable à la réalisation d'un processus de compilation. Il s'agit d'explorer un arbre de dépendances où chaque nœud est l'exécution d'un binaire. Par exemple pour faire un calcul *Full Configuration Interaction (Full-CI)*, il faut d'abord avoir effectué un calcul *Hartree-Fock* pour avoir des orbitales moléculaires. Que faire si ce calcul n'a pas été effectué ? Mon avis, est que le binaire *Full-CI* devrait toujours supposer que les fichiers dont il a besoin sont présents, et si ce n'est pas le cas il ne devrait pas pouvoir se lancer.

#### 4.4.2.7 Résumé

Nous avons introduit les notions de fichiers atomiques (via l'`EZFIO`) et de la génération automatique de l'`API` pour leur gestion ainsi que l'utilisation du typage fort (via le langage `OCaml`) pour la vérification des données d'entrée. Ceci nous a conduit à écrire un fichier de configuration afin de centraliser les informations. Ce fichier contient les noms, les types, les valeurs par défaut et la documentation des paramètres d'entrée.

Ces fichiers de configuration permettent de construire automatiquement certaines parties du programme `qp_edit`. Ce programme permet de générer un patron de fichier d'entrée contenant tous les paramètres

51. L'éditeur spécifié par la variable d'environnement `$EDITOR`.

d'entrée et les informations pertinentes associées soulageant l'utilisateur de la lourde tâche de l'apprentissage des mots clés.

Ainsi le développeur n'a pas à écrire la documentation à plusieurs endroits mais seulement dans le fichier de configuration. Il ne doit pas non plus écrire les routines d'I/O. L'utilisateur, quant à lui, n'a plus besoin d'apprendre de mots clés ni une syntaxe particulière, et la création de scripts est grandement facilitée.

#### 4.4.3 Fichier de sortie

Le poids de l'histoire offre aux utilisateurs de rediriger la sortie standard du programme dans un énorme fichier de sortie contenant toutes les informations, essentielles ou non, qu'il produit. La façon dont sont écrites ces informations est souvent propre à chaque développeur, à chaque routine, et à chaque version du programme. L'être humain n'en a cure, il peut aussi facilement lire la chaîne de caractère `Energy = -128.26` que `Energie: \n 128.26000000D+000`. Il en est autrement pour des logiciels de post-traitement (par exemple les tests de non-régression).

Une des solutions est de ne plus écrire seulement dans la sortie standard des chaînes de caractères non formatées, mais plutôt d'écrire dans un format bien précis les informations pertinentes (XML par exemple – format utilisé ou en cours de réflexion pour ABINIT (programme assez populaire dans la communauté des simulations en matière condensée) ou encore EZFIO dans Quantum Package). Ceci a l'avantage de considérablement améliorer la facilité la lecture ultérieure des résultats. *A contrario* la lecture pour l'*homo sapiens sapiens* est plus laborieuse, ainsi les deux méthodes cohabitent souvent.

## 4.5 CONCLUSION : RECETTE POUR RÉSOUDRE LA COMPLEXITÉ

Dans cette partie nous avons mis en place plusieurs outils permettant de faciliter l'utilisation et le développement des codes de calcul scientifique. La complexité liée à la création des binaires est réduite en rendant les Makefiles explicites et créés par des scripts tiers. De plus l'utilisation de Ninja a permis un gain de performance notable sur la compilation. L'utilisation du paradigme de programmation IRP a quant à lui permis une simplification drastique du développement dans un code complexe et toujours changeant. L'abandon du modèle de fichier unique d'entrée et de sortie et son remplacement par l'EZFIO a permis de rendre le code beaucoup plus facile à manipuler par des programme tiers. De plus son utilisation conjointe avec un langage fortement typé, comme

l'OCaml, permet de s'assurer de la cohérence des paramètres d'entrée. Nous avons aussi insisté sur la meta-programation. Elle nous a permis d'automatiser au maximum certaines tâches rébarbatives : la création de `providers`, des fichiers de configuration de `Ninja`, les routines d'I/O. Finalement, un processus de développement reposant sur `Git`, `Github` et `Travis CI` a permis de rendre le développement collaboratif plus aisé.

## 5 | CONCLUSION GÉNÉRALE

IMPLÉMENTATION DE NOUVELLES MÉTHODES. La première partie de ce manuscrit, à travers l'exemple des méthodes d'Interaction de Configuration (IC) et *Quantum Monte Carlo*, Monte Carlo quantique (QMC) sur lesquelles j'ai travaillé, a illustré la nécessité, dans le domaine de la chimie quantique, de continuer à développer de nouvelles approches afin de repousser les limites actuelles des simulations. Dans un premier temps, nous avons implémenté les bases de type Slater afin, non seulement d'améliorer la précision des calculs, mais également d'accélérer les simulations. Dans le cas de la série 3d des atomes de transition, nos simulations ont permis d'obtenir les énergies totales non-relativistes les plus précises jamais publiées pour ces atomes. Dans un deuxième temps, nous nous sommes penchés sur l'implémentation des pseudo-potentiels en QMC en adoptant une approche originale adaptée aux fonctions d'onde d'IC. Ceci nous a permis d'alléger le coût calculatoire de nos simulations QMC pour les énergies d'atomisation des 55 molécules composant l'ensemble G2 (*benchmarking* de la méthode). De plus, afin d'améliorer les compensations d'erreurs *fixed-node* lors du calcul des différences d'énergies, nous avons introduit et généralisé la notion de méthode dite « à  $E_{PT2}$  constante » qui permet de construire des fonctions d'onde d'essai pour le QMC cohérentes à nombre variable de déterminants.

QUESTIONNEMENT DES PARADIGMES INFORMATIQUES DE LA CHIMIE QUANTIQUE. Lors de mon travail j'ai été amené à questionner plusieurs paradigmes informatiques de la communauté du calcul scientifique appliqué au domaine des simulations électroniques. Bien que j'ai effectué ce questionnement dans ce contexte scientifique particulier, les aspects les plus fondamentaux que je soulève et les solutions adoptées sont grandement inspirés des réponses élaborées dans d'autres communautés<sup>1</sup> à des problématiques similaires. Ainsi, j'espère que le calcul scientifique de haute performance – *High Performance Computing* (HPC) – pourra bénéficier de cette transversalité.

L'informatique a cet avantage d'être une science où il est possible d'appréhender un problème dans son ensemble et de chercher à un moment donné la solution optimale pour la résolution de ce problème. Cependant – et c'est un point fondamental – eu égard la rapide évolution de l'informatique (tant au niveau de l'architecture des calculateurs que des

---

1. Grille, développement web, finance, etc.

technologies utilisées), les principes ayant guidé la définition de la solution optimale peuvent être invalidés.

Par exemple à l'aube de l'informatique, la création de fichiers ayant un arbre de dépendance assez complexe n'était pas aisée. Afin d'en assurer une construction automatique et fiable, l'utilitaire `Make` fut développé et fut pendant longtemps l'utilitaire idéal pour la création automatique de fichiers. Malheureusement – entropie oblige – `Make` est devenu de plus en plus complexe au cours de ces vingt dernières années ; rendant l'écriture de ses fichiers de configuration assez délicat. Nous avons montré qu'aujourd'hui il est préférable de revenir à une description explicite de l'arbre de dépendance.

De plus, à cette ancienne époque la latence des réseaux était encore acceptable comparativement au temps que pouvait prendre le calcul. Ainsi, de manière à implémenter un parallélisme efficace, il était censé de consacrer les efforts de développement sur la minimisation de la latence inter-processeurs ; c'est dans cet état d'esprit qu'a été développé le *Message Passing Interface* (MPI). Néanmoins, à l'heure actuelle les réseaux sont devenus très lents par rapport à la vitesse de traitement des processeurs ; il ne s'agit donc plus de diminuer les latences réseaux à tout prix mais plutôt de les masquer. La communauté des grilles de calcul, domaine où la latence des communications est par définition prohibitive, a résolu cette difficulté en adoptant un parallélisme non-concurrent et tolérant aux pannes. De plus *a contrario* du MPI qui a des difficultés à passer à l'échelle, ces technologies ont été développées dans le but d'une utilisation sur un très grand nombre de processeurs. Nous avons présenté la bibliothèque `ØMQ` que nous avons utilisée pour implémenter le parallélisme dans notre code `QMC`.

La *barrière de la mémoire* a aussi une conséquence concernant les performances des algorithmes. Il fut un temps où une amélioration de la puissance-crête des supercalculateurs était associée mécaniquement à une plus grande vitesse d'exécution des programmes ; ce n'est plus le cas. En effet, d'un côté nous avons une augmentation du nombre de processeurs par nœud et de l'autre la quasi stagnation de la quantité de mémoire et des débits réseau. Cela conduit à une diminution de la mémoire par cœur et de la bande passante lui étant allouée. Paradoxalement donc, il existe des programmes qui tournent de plus en plus lentement alors que la puissance-crête des supercalculateurs ne fait qu'augmenter. Les algorithmes à « parallélisme embarrassant » (*embarrassingly parallel*) ne semblent donc plus être une facilité mais bien un pré-requis si l'on souhaite tirer partie au maximum des ressources d'un supercalculateur. Nous avons montré l'accélération linéaire obtenue avec notre code stochastique `QMC` jusqu'à quelques dizaines de milliers de processeurs. Cependant, même avec ce type d'algorithme, il reste néanmoins des efforts à faire si l'on veut avoir une exécution efficace. Nous avons vu, par

exemple, les nombreuses stratégies mises en jeu afin d’optimiser le code (vectorisation, structures de données, communications asynchrones, etc).

Les codes progressant en nombre de lignes de code au fil des années, la tendance est d’adopter le paradigme dit *de la programmation orientée objet*. Ceci permet une plus grande modularité des codes par rapport à une écriture purement procédurale et donc un maintien qu’on dit plus facile. Cependant la programmation orientée objet, ne semble pas la plus adaptée au développement de programmes toujours changeants – comme les codes scientifiques – développés par des personnes ayant un niveau de compétence en informatique hétérogène. En effet, la programmation orientée objet, pour remplir son rôle simplificateur, nécessite une grande réflexion *a priori* et une grande expertise, ceci afin de définir correctement les objets et leurs attributs comme dans le cas du fameux problème du cercle et de l’ellipse[125]. Un code académique est par définition très changeant, suite au besoin incessant de tester, et donc d’implémenter, de nouvelles idées. Il est impossible de prévoir *a priori* les caractéristiques des objets qui seront utiles et donc comment les définir correctement. De plus, une bonne connaissance de l’implémentation déjà présente est nécessaire afin de ne pas dupliquer des classes déjà implémentées. Nous avons présenté la méthode *Implicit Reference to Parameters* (IRP) ne possédant aucun de ces défauts. L’IRP permet la création et la résolution automatique des arbres de dépendance des variables nodales d’un programme. Il est ainsi très facile d’ « entrer dans le code » et ainsi de développer de nouvelles approches. De plus, IRPF90 repose sur l’utilisation de structures de données primitives permettant une excellente utilisation des processeurs pour le HPC.

Dans notre domaine, les codes étaient traditionnellement écrits dans un langage unique, le but premier étant la performance. A l’heure actuelle, et au vu de la complexité croissante des codes, il convient peut-être de choisir le meilleur langage pour chaque tâche spécifique. Par exemple, les langages fonctionnels semblent utiles pour écrire les parties du code sensibles devant être dénués d’effets de bord – comme la gestion du parallélisme – ; les langages fortement typés quant à eux sont adaptés à l’écriture de codes où les conversions implicites doivent être proscrites – leur utilisation dans la validation des fichiers d’entrée semble pertinente – ; quant aux langages de bas niveau, ils seront très probablement toujours utilisés pour les calculs proprement dits ; les langages de haut niveau de type Python peuvent servir de liant entre toutes les parties du code [98].

A l’heure de l’*open source* et de la collaboration massive, il est important aussi d’avoir des procédures simples permettant de minimiser les efforts afin de fournir à la communauté un code toujours valide et à jour. Nous avons ainsi discuté l’utilisation de Git, Github et Travis CI et fourni un *workflow* robuste.



Pour conclure cette conclusion, j'aimerais témoigner du grand plaisir que j'ai eu à travailler dans le domaine des simulations scientifiques orientées HPC. Ces simulations se situent à l'interface entre deux mondes : celui de l'informatique, bien sûr ; et celui de la chimie quantique, dans mon cas particulier. Manier ces deux outils a fait de moi un petit « gestaltiste ». De façon moins nombriliste, et à force de travailler dans cet intermonde, je pense sincèrement que nous avons beaucoup à gagner du rapprochement et du dialogue entre les nombreuses diasporas de la communauté scientifique *open source*, autant celles académiques que celles industrielles.

# A | SQGIT : GESTION DES VERSIONS DES FICHIERS SQL AVEC GIT

Comme nous l'avons vu précédemment `Git` est un outil de gestion de versions adapté aux codes sources. Ainsi, il a été conçu pour être utilisé sur des fichiers textes et non pas sur des formats plus évolués (tels les exécutables, ou bien encore les fichiers de bases de données *Structured Query Language* (SQL)). Néanmoins, l'utilisation de `Git` sur de tels formats fonctionne peu ou prou. `Git` arrive à stocker leur historique de modification. Pourtant, la visualisation des différences ainsi que la résolution de conflits lors d'un *merge* est impossible. Nous proposons dans cette annexe une méthode permettant de contourner les problèmes dans le cas particulier des bases de données peu volumineuses (de l'ordre de quelques dizaines de Mo).

## A.1 IDÉE

L'idée principale est de passer d'un fichier binaire de base de données `SQLite` à un fichier *équivalent* écrit en texte brut ; ce fichier pourra être géré par `Git`. L'utilisateur quant à lui continuera à utiliser le fichier `SQLite`. Il nous faut donc d'une part pouvoir passer d'un type de fichier à l'autre (`SQLite`  $\Rightarrow$  texte brut) et d'autre part s'assurer de la cohérence des deux fichiers suscités.

## A.2 MISE EN ŒUVRE

**SQLITE VERS TEXTE.** Le passage du fichier binaire `SQLite` à un fichier texte (`SQLite`  $\Rightarrow$  texte brut) se fait grâce à la commande `SQL dump`. Cette commande permet d'afficher à l'écran l'ensemble des commandes `SQL` nécessaires à la création de la base de données. C'est donc ce *dump* qui sera utilisé par notre logiciel de versionnage.

```
sqlite3 file.db .dump > db_dump.txt
```

La réciproque de cette action (texte brut  $\Rightarrow$  `SQLite`) se fait tout simplement en passant le fichier *dump* au logiciel `SQLite` via le flux d'entrée standard.

```
sqlite3 file.db < db_dump.txt
```

**COHÉRENCE DES DONNÉES** À chaque modification de l'un des deux fichiers son partenaire doit être mis à jour. En effet, ces deux cas de figure peuvent se produire. Par exemple :

- L'utilisateur a modifié le fichier `SQLite`, le *dump* doit donc être remis au goût du jour ;
- Après un `git pull`, le fichier *dump* a été modifié, le fichier `SQLite` doit donc suivre le même chemin.

### A.3 ASPECT TECHNIQUE : PYTHON

Dans le cadre de nos projets, nous utilisons les bases de données `SQLite` via une interface `Python` disponible dans la bibliothèque standard.<sup>1</sup>

J'ai implémenté toutes les idées précédemment décrites dans une bibliothèque `Python` [7].

Permettez moi d'illustrer rapidement quelques points de programmation orientée objet : l'héritage et la redéfinition de méthode.

L'un des avantages de la programmation orientée objet est sa grande flexibilité. Pour l'utilisateur, utiliser un objet ou son enfant est transparent. En effet, les noms de leurs attributs ne changent pas d'une génération à l'autre. Dans notre cas, l'utilisateur utilisera toujours les mêmes fonctions `SQL`<sup>2</sup> qu'il utilise l'objet standard ou notre objet modifié.

Pour l'utilisateur la seule différence consiste à choisir la fonction d'instanciation de l'objet `Connection` (objet contenant toutes les méthodes permettant la modification de la base de données).

Soit pour l'utilisation habituelle :

```
import sqlite3
con = sqlite3.connect('file.db')
```

Soit pour l'utilisation adaptée à `Git` :

```
import sqgite3
con = sqgite3.connect('file.db')
```

Ainsi il est aisé de passer d'une implémentation à l'autre.

La seule différence entre ces deux objets et la redéfinition de la méthode `commit` dans l'enfant. Avant chaque `commit` cette méthode vérifie si elle doit mettre à jour la base de données (`dump`  $\Rightarrow$  `text`) puis met à jour le fichier `dump` (`text`  $\Rightarrow$  `dump`).

1. Un rapide tutoriel utilisant cette bibliothèque est disponible[17].

2. Pour exécuter une commande, pour récupérer des valeurs, pour les valider, etc.

# B

## DIDACTICIEL QUANTUM PACKAGE & QMC=CHEM

Dans cette partie nous allons présenter – *from strach* – un calcul *Fixed-Node Diffusion Monte Carlo* (FN-DMC) avec utilisation de pseudo-potentiels pour la molécule  $N_2$  avec une fonction d'onde d'Interaction de Configuration (IC) sélectionnée obtenue par *Configuration Interaction using Perturbative Selection done Iteratively* (CIPSI). Dans un premier temps l'installation du Quantum Package sera présentée ; puis son utilisation pour produire une fonction d'onde à 10 000 déterminants. Dans un second temps, on utilisera QMC=Chem pour effectuer le calcul FN-DMC.

### B.1 QUANTUM PACKAGE

Dans cette partie nous présentons en détail la configuration, puis l'installation du Quantum Package, son utilisation pour la création des fichiers d'entrée, un calcul *Hartree-Fock*, puis finalement un calcul CIPSI à 10 000 déterminants dans l'espace *Full Configuration Interaction* (Full-CI) pour la molécule  $N_2$ .

#### B.1.1 Téléchargement, configuration et compilation

Le schéma traditionnel d'installation est le fameux triptyque : `tar -xzf`, `./configure`, `make`. Le Quantum Package le suit dans les grandes lignes.

TÉLÉCHARGEMENT. On commence par récupérer le dépôt hébergé sur Github. On montre ici la récupération en ligne de commandes via Git et ssh<sup>1</sup>.

```
$ cd /tmp/
$ git clone git@github.com:LCPQ/quantum_package.git
Cloning into 'quantum_package'...
remote: Counting objects: 10561, done.
Receiving objects: 100% (10561/10561), 25.34 MiB, done.
$ ls
quantum_package
```

1. Une archive peut aussi être téléchargée via l'URL [https://github.com/LCPQ/quantum\\_package/archive/master.tar.gz](https://github.com/LCPQ/quantum_package/archive/master.tar.gz).

**CONFIGURATION.** Une fois le code téléchargé, il faut effectuer l'étape dite de configuration via le script `configure`. Ce script a deux objectifs principaux : d'une part l'installation des programmes tiers requis (les dépendances) et d'autre part de créer le fichier de configuration permettant la création automatique des binaires<sup>2</sup>; étape vulgairement appelée « étape de compilation ».

**Installation des dépendances.** Comme souvent en informatique, il s'agit de créer un arbre puis de le parcourir. Dans un premier temps, on vérifie la présence de tous les programmes disponibles<sup>3</sup>, puis en fonction de cela, on crée l'arbre de dépendances. Par exemple dans l'exemple qui suit, on a besoin d'installer entre autres choses `OCaml`, qui a besoin lui-même de `ZLIB`. Puis cet arbre est résolu grâce à `Ninja` : il s'occupe de télécharger, puis d'installer les programmes dans le bon ordre.

```
$ cd quantum_package
$ ./configure --production ./config/gfortran.cfg
Checking what you need to install and what is available

Checking resultsFile... [ FAIL ]
Checking graphviz... [ FAIL ]
Checking python... [ OK ]
( /home/travis/virtualenv/python2.6.9/bin/python )
Checking ezfio... [ FAIL ]
Checking irpf90... [ FAIL ]
Checking ninja... [ FAIL ]
Checking curl... [ OK ] ( /usr/bin/curl )
Checking gcc... [ OK ] ( /usr/bin/gcc )
Checking make... [ OK ] ( /usr/bin/make )
Checking zlib... [ FAIL ]
Checking ocaml... [ FAIL ]
Checking patch... [ OK ] ( /usr/bin/patch )
Checking m4... [ OK ] ( /usr/bin/m4 )
Checking emsl... [ FAIL ]
Checking docopt... [ FAIL ]
You will need to install:
* resultsFile * graphviz * emsl
* zlib        * ezfio    * ninja
* ocaml       * irpf90   * docopt
- -
| .- - -|_ -. | | -. _|_ o - .-
-|- | | _> |- (-| | | (-| |- | (-) | |

Install ninja... [ OK ]
Creating build.ninja... [ OK ]
(/home/travis/build/LCPQ/quantum_package/install/build.ninja)
Installing the dependencies with Ninja... [ OK ]
```

2. Makefile dans la plupart des cas, et `Build.ninja` pour `Quantum Package`.

3. De la même façon que le fait l'utilitaire `autoconf`.

**Création du fichier de configuration.** Ce script permet aussi la création du fichier de configuration pour *Ninja* permettant la création des binaires. Dans cet exemple nous utilisons le fichier de configuration pour *gfortran* – `./config/gfortran.cfg` – fourni par défaut avec le code. Il contient un ensemble d'options de configuration<sup>4</sup> écrit dans une grammaire simple. Il est aussi possible d'optimiser cette étape de création de binaires en passant l'option `--production` au script ; il n'est alors plus possible de créer chaque binaire indépendamment, mais la compilation est beaucoup plus rapide.

```

_
|_ o . _ _ . | o _ _
| | | | ( _ | | | / _ (/_

Creating quantum_package.rc... [ OK ]
(/home/travis/build/LCPQ/quantum_package/quantum_package.rc)
Creating build.ninja... [ OK ]
Now for standard installation:
source /home/travis/build/LCPQ/quantum_package/quantum_package.rc
ninja
cd ocaml; make

PS : For more info on compiling the code,
      read the COMPILE_RUN.md file.

```

Une fois la machine configurée – si *OCaml* est déjà installé, ceci prend une vingtaine de secondes –, on peut maintenant lancer l'étape dite de compilation.

**COMPILATION.** Pour ce faire, il faut dans un premier temps charger l'environnement nécessaire à *Quantum Package* contenant toutes les variables d'environnement nécessaires au bon fonctionnement du code<sup>5</sup>. Puis, on peut choisir d'installer des modules supplémentaires. En effet, afin de minimiser le temps de compilation et donc de ne rien compiler qui serait superflu, *Quantum Package* est livré nu, sans aucun binaire. Dans notre exemple nous allons seulement installer les modules qui nous intéressent, c'est-à-dire les modules *Hartree-Fock* et *Full-CI*. Une fois ces modules installés nous pouvons créer les binaires de calcul et les binaires plus généraux nécessaires à leur gestion : *Ninja* s'occupe de la compilation des binaires *fortran* (une demi-minute), et *Make* (pour le moment) s'occupe d'*OCaml* (une trentaine de secondes également). Au total, la compilation du code prend donc environ une minute.

4. Quel compilateur utiliser, avec quels options de compilation.

5. Ce fichier permet entre autre à *Python* de savoir où chercher ses bibliothèques.

```

$ source ./quantum_package.rc
$ qp_install_module.py install Hartree_Fock Full_CI
You will need all these modules
['Perturbation', 'Selectors_full', 'Generators_full',
 'Full_CI', 'Hartree_Fock', 'Properties']
Installation... [ OK ]
You can now compile as usual
'cd /home/travis/build/LCPQ/quantum_package ; ninja' for exemple
or --in developement mode-- you can cd in a directory and compile here
$ ninja
[250/250] Link: full_ci
$ make -C ocaml > /dev/null

```

Le code est maintenant prêt à être utilisé.

### B.1.2 Lancement des calculs

CRÉATION DES PARAMÈTRES GÉNÉRAUX. Nous allons commencer un calcul *ex-nihilo*. Il faut commencer par créer un répertoire contenant toutes les informations relatives aux données du système qu'on souhaite étudier, c'est-à-dire sa géométrie ainsi que l'ensemble de fonctions de base utilisées. Dans *Quantum Package*, nous n'utilisons pas de fichiers texte, mais une base de données –appelée *Easy Fortran I/O library generator* (EZFIO) – afin de conserver ces informations. Le façon la plus simple pour créer cette base de données est d'utiliser l'utilitaire `qp_create_ezfio_from_xyz`. Étonnamment, il prend comme paramètre un fichier `xyz` afin de créer un répertoire EZFIO<sup>6</sup>, afin de récupérer les coordonnées de la géométrie expérimentale de la molécule N<sub>2</sub>. Le second argument de cette fonction est le nom de la base utilisée<sup>7</sup>. L'argument `-p` permet quant à lui, de préciser que nous allons utiliser une base adaptée à l'utilisation de pseudo-potentiels.

```

$ cd /tmp/
$ ~/G2_CLI/scemama.py get_xyz --geo Experiment --ele N2 | tee N2.xyz
2
N2 Geo: Experiment Mult: 1 symmetry: 14
N 0.0 0.0 0.5488
N 0.0 0.0 -0.5488

$ qp_create_ezfio_from_xyz N2.xyz -p -b "vdz"
$ ls
N2.ezfio N2.xyz

```

HARTREE FOCK. Nous allons maintenant effectuer un calcul *Hartree-Fock*, nous utilisons tous les paramètres par défaut ; si l'on veut changer

6. Dans cet exemple, nous utilisons l'utilitaire `G2_CLI` [6].

7. Grâce à l'interfaçage avec la base de données locale d'*Environment Molecular Sciences Laboratory, Pacific Northwest National Laboratory, US Department of Energy* (EMSL) [8] nous pouvons utiliser n'importe quel type de fonctions base présent sur ce site. Ici nous utilisons la base `vdz`.

des paramètres de la simulation, l'outil `qp_edit` est disponible. Nous l'avons présenté dans les parties précédentes. Le lancement du calcul s'effectue grâce à l'utilitaire `qp_run`, qui permet un accès facile aux nombreux binaires éparpillés à travers le code.

```
$ qp_run SCF N2.ezfoo
=====
Quantum Package
=====
Date : 2015-09-02 19:35:00.479782+02:00

[...]

Nuclear Coordinates (Angstroms)
=====

=====
Atom      Charge      X      Y      Z
=====
N          5.000000    0.000000  0.000000  0.548800
N          5.000000    0.000000  0.000000 -0.548800
=====

-----

.. >> [ WALL TIME: 0.189600 s ] [ CPU TIME: 0.092005 s ] << ..

[...]

* thresh_scf                                0.1000000000000000E-09

.. >> [ WALL TIME: 1.021200 s ] [ CPU TIME: 3.364210 s ] << ..

* n_it_scf_max                               200

.. >> [ WALL TIME: 1.021400 s ] [ CPU TIME: 3.368210 s ] << ..

=====
N      Energy      Energy diff      Density diff      Save
=====
1      -17.8544319645  -1.0000000000    0.0000000000
2      -19.5080047304  -1.6535727659    0.0205635991  X
3      -19.5109273496  -0.0029226192    0.0055747399  X
4      -19.5109516326  -0.0000242830    0.0026777717  X
5      -19.5109520714  -0.0000004388    0.0018484098  X
6      -19.5109520799  -0.0000000085    0.0015354757  X
7      -19.5109520800  -0.0000000002    0.0013994696  X
=====

* Hartree-Fock energy                        -19.51095208001711

.. >> [ WALL TIME: 1.490800 s ] [ CPU TIME: 4.832301 s ] <<< ..

Wall time : 1.62728s
```



REMARQUE SUR L'UTILISATION DU CACHE. Chaque fois qu'une donnée est lue de l'input, on voit sa valeur et le temps CPU écoulé. On peut ainsi constater la localité temporelle (et donc la bonne utilisation du cache) conférée par l'*Implicit Reference to Parameters* (IRP). En effet, les Providers de `thresh_scf` et `n_it_scf_max` sont initialisés juste avant le cycle SCF.

*Full-CI.* De la même façon, nous effectuons un calcul *Full-CI* avec les valeurs par défaut, soit 10 000 déterminants et un calcul final de l'énergie  $E_{PT2}$ .

```

$ qp_run full_ci N2.ezfio
=====
Quantum Package
=====
Date : 2015-09-02 19:38:09.923640+02:00

[...]

.. >> [ WALL TIME: 16.948400 s ] [ CPU TIME: 66.668166 s ] << ..

Davidson Diagonalization
-----

* Number of states                                1
* Number of determinants                          10000

=====
Iter          Energy          Residual
=====
  1   -19.8290392011    0.164562E-03
  2   -19.8290774563    0.236402E-04
  3   -19.8290828211    0.193955E-05
  4   -19.8290834347    0.260692E-06
  5   -19.8290835161    0.276268E-07
  6   -19.8290835251    0.301596E-08
  7   -19.8290835262    0.421624E-09
  1   -19.8290835262    0.421624E-09
  2   -19.8290835263    0.587518E-10
=====

.. >> [ WALL TIME: 17.267600 s ] [ CPU TIME: 67.944246 s ] << ..

[...]

Final step
N_det    =    10000
N_states =         1
PT2      =    -1.711514185213581E-002
E         =    -19.8290835263397
E+PT2     =    -19.8461986681919
-----
* Saved determinants                                10000
Wall time : 2.03931m

```

Le calcul d'une bonne approximation du *Full-CI* pour la molécule N<sub>2</sub> avec pseudo-potentiels avec une base de type double-zeta ne prend que 2 min sur mon ordinateur de bureau.

**RÉCUPÉRATION DES DONNÉES** Le résultat est à la fois affiché dans la sortie standard mais est aussi écrit dans le répertoire EZFIO. Dans cet exemple nous utilisons l'*Application Program Interface* (API) EZFIO pour l'interpréteur Bash.

```
$ ezfio set_file N2.ezfio
$ ezfio get hartree_fock energy
-19.5109520800397
$ ezfio get full_ci energy
-19.8290835263397
```

## B.2 QMC=CHEM

Quantum Package a été développé principalement pour la création de fonctions d'essai pour le FN-DMC. Ainsi nous allons maintenant voir une utilisation de QMC=Chem<sup>8</sup> en utilisant la fonction d'onde précédemment générée.

### B.2.1 Commande

Pour utiliser QMC=Chem, une seule commande est nécessaire, la bien nommée `qmcchem`. Elle permet de modifier les paramètres, de lancer le calcul, d'afficher les résultats, *etc.*

```
$ source /tmp/QmcChem/qmcchemrc
$ qmcchem help
QMC=Chem command

qmcchem SUBCOMMAND

=== subcommands ===

debug    Debug ZeroMQ communications
edit     Edit input data
result   Displays the results computed in an EZFIO
         directory
run      Run a calculation
stop     Stop a running calculation
version  print version information
help     explain a given subcommand (perhaps recursively)
```

Voici par exemple, toutes les commandes possibles et les description utiles afin d'éditer les paramètres d'entrée en ligne de commande.

---

8. L'installation est en cours de réécriture ; ainsi elle sera omise.

```

$ qmcchem edit -help
Edit input data

qmcchem edit EZFIO_FILE [FLAGS]

=== flags ===

[-c Clear]          blocks
[-e energy]         Fixed reference energy to normalize DMC
                    weights
[-f 0|1]            Correct wave function to verify
                    electron-nucleus cusp condition
[-j jastrow_type]   Type of Jastrow factor
                    [ None | Core | Simple ]
[-l seconds]        Length (seconds) of a block
[-m method]         QMC Method : [ VMC | DMC ]
[-n norm]           Truncate CI coefficients below
[-s sampling]       Sampling algorithm :
                    [ Langevin | Brownian ]
[-t seconds]        Requested simulation time (seconds)
[-ts time_step]     Simulation time step
[-w walk_num]       Number of walkers per CPU core
[-wt walk_num_tot] Total number of stored walkers for restart
[-help]            print this help text and exit
                    (alias: -?)

```

### B.2.2 Calcul

Nous allons commencer par un calcul *Variational Monte Carlo*. (VMC), puis un calcul FN-DMC.

VARIATIONAL MONTE CARLO. Dans un premier temps, nous enlevons tous les déterminants qui au total contribuent à moins de  $1.0 \times 10^{-5}$  de la norme de la fonction d'onde, ce qui permet de réduire le nombre de déterminants d'un facteur deux. De plus, nous voyons que ces 5129 déterminants peuvent se réduire en 266 déterminants  $\alpha$  et  $\beta$  uniques. Nous voyons aussi, que ce calcul est identifié par une clé MD5 unique. A chaque modification des paramètres cette clé changera ; ceci permet de vérifier que l'input a correctement été déployé dans les simulations massivement parallèles, et permet aussi d'associer des résultats à des données d'entrée.

```

$ qmcchem edit -m VMC -t 60 -l 5 -n "1.e-5" N2.ezfio
Info : MD5 key changed
-> 5304e9a147ed539c72b0b350cf091d3a
$ qmcchem_info N2.ezfio
Number of determinants           : 5159
Number of unique alpha/beta determinants : 266 / 266
Closed-shell MOs                 : 0
Number of MOs in determinants    : 28
$ create_walkers N2.ezfio
<E_loc>      min            max
-21.67027    -195.6037      125.8289
-21.79582    -304.9212      131.0891
-21.66238    -199.8186      234.8026
-21.71794    -342.6403      229.9974
Generated      30 walkers
$ salloc -n 8
salloc: Granted job allocation 3555
$ qmcchem run N2.ezfio
MD5 : 5304e9a147ed539c72b0b350cf091d3a
Scheduler : SLURM
Launcher  : srun
27828 : qmcchem run -d N2.ezfio
Server address: tcp://130.120.229.139:24866
27849 : srun qmcchem run -q tcp://130.120.229.139:24866 N2.ezfio
Stopping
      Cpu : 8.14015m
      Wall : 1.04968m
      Accep : 0.9828424574
      E_loc : -19.8159048363 +/- 0.0046660196
      E_loc_qmcvar : 3.1180554377 +/- 0.2555312083
      Speedup : 7.75 x
Finalizing...Done

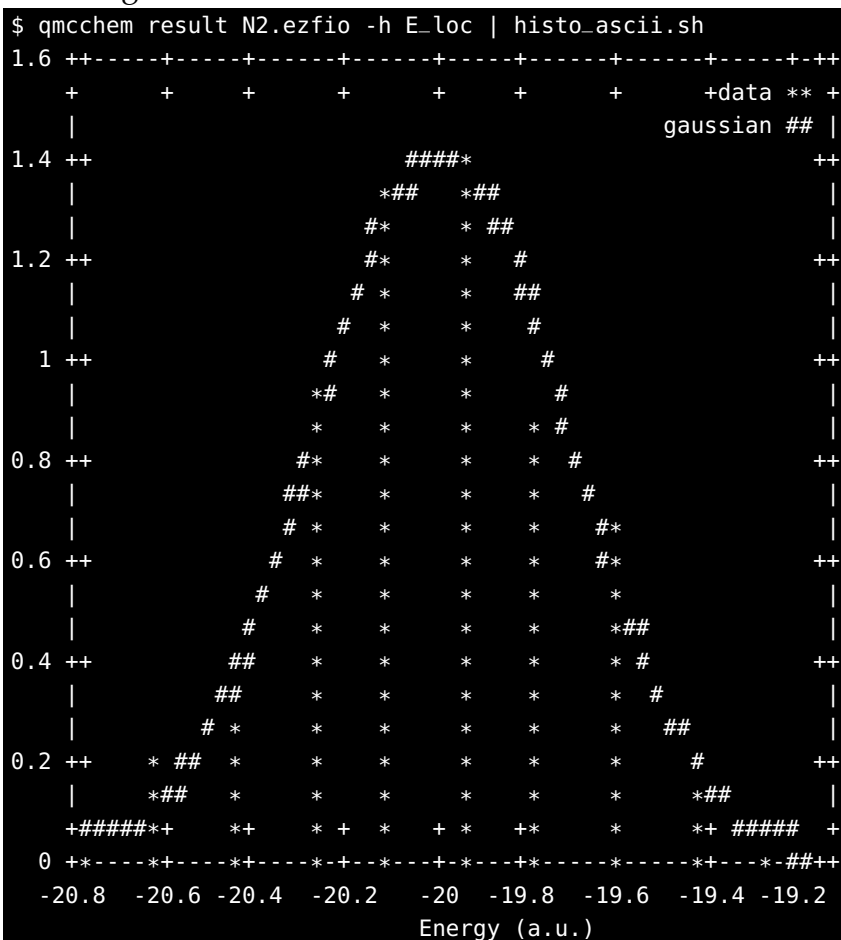
```

Nous voyons que l'énergie obtenue après troncature de la fonction d'onde est de  $-19.8159(47) E_h$  à comparer à la valeur  $-19.8290 E_h$  précédente. De plus, le *speedup* est somme toute assez raisonnable compte tenu de la faible durée du calcul : cela correspond à 99.5% de code parallèle, soit 0.29 s de code séquentiel.

**FN-DMC.** Nous allons maintenant effectuer un calcul FN-DMC qui est le cœur de ce code. Nous en profitons pour montrer une façon alternative de récupérer les valeurs stockées dans notre base de données grâce à un simple cat.

```
$ qmcchem edit -e $(cat N2.ezffio/full_ci/energy_pt2) \
-m DMC -ts 0.001 -s Brownian N2.ezffio
Info : MD5 key changed
      5304e9a147ed539c72b0b350cf091d3a
-> 216e3fc0853ceaab8881c741d96e8199
$ qmcchem run N2.ezffio
MD5 : 216e3fc0853ceaab8881c741d96e8199
Scheduler : SLURM
Launcher : srun
29660 : qmcchem run -d N2.ezffio
Server address: tcp://130.120.229.139:27138
29681 : srun qmcchem run -q tcp://130.120.229.139:27138 N2.ezffio
Stopping
      Cpu : 1.33481h
      Wall : 10.0245m
      Accep : 0.9985002026
      E_loc : -19.9851005473 +/- 0.0261395554
      E_loc_qmcvar : 1.2576693369 +/- 0.0997910496
      Speedup : 7.99 x
Finalizing...Done
```

Nous remarquons que l'énergie est bien plus basse que l'énergie variationnelle. On peut aussi vérifier que la répartition des énergies locales est bien gaussienne.





C

ACCURATE NON RELATIVIS-  
TIC GROUND-STATE ENER-  
GIES OF 3d TRANSITION ME-  
TAL ATOMS



# Accurate nonrelativistic ground-state energies of 3d transition metal atoms

A. Scemama, T. Applencourt, E. Giner, and M. Caffarel<sup>1</sup>

*Lab. Chimie et Physique Quantiques, CNRS-Université de Toulouse, France.*

We present accurate nonrelativistic ground-state energies of the transition metal atoms of the 3d series calculated with Fixed-Node Diffusion Monte Carlo (FN-DMC). Selected multi-determinantal expansions obtained with the CIPSI method (Configuration Interaction using a Perturbative Selection made Iteratively) and including the most prominent determinants of the full CI expansion are used as trial wavefunctions. Using a maximum of a few tens of thousands determinants, fixed-node errors on total DMC energies are found to be greatly reduced for some atoms with respect to those obtained with Hartree-Fock nodes. The FN-DMC/(CIPSI nodes) ground-state energies presented here are, to the best of our knowledge, the most accurate values reported so far. Thanks to the variational property of FN-DMC total energies, the results also provide lower bounds for the absolute value of all-electron correlation energies,  $|E_c|$ .

**Keywords:** Quantum Monte Carlo (QMC), Fixed-Node Diffusion Monte Carlo (FN-DMC), Configuration Interaction using a Perturbative Selection made Iteratively (CIPSI), Fixed-Node Approximation, Atomic Ground-state Energies, 3d Transition Metal atoms

## I. INTRODUCTION

An accurate knowledge of nonrelativistic ground-state energies of atoms is known to be of great interest for computational chemistry. Atomic total energies are indeed routinely used to calibrate theoretical studies in electronic structure theory. For example, let us cite the search for more accurate exchange-correlation energy functionals in Density Functional Theory (DFT), the calibration of various approximations in wavefunction-based approaches (finite basis set effects, truncation at a given order in multi-particle excitations, etc.), the study of the fixed-node approximation in quantum Monte Carlo (QMC), the definition of alternative/exotic electronic approaches, etc. Furthermore, by combining experimental results and accurate nonrelativistic values, some valuable information about relativistic effects can also be obtained.

Here, accurate nonrelativistic all-electron ground-state energies for the metal atoms of the 3d series (from Sc to Zn) are reported. Calculations are performed using the Fixed-Node Diffusion Monte Carlo (FN-DMC) approach, a quantum Monte Carlo (QMC) method known to be particularly powerful for computing ground-state energies.<sup>1,2</sup> An overwhelming number of works have been devoted to the calculation of accurate atomic ground-state energies using various highly-correlated approaches; here, we shall only restrict ourselves to briefly summarize the typical accuracies presently achievable. For small atoms (say, less than 10 electrons, that is, from H to Ne for neutral atoms) very accurate values with errors smaller than  $10^{-4}$ – $10^{-5}$  a.u. (or much smaller for the lightest atoms) can be obtained. For heavier atoms up to Ar (18 electrons), the accuracy reduces to the millihartree level ( $\sim$  chemical accuracy). For even bigger atoms (say, more than 20 electrons) to obtain a precision close the millihartree is problematic and only a small number of results have been published. Regarding quantum Monte Carlo studies using FN-DMC or a closely related QMC variant, most of the works have been concerned with atoms from

Li to Ne; for the most recent ones, see *e.g.* [3–6]. For heavier atoms, most calculations have been performed using pseudo-potentials to remove core electrons (see *e.g.* [7],[8], and [9]). At the all-electron level, very little has been done. We can essentially cite the FN-DMC calculations by Ma *et al.*<sup>10</sup> for the Ar, Kr, and Xe atoms, calculations for the Cu atom and its cation,<sup>11,12</sup> and two studies by Buendia and collaborators for 3d transition metal atoms.<sup>4,6</sup>

It is fair to say that FN-DMC is presently the most accurate method for computing total ground state energies for large enough electronic systems. Potentially, diffusion Monte Carlo allows an exact stochastic solution of the Schrödinger equation. Several sources of error make in practice FN-DMC simulations non-exact. However, most of the errors are not of fundamental nature and can be easily kept under control (mainly, the statistical, finite time-step, and population control errors). In contrast, the fixed-node error resulting from the use of trial wavefunctions with approximate nodes is much more problematic since, up to now, no simple and systematic scheme to control this error has been devised. Note that the fixed-node approximation is variational,  $E_{\text{FN}} \geq E_0$ , a convenient property to get upper and lower bounds for total energies and absolute values of correlation energies, respectively [in contrast, *e.g.*, with the non-variational character of the commonly used CCSD(T) or Møller-Plesset approaches].

To decrease the fixed-node error, the common strategy is to use trial wavefunctions of the best possible quality and to resort to (large-scale) optimization techniques to get the best parameters entering the trial wavefunction (usually, via minimization of the total energy and/or its variance). A great variety of functional forms have been introduced for the wavefunction (see, *e.g.* [12–22]), and different optimization techniques designed to be efficient in a Monte Carlo context have been developed (*e.g.* [23]). In this work, accurate nodes are built by employing a new class of trial wavefunctions very recently introduced in the context of QMC simulations.<sup>24</sup> The wavefunction is

expressed as a truncated Configuration Interaction (CI) expansion containing up to a few tens of thousands of determinants. The expansion is built thanks to the CIPSI method (Configuration Interaction using a Perturbative Selection made Iteratively). The key point with CIPSI is the possibility of extracting the most prominent determinants of the FCI expansion. Very recent applications on several systems have shown that accurate nodes can be obtained.<sup>24,25</sup> Furthermore, it has been observed that the quality of nodes appears to systematically improve when the number of determinants is increased. This property is remarkable since it allows a simple control of the fixed-node error. Finally, an important practical property of CIPSI is that trial wavefunctions are generated in an automated way through the deterministic selection and diagonalization steps and the initial many-parameter stochastic optimization usually performed in QMC is avoided here.

The FN-DMC/(CIPSI nodes) total ground-state energies of metal atoms of the 3d series obtained here are compared to the very recent results of Buendia *et al.*<sup>6</sup> An important and systematic improvement is obtained (lower total fixed-node energies). To the best of our knowledge, the data presented here are the best ones reported so far. Thanks to the variational property of FN-DMC total energies, the results also provide lower bounds for the absolute value of all-electron correlation energies,  $|E_c|$ .

## II. METHODS AND COMPUTATIONAL DETAILS

### A. Configuration Interaction using a Perturbative Selection made Iteratively (CIPSI)

The CIPSI method, and similar approaches closely related, have been introduced and developed a long time ago by a number of authors (see, *e.g.*, [26–34]). In a few words, the approach consists in building the multi-determinantal expansion *iteratively* by selecting at each step one determinant (or a group of determinants) according to a perturbative criterion. A determinant  $D_i$  (or a group of determinants) is added to the current wavefunction if its (their) energetic contribution(s) calculated by second-order perturbation theory is (are) sufficiently large. In this way, the wavefunction is built hierarchically, the most important determinants of the FCI solution entering first in the expansion. Such a construction must be contrasted with standard approaches (CIS, CISD, etc.) where the contributions at a given order are calculated by considering all possible particle-excitations with respect to some reference wavefunction (usually, the Hartree-Fock (HF) solution). The CIPSI multi-determinantal expansion is thus much more compact than standard expansions, an important practical point for FN-DMC where the trial wavefunction and its derivatives must be computed a very large number of times during the simulations. Let us now briefly summarize the algorithm. More details can be found in Ref.

[24] and in the original works cited above.

In multi-determinantal expansions the ground-state wavefunction  $|\Psi_0\rangle$  is written as a linear combination of Slater determinants  $\{|D_i\rangle\}$ , each determinant corresponding to a given occupation by the  $N_\alpha$  and  $N_\beta$  electrons of  $N = N_\alpha + N_\beta$  electrons among a set of  $M$  spin-orbitals  $\{\phi_1, \dots, \phi_M\}$  (restricted case). The best representation of the exact wavefunction in the entire determinantal basis is the Full Configuration Interaction (FCI) wavefunction written as

$$|\Psi_0\rangle = \sum_i c_i |D_i\rangle \quad (1)$$

where  $c_i$  are the ground-state coefficients obtained by diagonalizing the Hamiltonian matrix,  $H_{ij} = \langle D_i | H | D_j \rangle$ , within the orthonormalized set,  $\langle D_i | D_j \rangle = \delta_{ij}$ , of determinants  $|D_i\rangle$ .

In its simplest form, the multi-determinant wavefunction is iteratively built as follows. Let us call  $|\Psi_0^{(n)}\rangle = \sum_{i \in S_n} c_i^{(n)} |D_i\rangle$  the current wavefunction at iteration  $n$  where  $S_n$  is the set of selected determinants at iteration  $n$ . Typically, at the initial step  $n = 0$  a mono-determinantal HF-type or a short CAS-SCF-type wavefunction is used. The first step consists in collecting all *different* determinants  $|D_{i_c}\rangle$  connected by  $H$  to  $|\Psi_0^{(n)}\rangle$ , that is  $\langle \Psi_0^{(n)} | H | D_{i_c} \rangle \neq 0$ . Then, the second-order correction to the total energy resulting from each connected determinant is computed

$$\delta e(|D_{i_c}\rangle) = -\frac{\langle \Psi_0^{(n)} | H | D_{i_c} \rangle^2}{\langle D_{i_c} | H | D_{i_c} \rangle - E_0^{(n)}} \quad (2)$$

and the determinant (or group of determinants)  $|D_{i_c}^*\rangle$  associated with the largest  $|\delta e|$  (or greater than a given threshold) is (are) added to the reference subspace:

$$S_n \rightarrow S_{n+1} = S_n \cup \{|D_{i_c}^*\rangle\}$$

Finally, the Hamiltonian matrix is then diagonalized within  $S_{n+1}$  to obtain the new wavefunction at iteration  $n + 1$  and the process is iterated until a target size  $N_{\text{dets}}$  for the reference subspace is reached. The CIPSI wavefunction issued from this selection process is the trial wavefunction used here for FN-DMC.

### B. Fixed-Node Diffusion Monte Carlo (FN-DMC)

For a detailed presentation of the theoretical and practical aspects of FN-DMC, the reader is referred to the literature, *e.g.* [35–37]. Here, let us just emphasize that the central quantity of such approaches is the trial wavefunction  $\Psi_T$  determining both the magnitude of the fixed-node error through its approximate nodes and the quality of the statistical convergence (good trial wavefunctions

imply small statistical fluctuations). The computational cost of FN-DMC is almost entirely determined by the evaluation at each Monte Carlo step of the value of  $\Psi_T$  and its first (drift vector) and second derivatives (Laplacian needed for the local energy). In view of the very large number of MC steps usually required (typically at least billions and often much more) it is essential to be able of computing such quantities very rapidly. In the present work, the typical size of the expansion considered is a few tens of thousands of determinants. Some care is thus required when computing such expansions to keep the computational cost reasonable. The various aspects regarding this problem are presented in Ref. [38].

### C. Computational Details

The atomic basis sets used for the calculations were the Slater-type orbitals of Bunge<sup>39</sup> supplemented with four additional  $4f$  and three  $5g$  functions (a total of 112 atomic basis functions). All the CIPSI calculations were performed using Hartree-Fock molecular orbitals using the code developed in our group (quantum package), and all the FN-DMC calculations were performed using QMC=Chem.<sup>40</sup>

For each atom, the CIPSI calculation was stopped when more than  $10^6$  determinants were selected in the variational wave function. This wave function was then truncated such that the least significant determinants contributing to 0.5% of the norm of the wave function were discarded:  $10^4$ – $5 \cdot 10^4$  determinants were kept. This wave function was used *without any modification* as the trial wave function for the FN-DMC calculations (no Jastrow factor was used).

For the FN-DMC calculations, we have employed the algorithm described in ref [41] allowing us to use a small constant number of walkers. A block consisted in 30 walkers performing 5000 steps with a time step of  $10^{-5}$  a.u., a value chosen such that the time-step error was smaller than the statistical error. Long enough simulations have been performed to make the statistical error negligible with respect to the fixed-node one: depending on the atom, a number of blocks between  $7 \cdot 10^4$  and  $1.5 \cdot 10^5$  were calculated ( $\sim 10^{10}$  MC steps).

## III. RESULTS

In table I the variational energy, the number of determinants in the CIPSI expansion, and an estimate of the percentage of the total correlation energy (CE) recovered for each trial wavefunction  $\Psi_T$  used in FN-DMC are given. The CE's reported are calculated from the recommended values given recently by McCarthy and Thakkar (denoted as McCT in what follows).<sup>42</sup> In sharp contrast with the present work, these values have not been computed directly from a unique (very) accurate energy calculation but have been obtained indirectly by combin-

ing Møller-Plesset (MP2) correlation energies extrapolated at the complete-basis-set (CBS) limit and CCSD(T) calculations using Dunning's basis sets of various sizes. Note that the percentage of correlation energy already retrieved at the CIPSI variational level is around 60%, a relatively important amount according to the standards of post-HF wavefunction theories for such systems. In table II the Fixed-Node DMC total energies obtained using standard Hartree-Fock nodes and newly proposed CIPSI nodes are reported. For the sake of comparison, we also give the very recent results of Buendia *et al.*<sup>6</sup> that were up to now the lowest variational total energies reported for these atoms. In their study the trial wavefunctions employed are written as the product of a nodeless correlation factor and a so-called model function obtained within the parametrized Optimized Effective Potential (OEP) approximation. The model function determining the nodal structure is written as a linear combination of a few Configuration State Functions (CSFs), mainly to take into account  $4s - 4p$  near-degeneracy effects. For the Cr and Cu atoms with a singly occupied  $4s$  shell the model function is represented by a single CSF, while for the other atoms  $4s^2 3d^n$  and  $4p^2 3d^n$  configurations are mixed. For each type of nodes used, an estimate of the percentage of the correlation energy is also reported. The percentages retrieved by all FN-DMC calculations presented are important and range between 89 and 94%. A first observation is that energies resulting from HF and OEP nodes are of comparable quality, while CIPSI nodes may lead to significantly lower fixed-node energies. The gain in energy with the new nodes is found to decrease almost uniformly with  $Z$ . For the lightest elements (Sc, V and Ti) a maximum gain of about 0.04 a.u is achieved; for the intermediate atoms (Cr to Ni) about 0.02-0.03 a.u. is obtained, while for the two heaviest elements (Cu and Zn) no gain is observed within statistical fluctuations. The fact that CIPSI performs better for lightest elements is not surprising since Hartree-Fock nodes are known to be well-adapted to atoms with spherical symmetry. In the extreme case of the Cu and Zn atoms having a totally filled and spherically symmetric  $3d$  shell, HF and CIPSI nodes give similar results. In the opposite case of light atoms, the CIPSI wavefunctions, that have many more degrees of freedom than the single-configuration HF solution to describe non-symmetrical electronic configurations, lead to much improved results. In table III the correlation energies resulting from our FN-DMC simulations are reported and compared to the recommended values of McCT. As already noted, these latter results have been obtained with a mixed approach including MP2-CBS and CCSD(T) calculations. According to the authors, the errors in these values are estimated to be  $\pm 3\%$ . The relative differences between FN-DMC/[HF nodes] or FN-DMC/[OEP nodes] and the McCT values go from 8 to 11%. Using CIPSI nodes the differences are reduced and range between 6 and 8%. Note that the typical statistical error on these percentages is small and about 0.2%. Although our final values for correlation

Atom	$E_{\text{var}}(\text{CIPSI})$	[CE in %]	$N_{\text{dets}}$
Sc	-760.32556	[66.5%]	11 389
Ti	-849.02624	[66.9%]	14 054
V	-943.53667	[64.9%]	12 441
Cr	-1044.03692	[63.6%]	10 630
Mn	-1150.57902	[63.0%]	11 688
Fe	-1263.21805	[62.5%]	13 171
Co	-1382.24964	[62.8%]	15 949
Ni	-1507.74694	[62.3%]	15 710
Cu	-1639.96605	[63.3%]	48 347
Zn	-1778.82784	[60.5%]	44 206

Table I. Variational energy,  $E_{\text{var}}(\text{CIPSI})$ , of the CIPSI trial wavefunctions  $\Psi_T$  used in FN-DMC, estimated percentages of the correlation energy (CE) recovered, and number of determinants,  $N_{\text{dets}}$ , in the expansions. Energy in hartree.

energies are slightly less accurate than the estimations made by McCT, we would like emphasize and conclude on three important points: i.) In contrast with what has been done by McCT, our correlation energies have been directly computed with a unique highly-correlated electronic structure method. No hybrid scheme mixing results of two different approaches has been employed. To the best of our knowledge, the FN-DMC values presented here are the most accurate (lowest) nonrelativistic total energies ever reported for the 3d transition metal atoms. ii.) As a consequence of the variational property of FN-DMC total energies and, also in contrast with McCT's results, the absolute values of our correlation energies are *exact lower bounds* of the unknown CE's. iii) Finally, in view of the great versatility of FN-DMC/CIPSI, there is no reason why improved lower bounds would not be achieved in the near future, thus leading to benchmark-type results for such atoms.

**Acknowledgments.** AS and MC thank the Agence Nationale pour la Recherche (ANR) for support through Grant No ANR 2011 BS08 004 01. This work has been possible thanks to the computational support of CALMIP (Toulouse) through a Meso-Challenge on their new Eos supercomputer (<http://www.calmip.univ-toulouse.fr>).

## REFERENCES

- <sup>1</sup>B.L. Hammond, W.A. Lester Jr., and P.J. Reynolds. *Monte Carlo Methods in Ab Initio Quantum Chemistry*, volume 1 of *World Scientific Lecture and Course Notes in Chemistry*. 1994.
- <sup>2</sup>W.M.C. Foulkes, L. Mitás, R.G. Needs, and G. Rajagopal. *Rev. Mod. Phys.*, 73:33, 2001.
- <sup>3</sup>M.D. Brown, J.R. Trail, P. Lopez Rios, and R.J. Needs. *J. Chem. Phys.*, 126:224110, 2007.
- <sup>4</sup>A. Sarsa, E.Buendia, F.J. Galvez, and P. Maldonado. *J. Phys. Chem. A*, 112:2074, 2008.
- <sup>5</sup>P. Lopez Rios, P. Seth, N.D. Drummond, and R.J. Needs. *Phys. Rev. E*, 86:036703, 2012.
- <sup>6</sup>E.Buendia, F.J. Galvez, P. Maldonado, and A. Sarsa. *Chem. Phys. Lett.*, 559:12, 2012.
- <sup>7</sup>L. Mitás. *Phys. Rev. A*, 49:4411, 1994.
- <sup>8</sup>L. Wagner and M. Mitás. *Chem. Phys. Lett.*, 370:412, 2003.
- <sup>9</sup>S. Sokolova and A. Luchow. *Chem. Phys. Lett.*, 320:421, 2000.
- <sup>10</sup>A. Ma, N.D. Drummond, M.D. Towler, and R.J. Needs. *Phys. Rev. E*, 71:066704, 2005.
- <sup>11</sup>M. Caffarel, J.P. Daudey, J.L. Heully, and A. Ramirez-Solis. *J. Chem. Phys.*, 123:094102, 2005.
- <sup>12</sup>T. Bouabça, B. Braïda, and M. Caffarel. *J. Chem. Phys.*, 133:044111, 2010.
- <sup>13</sup>K.E. Schmidt and J.W. Moskowitz. *J. Chem. Phys.*, 93:4172, 1990.
- <sup>14</sup>H.J. Flad, M. Caffarel, and A. Savin. *Quantum Monte Carlo calculations with multi-reference trial wave functions*. Recent Advances in Quantum Monte Carlo Methods. World Scientific Publishing, 1997.
- <sup>15</sup>C. Filippi and C.J. Umrigar. *J. Chem. Phys.*, 105:213, 1996.
- <sup>16</sup>B. Braïda, J. Toulouse, M. Caffarel, and C. J. Umrigar. *J. Chem. Phys.*, 134:0184108, 2011.
- <sup>17</sup>A. G. Anderson and W.A. Goddard III. *J. Chem. Phys.*, 132:164110, 2010.
- <sup>18</sup>F. Fracchia, C. Filippi, and C. Amovilli. *J. Chem. Theory Comput.*, 8:1943, 2012.
- <sup>19</sup>A. Monari, A. Scemama, and M. Caffarel. Large-scale quantum monte carlo electronic structure calculations on the egee grid. In Franco Davoli, Marcin Lawenda, Norbert Meyer, Roberto Pugliese, Jan Wglarz, and Sandro Zappatore, editors, *Remote Instrumentation for eScience and Related Aspects*, pages 195–207. Springer New York, 2012. [http://dx.doi.org/10.1007/978-1-4614-0508-5\\_13](http://dx.doi.org/10.1007/978-1-4614-0508-5_13).
- <sup>20</sup>M. Bajdich, L. Mitás, G. Drobný, and L.K. Wagner. *Phys. Rev. Lett.*, 96:240402, 2006.
- <sup>21</sup>M. Casula, C. Attaccalite, and S. Sorella. *J. Chem. Phys.*, 121:7110, 2004.
- <sup>22</sup>P. Lopez Rios, A. Ma, N.D. Drummond, M.D. Towler, and R.J. Needs. *Phys. Rev. E*, 74:066701, 2006.
- <sup>23</sup>C.J. Umrigar, J. Toulouse, C. Filippi, S. Sorella, and R.G. Hennig. *Phys. Rev. Lett.*, 98:110201, 2007.
- <sup>24</sup>E. Giner, A. Scemama, and M. Caffarel. *Can. J. Chem.*, 91:879, 2013.
- <sup>25</sup>E. Giner, A. Scemama, and M. Caffarel. 2014.
- <sup>26</sup>C. F. Bender and E. R. Davidson. *Phys. Rev.*, 183:23, 1969.
- <sup>27</sup>B. Huron, P. Rancurel, and J.P. Malrieu. *J. Chem. Phys.*, 58:5745, 1973.
- <sup>28</sup>R. J. Buenker and S. D. Peyerimhoff. *Theor. Chim. Acta*, 35:33, 1974.
- <sup>29</sup>R. J. Buenker and S. D. Peyerimhoff. *Theor. Chim. Acta*, 39:217, 1975.
- <sup>30</sup>R. J. Buenker, S. D. Peyerimhoff, and W. Butscher. *Mol. Phys.*, 35:771, 1978.
- <sup>31</sup>P. J. Bruna, D. S. Peyerimhoff, and R. J. Buenker. *Chem. Phys. Lett.*, 72:278, 1980.
- <sup>32</sup>R. J. Buenker, S. D. Peyerimhoff, and P. J. Bruna. *Computational Theoretical Organic Chemistry*. Reidel, Dordrecht, 1981.
- <sup>33</sup>S. Evangelisti, J.P. Daudey, and J.P. Malrieu. *Chem. Phys.*, 75:91, 1983.
- <sup>34</sup>R. J. Harrison. *J. Chem. Phys.*, 94:5021, 1991.
- <sup>35</sup>B. L. Hammond, W.A. Lester Jr., and P.J. Reynolds. *Monte Carlo Methods in Ab Initio Quantum Chemistry*, volume 1 of *Lecture and Course Notes in Chemistry*. World Scientific, Singapore, 1994. World Scientific Lecture and Course Notes in Chemistry Vol.1.
- <sup>36</sup>M.D. Towler. *Quantum Monte Carlo, or, how to solve the many-particle Schrödinger equation accurately whilst retaining favourable scaling with system size*. Wiley, 2011.
- <sup>37</sup>M. Caffarel. *Quantum Monte Carlo Methods in Chemistry*. Encyclopedia of Applied and Computational Mathematics. Springer, 2012.



Atom <sup>a</sup>	HF nodes [CE in %]	OEP nodes <sup>b</sup> [CE in %]	CIPSI nodes [CE in %]	FN energy gain with CIPSI nodes <sup>c</sup>
Sc [ $s^2d^1$ ]	-760.5265(13) [89.2(2)%]	-760.5288(6) [89.50(7)%]	-760.5718(16) [94.4(2)%]	-0.0453(21)
Ti [ $s^2d^2$ ]	-849.2405(14) [89.6(2)%]	-849.2492(7) [90.55(7)%]	-849.2841(19) [94.2(2)%]	-0.0436(24)
V [ $s^2d^3$ ]	-943.7843(13) [89.6(1)%]	-943.7882(6) [89.95(6)%]	-943.8234(16) [93.4(2)%]	-0.0391(21)
Cr [ $s^1d^5$ ]	-1044.3292(16) [91.0(2)%]	-1044.3289(6) [90.93(6)%]	-1044.3603(17) [93.9(2)%]	-0.0311(23)
Mn [ $s^2d^5$ ]	-1150.8880(17) [90.4(2)%]	-1150.8897(7) [90.54(6)%]	-1150.9158(20) [92.9(2)%]	-0.0278(26)
Fe [ $s^2d^6$ ]	-1263.5589(19) [90.1(2)%]	-1263.5607(6) [90.26(5)%]	-1263.5868(21) [92.4(2)%]	-0.0279(28)
Co [ $s^2d^7$ ]	-1382.6177(21) [90.5(2)%]	-1382.6216(8) [90.85(6)%]	-1382.6377(24) [92.1(2)%]	-0.0200(32)
Ni [ $s^2d^8$ ]	-1508.1645(23) [91.6(2)%]	-1508.1743(7) [92.27(5)%]	-1508.1901(25) [93.4(2)%]	-0.0256(34)
Cu [ $s^1d^{10}$ ]	-1640.4271(26) [92.4(2)%]	-1640.4266(7) [92.34(4)%]	-1640.4328(29) [92.7(2)%]	-0.0057(39)
Zn [ $s^2d^{10}$ ]	-1779.3371(26) [91.9(2)%]	-1779.3425(8) [92.24(5)%]	-1779.3386(31) [92.0(2)%]	-0.0015(40)

Table II. FN-DMC total energies for the 3d series of transition metal atoms together with the percentage of correlation energy recovered for different nodal structures. Energy in hartree.

<sup>a</sup> Atom given with its electronic configuration, the common argon core [Ar]=(1s<sup>2</sup>2s<sup>2</sup>2p<sup>6</sup>3s<sup>2</sup>3p<sup>6</sup>) being not shown.

<sup>b</sup> Ref. [6].

<sup>c</sup> Difference between FN-DMC energies obtained with HF nodes (column 1) and newly proposed CIPSI nodes (column 3).

Atom	HF nodes	OEP nodes <sup>a</sup>	CIPSI nodes	McCT <sup>b</sup>
Sc	0.7900(13)	0.7923(6)	0.8353(16)	0.8853
Ti	0.8454(14)	0.8541(7)	0.8890(19)	0.9433
V	0.9000(13)	0.9039(6)	0.9390(16)	1.0049
Cr	0.9728(16)	0.9725(6)	1.0039(17)	1.0695
Mn	1.0218(17)	1.0235(7)	1.0495(20)	1.1304
Fe	1.1122(19)	1.1140(6)	1.1401(21)	1.2343
Co	1.2016(21)	1.2055(8)	1.2216(24)	1.3270
Ni	1.3043(23)	1.3141(7)	1.3299(25)	1.4242
Cu	1.4634(26)	1.4629(7)	1.4691(29)	1.5842
Zn	1.4890(26)	1.4944(8)	1.4905(31)	1.6202

Table III. Fixed-Node DMC correlation energies,  $-E_c$ , in hartree using HF and CIPSI nodes. Comparison with the recommended values of McCarthy and Thakkar (McCT).<sup>42</sup>

<sup>a</sup> Ref. [6].

<sup>b</sup> Ref. [42]

<sup>38</sup>E.Buendia, F.J. Galvez, P. Maldonado, and A. Sarsa. *J. Comp. Chem.*, 34:938, 2013.

<sup>39</sup>C. F. Bunge, J.A. Barrientos, and A. Bunge-Vivier. Roothaan-hartree-fock ground-state atomic wave functions: Slater-type orbital expansions and expectation values for  $z=2-54$ . *Atomic data and nuclear data tables*, 53(1):113–162, 1993.

<sup>40</sup>See web site: "Quantum Monte Carlo for Chemistry@Toulouse", <http://qmcchem.ups-tlse.fr>.

<sup>41</sup>Roland Assaraf, Michel Caffarel, and Anatole Khelif. Diffusion monte carlo methods with a fixed number of walkers. *Phys. Rev. E*, 61(4):4566–4575, Apr 2000.

<sup>42</sup>S.P. McCarthy and Thakkar A.J. *J. Chem. Phys.*, 136:054107, 2012.

# D | QUANTUM MONTE CARLO WITH VERY LARGE MULTI- DETERMINANT EXPAN- SIONS

# Quantum Monte Carlo with very large multi-determinant expansions

Anthony Scemama, Thomas Applencourt, Emmanuel Giner, and Michel Caffarel  
*Affiliation not available*

## I. INTRODUCTION

Recently we have proposed to use very large configuration interaction (CI) trial wave functions (near Full-CI) to be able to control the fixed-node errors when computing energy differences in the diffusion Monte Carlo (DMC) framework.[1–4] In the present paper, we give all the implementation details that enable Fixed-Node Diffusion Monte-Carlo (FN-DMC) simulations with up to a million of Slater determinants. At first sight, one would expect that using a million of Slater determinants in a FN-DMC calculation should be a million of times more expensive in terms of computational resources. To address the problem of the explosion of the computational time, Clark *et al* have proposed the *table method*. [5] They obtain up to a  $50\times$  speed-up but still have a linear scaling with the number of excitation operators. More recently, Weerasinghe *et al* proposed a compression algorithm [6] where multiple single excitations with respect to a reference determinant are replaced by one determinant built on auxiliary non-orthonormal molecular orbitals, reducing the total number of determinants. Indeed, they obtain a reduction of the total number of determinants by a factor of  $26\times$ , and have obtained a sublinear scaling of the computational cost with the number of determinants.

We present here a different approach to reduce the computational cost of large multi-configurational expansions, as is implemented in the QMC=Chem program developed in our group. [7] The proposed algorithm scales asymptotically as the square root of the number of determinants.

The  $N_{\text{elec}}$ -electron and  $N_{\text{det}}$ -determinant CI wave function  $\Psi$  is expressed as

$$\Psi(\mathbf{R}) = \sum_{k=1}^{N_{\text{det}}} c_k \mathcal{D}_k(\mathbf{R}) \quad (1)$$

In the spin-free formalism, each Slater determinant  $\mathcal{D}_k(\mathbf{R})$  can be written as a Waller-Hartree double-determinant [8, 9] which is the product of two spin-specific determinants  $D_i^\alpha(\mathbf{R}_\alpha)$  and  $D_j^\beta(\mathbf{R}_\beta)$  made respectively of  $\alpha$  and  $\beta$  spin-orbitals and occupied respectively with  $N_{\text{elec}}^\alpha$  and  $N_{\text{elec}}^\beta$  electrons. The key point is that each determinant  $D_i^\alpha$  or  $D_j^\beta$  appears in multiple determinant products  $\mathcal{D}_k$ , such that the wave function can be expressed in a bilinear form as

$$\Psi(\mathbf{R}) = \sum_{i=1}^{N_{\text{det}}^\alpha} \sum_{j=1}^{N_{\text{det}}^\beta} C_{ij} D_i^\alpha(\mathbf{R}_\alpha) D_j^\beta(\mathbf{R}_\beta) = \mathbf{D}_\alpha^\dagger \mathbf{C} \mathbf{D}_\beta \quad (2)$$

where  $\mathbf{D}_\alpha$  and  $\mathbf{D}_\beta$  are column vectors containing the values of the spin-specific determinants evaluated at the electron positions. In practice, the constant coefficient matrix  $\mathbf{C}$  is sparse as it contains  $N_{\text{det}}$  non-zero entries where  $N_{\text{det}} \ll N_{\text{det}}^\alpha \times N_{\text{det}}^\beta$ . In the Full Configuration Interaction (FCI) limit,  $N_{\text{det}} = N_{\text{det}}^\alpha \times N_{\text{det}}^\beta = (N_{\text{det}}^\alpha)^2$  for closed shell systems.

We first show that using this representation allows the effective scaling of the QMC calculations to be in  $\mathcal{O}(N_{\text{det}}^\alpha + N_{\text{det}}^\beta) = \mathcal{O}(\sqrt{N_{\text{det}}})$ . Next, we give the implementation details we used to make the computations efficient in order to reduce the prefactor of the computational cost. To reduce even more the computational time, we then show how these wave functions can be truncated without affecting significantly the fixed-node error. Numerical experiments on the Chlorine atom illustrate all the presented ideas using wave functions with up to one million of Slater determinants.

## II. ALGORITHM AND IMPLEMENTATION

At every Monte Carlo step, the values of all the  $N_{\text{MO}}$  non-empty molecular orbitals (MOs) are computed at all the electron positions and stored in an  $(N_{\text{elec}}^\alpha + N_{\text{elec}}^\beta) \times N_{\text{MO}}$  array  $\Phi$ . Similarly, the derivatives of the MOs with respect to the electron coordinates (gradients  $\nabla_{x,i}$ ,  $\nabla_{y,i}$ ,  $\nabla_{z,i}$ , and Laplacian  $\Delta_i$ ) are stored in four arrays  $\nabla_x \Phi$ ,  $\nabla_y \Phi$ ,  $\nabla_z \Phi$ , and  $\Delta \Phi$ . Our implementation has already been detailed in reference [10], but let us recall that this step can be computed very efficiently on modern x86 central processing units (CPUs) as it makes an intensive use of vector fused multiply-add (FMA) instructions and has a very low memory footprint. In this section, we describe how the multi-determinant wave function is evaluated, as well as its derivatives (gradients and Laplacian).

### A. Pre-processing

The wave function has to be transformed from the usual representation, the linear combination of determinant products of Eq.(1), to the bilinear form of Eq.(2). This has to be done only once before running the quantum Monte Carlo (QMC) calculation.

Determinants are initially encoded in an of 64-bit integers as described in reference [11]: when the number of MOs is less or equal to 64, one integer encodes the occupation of the  $\alpha$  spinorbitals and another integer encodes the occupation of the  $\beta$  spinorbitals by setting to one the bits corresponding to the positions of occu-

List index	Decimal	Binary	Determinant
1	15	00001111	1234>
2	23	00010111	1235>
3	27	00011011	1245>
4	29	00011101	1345>
5	30	00011110	2345>
6	39	00100111	1236>
7	43	00101011	1246>
8	45	00101101	1346>
...	...	...	...

TABLE I. Ordering of determinants given by Eq. (3).

pied MOs. For instance, the Hartree-Fock determinant for the Chlorine atom (9  $\alpha$ -electrons and 8  $\beta$ -electrons) is encoded as (255,127), which is in binary representation (11111111,1111111), and the doubly excited determinant resulting from  $T_{7 \rightarrow 12}^\alpha T_{7 \rightarrow 12}^\beta$  is (2495,2239) or (10011011111,10001011111). When the system contains more than 64 MOs,  $N_{\text{int}}$  64-bit integers are used for each spin-specific determinant. The initial storage requirement is therefore  $N_{\text{det}} \times (\lfloor N_{\text{MOs}}/64 \rfloor + 1) \times 16$  bytes.

In a first step, we separate the  $\alpha$  and  $\beta$  parts of all the determinant products  $\mathcal{D}_k$  into two distinct lists: a list of  $\alpha$  determinants and a list of  $\beta$  determinants. Each spin-specific list is then treated independently in a similar way as follows.

First, the list is sorted with respect to some arbitrary comparison function. The comparison function used is the comparison of a key  $\omega$  which is the numerical value of the 64-bit integer obtained by accumulating an **xor** operation ( $\oplus$ ) on all the  $N_{\text{int}}$  64-bit integers  $i_n$  constituting the determinant

$$\omega = i_1 \oplus \dots \oplus i_{N_{\text{int}}} - 2^{63} \quad (3)$$

As Fortran doesn't handle unsigned integers, we shift the value by  $-2^{63}$  to get an ordering consistent with the unsigned representation.

This choice of comparison function is motivated by the fact that contiguous determinants are likely to have a small number of differences. Table I gives an example of the ordering with 4 electrons in 8 orbitals. One can remark that the probability of using a single excitation to go from one determinant to the next one in the list is very high. (This property will be used in the section IID)

The sort is performed in a linear time with respect to  $N_{\text{det}}^\alpha$  and  $N_{\text{det}}^\beta$  thanks to the radix sort algorithm.[12] Then, duplicate determinants are filtered out by searching for duplicates among determinants giving the same key  $\omega$ . At this point, we have two spin-specific lists of sorted determinants containing respectively  $N_{\text{det}}^\alpha$  and  $N_{\text{det}}^\beta$  unique determinants.

Now we want to build the matrix  $\mathbf{C}$  directly in sparse coordinate format: with an array of values, an array of

column indices and an array of row indices. Note that we already know that all three arrays can be dimensioned exactly with  $N_{\text{det}}$ .

For each determinant product  $\mathcal{D}_k$ , we compute the key  $\omega$  corresponding to the  $\alpha$  determinant. As the list of unique determinants is sorted, we can use a binary search to find its position  $i$  in the list in logarithmic time. This position is appended to the list of row indices. Similarly, the list of column indices is updated by finding the position  $j$  of the  $\beta$  determinant. To improve the memory access patterns in the next steps, the value  $N_{\text{det}}^\alpha \times (j-1) + i$  is appended to an additional temporary array.

Finally, the additional temporary array is sorted (in linear time with the radix sort), and we apply the corresponding ordering to the three arrays containing the sparse representation of the  $\mathbf{C}$  matrix. Now, the elements of the  $\mathbf{C}$  matrix are ordered such that reading the arrays sequentially corresponds to reading the matrix column by column.

This pre-processing step is not a bottleneck as it scales linearly with the number of determinant products and has to be done only once. For instance, this pre-processing step takes roughly 3 seconds on a single core for a wave function with a million of Slater determinants. On the contrary, the computations described in the next paragraphs need to be performed at every Monte Carlo step.

## B. Calculation of the vectors $\mathbf{D}_\alpha$ and $\mathbf{D}_\beta$

The list of integers corresponding to the indices of the molecular orbitals occupied in the first determinant  $D_1^\alpha$  is decoded from its compressed 64-bit integer representation. This list is used to build the Slater matrix  $\mathbf{S}_1^\alpha$  corresponding to  $D_1^\alpha$  by copying the appropriate  $N_\alpha$  columns of  $\Phi$ . We then evaluate the determinant and the inverse Slater matrix in the usual way: we perform the LU factorization of  $\mathbf{S}_1^\alpha$  using partial pivoting (using the **dgetrf** LAPACK routine[13],  $\mathcal{O}(N_{\text{elec}}^{\alpha 3})$ ). It is now straightforward to obtain the determinant  $D_1^\alpha(\mathbf{R}_\alpha)$ , and the inverse Slater matrix  $(\mathbf{S}_1^\alpha)^{-1}$  is obtained using the **degtri** LAPACK routine in  $\mathcal{O}(N_{\text{elec}}^\alpha)$ . If the dimension of the Slater matrix is smaller than  $6 \times 6$ , one can remark that this  $\mathcal{O}(N_{\text{elec}}^{\alpha 3})$  algorithm will cost more than the naïve  $\mathcal{O}(N_{\text{elec}}^\alpha!)$  algorithm. Moreover, linear algebra packages are optimized for large matrices and usually don't perform well on such small matrices. Therefore, we used a script to generate hard-coded subroutines implementing the naïve algorithm for the calculation of the determinant and the inversion of  $1 \times 1$  to  $5 \times 5$  matrices.

For all the remaining determinants  $\{D_{i>1}^\alpha\}$ , the Sherman-Morrison scheme is used to update the inverse Slater matrix *in place* in  $\mathcal{O}(N_{\text{elec}}^{\alpha 2})$ . Other implementations[5] compare the Slater matrix to a common reference, but here we perform the Sherman-Morrison updates with respect to the previously computed determinant  $D_{i-1}^\alpha$ . The column updates are exe-



cuted sequentially by substituting one column at a time. In the case of a double excitation for instance, a sequence of two updates will be performed. To avoid the propagation of numerical errors, if the absolute value of the ratio of the determinant with the substituted column over the previous determinant is below  $10^{-3}$ , the current column substitution is not realized and stored in a list of failed updates. When all updates have been tried, the list of updates to do is overwritten by the list of failed updates and all the remaining updates are tried again, until the list of failed updates becomes empty. If at one iteration the length of the list of failed updates has the same non-zero length as in the previous iteration of the sequence, the Sherman-Morrison updates are cancelled and the determinant is re-computed with the  $\mathcal{O}(N_{\text{elec}}^{\alpha^3})$  algorithm.

The Sherman-Morrison updates are hot spots in large multi-determinant calculations, so some particular effort was invested in their computational efficiency. One can first remark that for Slater matrices with dimensions  $2 \times 2$  and  $3 \times 3$ , the hard-coded naïve algorithm for the computation of the inverse Slater matrix from scratch is more efficient than doing the Sherman-Morrison update, so this version is used instead. Secondly, if  $N_{\text{elec}}^{\alpha}$  is small (typically less than 20), a general routine is very likely to be inefficient: in loops written as

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

the compiler is not aware of the number of loop cycles  $n$  at compile time, so it will generate code to try to vectorize the loops (peeling loop, scalar loop, vector loop and tail loop) and test which branch to choose at execution time. If the loop count is low, the overhead dramatically affects the performance. For all matrix sizes in the  $[4 \times 4 : 20 \times 20]$  range, we have generated size-specific subroutines from a template where the loop counts and matrix dimensions are hard-coded, in such a way to force the compiler to generate 100% vectorized loops. When needed, the tail loops are written explicitly. The binary code produced by the compiler was validated with the MAQAO[14] static analysis tool by checking that vector fused-multiply-add (FMA) instructions were produced extensively in the innermost loops of the Sherman-Morrison updates. For larger matrix sizes, a general subroutine is used. In all the different versions, we use padding in the matrices to enforce the proper memory alignment of all the columns of the matrices to enable vectorization without the peeling loop.

The scaling of this step is  $\mathcal{O}(N_{\text{elec}}^{\alpha^2} \times N_{\text{det}}^{\alpha})$ , which becomes  $\mathcal{O}(N_{\text{det}}^{\alpha})$  for large multi-determinant expansions.

### C. Calculation of the gradients and Laplacian

The bilinear expression of the wave function in Eq.(2) yields the following expressions for the derivatives:

$$\nabla_{x,i}\Psi = (\nabla_{x,i}\mathbf{D}_{\alpha})^{\dagger}\mathbf{C}\mathbf{D}_{\beta} + \mathbf{D}_{\alpha}^{\dagger}\mathbf{C}(\nabla_{x,i}\mathbf{D}_{\beta}) \quad (4)$$

$$\begin{aligned} \Delta_i\Psi = & (\Delta_i\mathbf{D}_{\alpha})^{\dagger}\mathbf{C}\mathbf{D}_{\beta} + \mathbf{D}_{\alpha}^{\dagger}\mathbf{C}(\Delta_i\mathbf{D}_{\beta}) + 2[ \\ & (\nabla_{x,i}\mathbf{D}_{\alpha})^{\dagger}\mathbf{C}(\nabla_{x,i}\mathbf{D}_{\beta}) + \\ & (\nabla_{y,i}\mathbf{D}_{\alpha})^{\dagger}\mathbf{C}(\nabla_{y,i}\mathbf{D}_{\beta}) + \\ & (\nabla_{z,i}\mathbf{D}_{\alpha})^{\dagger}\mathbf{C}(\nabla_{z,i}\mathbf{D}_{\beta})] \end{aligned} \quad (5)$$

In the expression of the Laplacian of the wave function (Eq.(5)), the gradient terms  $\nabla_i\mathbf{D}_{\alpha}$  vanish when  $i$  is a  $\beta$  electron. Similarly, the terms  $\nabla_i\mathbf{D}_{\beta}$  vanish when  $i$  is an  $\alpha$  electron. As a consequence, the cross-terms involving both the gradients  $\nabla_i\mathbf{D}_{\alpha}$  and  $\nabla_i\mathbf{D}_{\beta}$  are always zero, and the  $3N_{\text{elec}}$  components of the of the gradient and of the Laplacian can be computed using the same instructions with different input data. Hence we define  $\tilde{\nabla}_i$  as a four-dimensional vector  $[\nabla_x, \nabla_y, \nabla_z, \Delta]$ , and one only needs to implement

$$\tilde{\nabla}_i\Psi = (\tilde{\nabla}_i\mathbf{D}_{\alpha})^{\dagger}\mathbf{C}\mathbf{D}_{\beta} + \mathbf{D}_{\alpha}^{\dagger}\mathbf{C}(\tilde{\nabla}_i\mathbf{D}_{\beta}) \quad (6)$$

The gradients and the Laplacian of the wave function are computed together using Eq. 6 in an array of dimension  $4 \times N_{\text{elec}}$ , using the arrays  $\tilde{\nabla}\mathbf{D}^{\alpha}$  and  $\tilde{\nabla}\mathbf{D}^{\beta}$ , dimensioned respectively as  $4 \times N_{\text{elec}}^{\alpha} \times N_{\text{det}}^{\alpha}$  and  $4 \times N_{\text{elec}}^{\beta} \times N_{\text{det}}^{\beta}$ . The computation of  $\tilde{\nabla}\Psi$  can be performed in parallel on a single CPU core using Single Instruction Multiple Data (SIMD) vector instructions. Indeed, modern x86 CPUs can use vector operations on 256- or 512 bit-wide vectors, which correspond to 4 or 8 double precision elements, if the arrays are properly aligned in memory. Hence we aligned the arrays on 512-bit boundaries using compiler directives.

The  $\tilde{\nabla}\mathbf{D}_j^{\alpha}$  are computed using the array  $\tilde{\nabla}\Phi$  and the inverse Slater matrix:

$$\tilde{\nabla}_i\mathbf{D}_j^{\alpha} = \sum_k S_{ik}^{\alpha-1}\tilde{\nabla}\Phi_{ki}^{\alpha} \quad (7)$$

The innermost loop is the loop over the 4 components (gradients and Laplacian) of  $\tilde{\nabla}$ , so we unroll twice the loop over  $k$  to enable vector instructions also on the AVX-512 micro-architecture which requires 8 double precision elements.

The scaling of this step is  $\mathcal{O}(N_{\text{elec}}^{\alpha^2} \times N_{\text{det}}^{\alpha})$ , which is  $\mathcal{O}(N_{\text{det}}^{\alpha})$  for large multi-determinant expansions.

The calculation of the derivatives of the wave function is performed using two dense matrix-vector products:  $(\tilde{\nabla}\mathbf{D}_{\alpha})^{\dagger} \cdot (\mathbf{C}\mathbf{D}_{\beta})$  and  $(\mathbf{D}_{\alpha}^{\dagger}\mathbf{C}) \cdot (\tilde{\nabla}\mathbf{D}_{\beta})$ , as detailed in the next subsection.

#### D. Computation of the intermediate vectors and $\Psi$

We can identify that the two matrix-vector products  $\mathbf{D}_\alpha^\dagger \mathbf{C}$  and  $\mathbf{C} \mathbf{D}_\beta$  can be performed once, and the resulting vectors are used for the computation of  $\Psi$  and  $\bar{\nabla} \Psi$ . As this step consists in two sparse matrix / dense vector product, it has inevitably a low arithmetic intensity (small number of floating point operations per data loaded or stored) and the execution speed is limited by data access. It is therefore critical to optimize the data movement from the main memory to the CPU cores for this step. As the same matrix  $\mathbf{C}$  is used in both products, the two products can be computed simultaneously:

```
do k=1,det_num
  i = C_rows(k)
  j = C_columns(k)
  Da_C(j) = Da_C(j) + C(k)*Da(i)
  C_Db(i) = C_Db(i) + C(k)*Db(j)
enddo
```

In this way, the three arrays corresponding to the  $\mathbf{C}$  matrix are streamed from the main memory through the CPU registers only once. The data relative to the matrix  $\mathbf{C}$  can be moved from the main memory with a very low latency as the hardware prefetchers of x86 CPUs are very efficient on unit stride access patterns. Also, the ordering of the arrays of the  $\mathbf{C}$  matrix in the pre-processing phase (section II A) maximizes the probability of  $\text{Da\_C}(j)$  and  $\text{Db}(j)$  to be in already in the CPU registers as the column index  $j$  is very likely to be constant from one iteration to the next.  $\text{Da}(i)$  and  $\text{C\_Db}(i)$  are likely to be in a low-level cache (L1 or L2) as the arrays are always small, dimensioned by  $N_{\text{det}}^\alpha$  and  $N_{\text{det}}^\beta$  (typically 25 KiB for a wave function with a million of Slater determinants).

The asymptotic computational cost of this step is  $\mathcal{O}(N_{\text{det}})$ , but the prefactor is so small that we observed that it is never the time-limiting step, even in the regime where  $\Psi$  has a million of determinant products.

We chose the convention that the number of  $\alpha$  electrons is greater or equal to the number of  $\beta$  electrons. As a consequence the general case is that  $N_{\text{det}}^\alpha \geq N_{\text{det}}^\beta$  so we choose to compute the value of  $\Psi$  using the dot product  $(\mathbf{D}_\alpha \mathbf{C}) \cdot \mathbf{D}_\beta$  as it involves only  $N_{\text{det}}^\beta$  operations.

#### E. Truncation of the expansion

It is common practice in QMC simulations to reduce the number of determinants by truncating the CI expansion of Eq.(1) based on the absolute value of the coefficients, or by keeping the smallest number of determinants contributing to a given percentage of the norm of the wave function. Truncating coefficients of Configuration State Functions (CSFs) is an improvement as it doesn't break the property of the wave function to be an eigenstate of  $S^2$ .

The truncation of the wave function is motivated by the reduction of the computational cost of the simulation. Some determinant products, when removed, don't contribute to the reduction of the cost if both the  $\alpha$  and  $\beta$  spin-specific determinants are used in other determinant products. In other words, truncating the wave function based on the absolute value of the CI coefficients is the same as increasing the sparsity of the  $\mathbf{C}$  matrix until lines and columns of  $\mathbf{C}$  become entirely filled with zeros. Only at this point some spin-specific determinants can be removed and computational time reduces. The collateral damage of strategy is the useless cancelling of matrix elements of  $\mathbf{C}$  decreasing the quality of the wave function.

We propose to truncate the wave function by trying to remove independently  $\alpha$  and  $\beta$  determinants. To do this, we decompose the norm of the wave function as

$$\mathcal{N} = \sum_{ij} C_{ij}^2 = \sum_i \mathcal{N}_i^\alpha = \sum_j \mathcal{N}_j^\beta \quad (8)$$

where  $\mathcal{N}_i^\alpha = \sum_j C_{ij}^2$  and  $\mathcal{N}_j^\beta = \sum_i C_{ij}^2$  are the contributions to the norm of determinants  $D_i^\alpha$  and  $D_j^\beta$ . We approximate the wave function by removing spin-specific determinants whose contribution to the norm are less than a threshold given in input by the user.

### III. RESULTS

The Chlorine atom (17 electrons) was chosen as a benchmark in the cc-pVDZ and cc-pVTZ basis sets[15] expressed in cartesian coordinates (respectively 19 and 39 molecular orbitals). Timings were measured as the total CPU time needed for one walker to realize one Monte Carlo step. This includes the calculation of the wave function, the drift vector and the local energy.

The benchmarks were run on a single-socket desktop computer, with an Intel® Xeon® E3-1271 v3 quad-core processor at 3.60 GHz with the Turbo feature disabled. QMC=Chem was compiled with the Intel® Fortran Compiler version 15.0.2 with options to generate code optimized for the AVX2 micro-architecture, and linked with the Intel® Math Kernel Library (MKL).

The calculation of the FN-DMC energies were performed using 800 cores on the Curie machine (TGCC/CEA/Genci). The total computational time we used to generate Figure 2 was 182 500 CPU hours.

#### A. Speedup due to the bilinear form

As a first numerical experiment, CIPSI wave functions in the Full-CI space were prepared with our code (Quantum Package[16]) from one to one million determinants. Let us recall that a CIPSI wave function with  $N_{\text{det}}$  determinants has an energy almost equal to the lowest possible energy one can obtain with any combination of  $N_{\text{det}}$  determinants in the same basis set.

$N_{\text{det}}$	$N_{\text{det}}^{\alpha}$	$N_{\text{det}}^{\beta}$	$N_{\text{MOs}}$	RAM (MiB)	CPU time (ms)
1	1	1	9	6.12	0.0179
10	7	7	16	6.20	0.0470
100	40	32	18	6.24	0.1765
1 000	250	186	19	6.42	0.9932
10 000	1 143	748	19	7.35	4.5962
100 000	5 441	3 756	19	13.20	20.5972
1 000 000	21 068	14 516	19	45.84	83.1611

TABLE II. Number of determinants ( $N_{\text{det}}$ ,  $N_{\text{det}}^{\alpha}$  and  $N_{\text{det}}^{\beta}$ ), number of occupied molecular orbitals ( $N_{\text{MOs}}$ ), amount of memory per core and CPU time per per core per Monte Carlo step (without Sherman-Morrison updates).

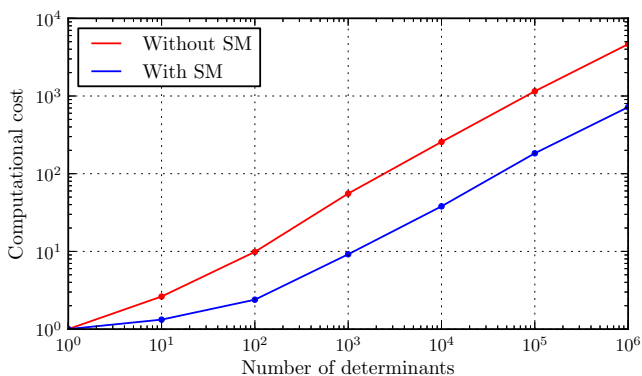


FIG. 1. Computational cost with respect to the number of determinants, normalized to the cost of a single-determinant calculation. Results are given with and without Sherman-Morrison updates.

To illustrate the fact that the bilinear representation in Eq.2 gives rise to an  $\mathcal{O}(\sqrt{N_{\text{det}}})$  asymptotic scaling, we have measured the CPU time needed to realize one Monte Carlo step if all the determinants are computed with the cubic algorithm (without Sherman-Morrison updates). The results are given in table III, as well as the number of  $\alpha$  and  $\beta$  unique determinants, the number of occupied molecular orbitals, and the amount of RAM needed per CPU core. The computational cost compared to a single-determinant calculation is given in figure 1.

From the data of table II, one can observe as expected a CPU time scaling perfectly linearly with  $N_{\text{det}}^{\alpha} + N_{\text{det}}^{\beta}$  when the computation of the determinants becomes the limiting step (100 determinants and above). The scaling obtained with respect to the total number of determinant products is  $N_{\text{det}}^{0.63}$ . This scaling is slightly higher than the expected  $N_{\text{det}}^{0.5}$  due to the sparsity of the  $\mathbf{C}$  matrix: among all possible determinant products, many have a zero coefficient and those are not counted in  $N_{\text{det}}$ .

$N_{\text{det}}$	$N_{\text{det}}^{\alpha}$	$N_{\text{det}}^{\beta}$	$N_{\text{inv}}$	$N_{\text{subst}}$	CPU time(ms)
1	1	1	2.000	0.000	0.0179
10	7	7	2.011	28.981	0.0237
100	40	32	2.058	129.924	0.0428
1 000	250	186	2.565	925.064	0.1643
10 000	1 143	748	5.087	4 282.750	0.6808
100 000	5 441	3 756	17.737	19 049.286	3.2794
1 000 000	21 068	14 516	54.733	63 324.510	12.8688

TABLE III. Number of determinants ( $N_{\text{det}}$ ,  $N_{\text{det}}^{\alpha}$  and  $N_{\text{det}}^{\beta}$ ), average number of matrix inversions in  $\mathcal{O}(N^3)$  ( $N_{\text{inv}}$ ), average number of Sherman-Morrison column substitutions ( $N_{\text{subst}}$ ) and CPU time per core per Monte Carlo step.

Excitation degree	Number of $\mathcal{D}_k$	Number of $\mathcal{D}_i^{\alpha}$	Number of $\mathcal{D}_j^{\beta}$
0	1	1	1
1	38	90	88
2	2 177	1603	1520
3	43 729	7811	6507
4	308 045	8581	5071
5	351 182	2090	579
6	291 481	77	0
7	3 067	0	0
8	280	0	0

TABLE IV. In the 1 000 000-determinant wave function, the number of determinants resulting from excitation operators of degree 0 to 8 applied on the Hartree-Fock reference ( $\mathcal{D}_1$ ,  $\mathcal{D}_1^{\alpha}$  or  $\mathcal{D}_1^{\beta}$ ).

## B. Speedup due to Sherman-Morrison updates

Now, including the Sherman-Morrison updates, a speedup of 6 – 7 $\times$  is obtained. According to the documentation of the MKL library, the full matrix inversion (`dgetrf` followed by `dgetri`) uses approximately  $2n$  floating-point (FP) operations for an  $n \times n$  matrix, whereas one Sherman-Morrison column substitution in our implementation uses  $5n^2 + 2n + 3$  FP operations. From the data of table III, the average number of Sherman-Morrison updates per determinant ranges between 1.7 and 2.5. From these results, one can conclude that our implementation of column substitutions has an efficiency higher than the efficiency of the matrix inversion using the MKL library for such small matrices: for an  $9 \times 9$  matrix, two column substitution involve 1.71 times less FP operations than the full matrix inversion, which is four times less than the speedup we measure.

The average number of substitutions is lower than what one would have obtained with a fixed reference determinant. For instance, if the Hartree-Fock  $\alpha$  and  $\beta$  determinants had been taken as a fixed reference and assuming all substitutions were successful (with a determinant ratio greater than  $10^{-3}$  in absolute value), the average number of Sherman-Morrison substitutions would

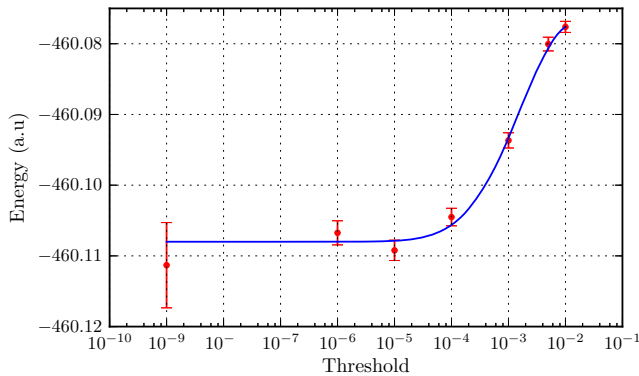


FIG. 2. Total FN-DMC energy of the Chlorine atom with truncated near-Full-CI/cc-pVTZ wave functions. Spin-specific determinants with a contribution to the norm less than the threshold are removed.

have been equal to 3.46 according to the data of table IV where we have measured an average of 1.78 for the same wave function with our implementation.

The average number of matrix inversions per step is at least two since one determinant has to be computed for each spin. Then, the probability of using the  $\mathcal{O}(N^3)$  algorithm instead of the Sherman-Morrison updates to reduce the propagation of numerical errors stays very low below 0.17 %. We have checked that for a given set of electron coordinates the local energies computed with and without the Sherman-Morrison updates differ by no more than  $2 \cdot 10^{-5}$  atomic units on all the wave functions. These data confirm that the numerical stability of the Sherman-Morrison updates can be controlled without affecting significantly the computational time.

### C. Speedup due to the truncation of the wave function

We have generated a wave function for the Chlorine atom in the cc-pVTZ basis set with one million of determinants. The determinants are generated with the CIPSI algorithm in the FCI space with 2 frozen electrons in the 1s orbital (2 doubly occupied MOs and 15 active electrons in 37 MOs). We have truncated this wave function using the truncation based on the absolute value of the CI coefficients, and using the contribution of spin-specific determinants to the norm of the wave function. The wave functions obtained after truncation as well as the computational time for one Monte Carlo step are detailed in table V.

Table ?? gives the energies of the truncated wave functions, and the CPU time needed to run the calculations. On figure 2, we can see that the FN-DMC energy is converged within the error bars with a threshold of  $10^{-6}$  on the contributions  $\mathcal{N}_i^\alpha$  and  $\mathcal{N}_j^\beta$  to the norm of the wave function.

Figure 3 compares the computational cost obtained

Threshold $\epsilon$	$N_{\text{det}}$	$N_{\text{det}}^\alpha$	$N_{\text{det}}^\beta$	CPU time(ms)
$ c_k  > \epsilon$				
0.96	1	1	1	0.02450
0.0404	10	8	8	0.03080
0.0103	100	53	35	0.05888
$10^{-2}$	110	54	37	0.06004
$2.02 \cdot 10^{-2}$	1000	254	168	0.1745
$10^{-3}$	2003	496	335	0.3364
$2.53 \cdot 10^{-3}$	10000	1700	994	0.9732
$10^{-4}$	30198	3668	1853	1.920
$3.55 \cdot 10^{-5}$	100000	9256	4524	4.912
$10^{-5}$	348718	24758	12511	14.20
$10^{-6}$	993811	52291	26775	31.11
0.0	1000000	52433	26833	31.92

Threshold $\epsilon$	$N_{\text{det}}$	$N_{\text{det}}^\alpha$	$N_{\text{det}}^\beta$	CPU time(ms)
$\mathcal{N}_i^\alpha > \epsilon ; \mathcal{N}_i^\beta > \epsilon$				
$10^{-2}$	1	1	1	0.02450
$5 \cdot 10^{-3}$	3	3	2	0.02688
$10^{-3}$	86	21	17	0.03899
$5 \cdot 10^{-4}$	214	29	28	0.04636
$10^{-4}$	1361	93	74	0.08808
$5 \cdot 10^{-5}$	2424	120	89	0.1054
$10^{-5}$	9485	234	166	0.1855
$10^{-6}$	54016	772	523	0.5960
$10^{-7}$	207995	2279	1389	1.740
$10^{-8}$	459069	5797	3291	4.196
$10^{-9}$	748835	14456	8054	9.724
$10^{-10}$	926299	30320	16571	19.18
0	1000000	52433	26833	31.92

TABLE V. Number of determinants, number of spin-specific determinants and computational cost as a function of the truncation threshold with two different truncation approaches.

Threshold	Energy (a.u.)	CPU time (hours)
$10^{-2}$	-460.0776(08)	15 820
$5 \cdot 10^{-3}$	-460.0800(10)	14 521
$10^{-3}$	-460.0937(11)	14 522
$10^{-4}$	-460.1045(12)	19 109
$10^{-5}$	-460.1092(14)	22 344
$10^{-6}$	-460.1067(17)	47 679
$10^{-9}$	-460.1113(60)	48 530

with the two truncation approaches. For a fixed computational cost, our truncation strategy keeps a much larger number of determinant products than the standard truncation scheme. Removing spin-specific determinants which contribute to less than  $10^{-5}$  of the norm can make the calculation of a wave function with 9 485 determinants cost only  $7.6 \times$  more than a single determinant calculation with a FN-DMC energy equivalent to the energy of the full wave function with a million of

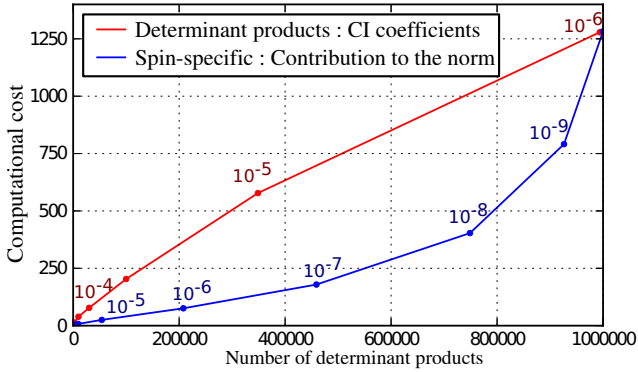


FIG. 3. Computational cost with respect to the number of determinants, normalized to the cost of a single-determinant calculation. The truncation is applied to the absolute value of the CI coefficients or to the contribution to the norm of the spin-specific determinants. The value of the truncation threshold is given on the figure next to the corresponding points.

Slater determinants.

#### IV. CONCLUSIONS

We have shown how pre-processing the wave function could give a scaling in  $\mathcal{O}(\sqrt{N_{\text{det}}})$  for QMC calculations.

The cost of the pre-processing is negligible (3 seconds for one million of determinants) and the scaling of this step is linear with the number of determinants in the wave function. Then, we have shown that the Sherman-Morrison formulas are implemented in QMC=Chem with an efficiency better than the efficiency of the matrix inversion routines present in the Intel MKL library for small matrices. We have also shown that performing the Sherman-Morrison updates in-place with respect to the previous determinant in the list could be better than comparing the determinants to a common reference if the list respects a particular ordering. Finally, we have shown that using a better truncation scheme of the wave function allows us to obtain a FN-DMC energy of the Chlorine atom compatible with the energy we obtain with one million of determinants for a computational cost only  $7.6\times$  larger than the single determinant FN-DMC calculation.

To improve further the efficiency of QMC calculations for large systems, the table method of Clark *et al* can be applied in addition to the proposed algorithm. This implies to decompose the excitation operators as products of spin-specific excitation operators such that the table method would be applied independently for each spin.

*Acknowledgments.*

AS and MC thank the Agence Nationale pour la Recherche (ANR) for support through Grant No ANR 2011 BS08 004 01. This work has been possible thanks to the computational support of CALMIP (Toulouse), and GENCI projects x2015067347 and x2015081738.

(<http://www.calmip.univ-toulouse.fr>).

- 
- [1] E. Giner, A. Scemama, and M. Caffarel, J. Chem. Phys. **142**, 044115 (2015).
  - [2] A. Scemama, T. Applencourt, E. Giner, and M. Caffarel, J. Chem. Phys. **141**, 244110 (2014).
  - [3] E. Giner, A. Scemama, and M. Caffarel, Canadian Journal of Chemistry **91**, 879 (2013).
  - [4] M. Caffarel, E. Giner, A. Scemama, and A. Ramírez-Solís, J. Chem. Theory Comput. **10**, 5286 (2014).
  - [5] B. K. Clark, M. A. Morales, J. McMinis, J. Kim, and G. E. Scuseria, J. Chem. Phys. **135**, 244105 (2011).
  - [6] G. L. Weerasinghe, P. L. Ríos, and R. J. Needs, Physical Review E **89** (2014), 10.1103/physreve.89.023304.
  - [7] A. Scemama, M. Caffarel, E. Oseret, and W. Jalby, in *High Performance Computing for Computational Science - VECPAR 2012* (Springer Science Business Media, 2013) pp. 118–127.
  - [8] I. Waller and D. R. Hartree, Proceedings of the Royal Society A: Mathematical Physical and Engineering Sciences **124**, 119 (1929).
  - [9] R. Pauncz, Int. J. Quantum Chem. **35**, 717 (1989).
  - [10] A. Scemama, M. Caffarel, E. Oseret, and W. Jalby, J. Comput. Chem. **34**, 938 (2013).
  - [11] A. Scemama and E. Giner, ArXiv e-prints (2013), arXiv:1311.6244 [physics.comp-ph].
  - [12] A. Andersson, T. Hagerup, S. Nilsson, and R. Raman, in *Proceedings of the twenty-seventh annual ACM symposium on Theory of computing - STOC '95* (ACM Press, 1995).
  - [13] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen, *LAPACK Users' Guide*, 3rd ed. (Society for Industrial and Applied Mathematics, Philadelphia, PA, 1999).
  - [14] L. Djoudi, D. Barthou, P. Carribault, C. Lemuet, J.-T. Acquaviva, and W. Jalby, in *Workshop on EPIC Architectures and Compiler Technology San Jose, California, United-States* (2005).
  - [15] D. E. Woon and T. H. Dunning, J. Chem. Phys. **98**, 1358 (1993).
  - [16] A. Scemama, E. Giner, T. Applencourt, and M. Caffarel, “Quantum Package,” [https://github.com/LCPQ/quantum\\_package](https://github.com/LCPQ/quantum_package) (2014), [Online; accessed 20-July-2015].

## BIBLIOGRAPHIE

1. AMDAHL, G. M. *Validity of the single processor approach to achieving large scale computing capabilities in Proceedings of the April 18-20, 1967, spring joint computer conference on - AFIPS '67 (Spring)* (ACM Press, 1967). doi :[10 . 1145 / 1465482 . 1465560](https://doi.org/10.1145/1465482.1465560). <<http://dx.doi.org/10.1145/1465482.1465560>>.
2. ANDERSON, A. G. & GODDARD III, W. A. Generalized valence bond wave functions in quantum Monte Carlo. *The Journal of Chemical Physics* **132**, 164110 (2010).
3. ANDERSSON, A., HAGERUP, T., NILSSON, S. & RAMAN, R. *Sorting in linear time ? in Proceedings of the twenty-seventh annual ACM symposium on Theory of computing - STOC '95* (ACM Press, 1995). doi :[10 . 1145 / 225058 . 225173](https://doi.org/10.1145/225058.225173). <<http://dx.doi.org/10.1145/225058.225173>>.
4. ANGELI, C. & CIMIRAGLIA, R. Multireference perturbation configuration interaction v. third-order energy contributions in the møller-plesset and epstein-nesbet partitions. *Theoretical Chemistry Accounts* **107**, 313–317 (2002).
5. ANGELI, C., CIMIRAGLIA, R., PERSICO, M. & TONIOLO, A. Multireference perturbation CI I. Extrapolation procedures with CAS or selected zero-order spaces. *Theoretical Chemistry Accounts* **98**, 57–63 (1997).
6. APPLENCOURT, T. *A G2 TestSet CLI* <[http://github.com/TApplencourt/G2\\_TestSet\\_CLI](http://github.com/TApplencourt/G2_TestSet_CLI)> (2015).
7. APPLENCOURT, T. *SQGit : A workaround for binary support in git* <<https://gist.github.com/TApplencourt/de59dbc33a4bb49f17ab>> (2015).
8. APPLENCOURT, T. *The local EMSL Basis set* <[https://github.com/TApplencourt/EMSL\\_Basis\\_Set\\_Exchange\\_Local](https://github.com/TApplencourt/EMSL_Basis_Set_Exchange_Local)> (2015).
9. APPLENCOURT, T. & SCEMAMA, A. *Cache Compile for fortran90* <<https://gist.github.com/TApplencourt/56f135470bf79ba4f694>> (2015).
10. APPLENCOURT, T. & SCEMAMA, A. *Serveur/Client Cloud Challenge* <[https://github.com/TApplencourt/CloudChallenge%5C\\_FranceGrilles.git](https://github.com/TApplencourt/CloudChallenge%5C_FranceGrilles.git)> (2015).
11. ASIMOV, I. The Endochronic Properties of Resublimated Thiotimoline. *Astounding Science Fiction* **41**. <<http://danm.ucsc.edu/~phoenix/danm203/thiotimoline.pdf>> (mar. 1948).



12. ASSARAF, R. & CAFFAREL, M. Zero-Variance Principle for Monte Carlo Algorithms. *Phys. Rev. Lett.* **83**, 4682–4685 (déc. 1999).
13. ASSARAF, R. & CAFFAREL, M. Zero-variance zero-bias principle for observables in quantum Monte Carlo : Application to forces. *The Journal of Chemical Physics* **119**, 10536 (2003).
14. BAJDICH, M., MITAS, L., WAGNER, L. K. & SCHMIDT, K. E. Pfaffian pairing and backflow wavefunctions for electronic structure quantum Monte Carlo methods. *Phys. Rev. B* **77**. doi :[10.1103/physrevb.77.115112](https://doi.org/10.1103/PhysRevB.77.115112). <<http://dx.doi.org/10.1103/PhysRevB.77.115112>> (mar. 2008).
15. BANKER, R. D., DAVIS, G. B. & SLAUGHTER, S. A. Software development practices, software complexity, and software maintenance performance : A field study. *Management science* **44**, 433–450 (1998).
16. BENDER, C. F. & DAVIDSON, E. R. Studies in configuration interaction : The first-row diatomic hydrides. *Physical Review* **183**, 23 (1969).
17. BODNAR, J. *SQLite Python tutorial* <<http://zetcode.com/db/sqlitepythontutorial/>> (2015).
18. BOOTH, G. H., GRÜNEIS, A., KRESSE, G. & ALAVI, A. Towards an exact description of electronic wavefunctions in real solids. *Nature* **493**, 365–370 (déc. 2012).
19. BOOTH, G. H., THOM, A. J. W. & ALAVI, A. Fermion Monte Carlo without fixed nodes : A game of life, death, and annihilation in Slater determinant space. *The Journal of Chemical Physics* **131**, 054106 (2009).
20. BOUABÇA, T., BRAÏDA, B. & CAFFAREL, M. Multi-Jastrow trial wavefunctions for electronic structure calculations with quantum Monte Carlo. *The Journal of Chemical Physics* **133**, 044111 (2010).
21. BOX, G. E. & MULLER, M. E. A note on the generation of random normal deviates. *The annals of mathematical statistics*, 610–611 (1958).
22. BRAÏDA, B., TOULOUSE, J., CAFFAREL, M. & UMRIGAR, C. J. Quantum Monte Carlo with Jastrow-valence-bond wave functions. *The Journal of Chemical Physics* **134**, 084108 (2011).
23. BUENDÍA, E., GÁLVEZ, F., MALDONADO, P. & SARSA, A. Quantum Monte Carlo ionization potential and electron affinity for transition metal atoms. *Chemical Physics Letters* **559**, 12–17 (fév. 2013).
24. BUENKER, R. J. & PEYERIMHOFF, S. D. Energy extrapolation in CI calculations. *Theoretica chimica acta* **39**, 217–228 (1975).

25. BUENKER, R. J. & PEYERIMHOFF, S. D. Individualized configuration selection in CI calculations with subsequent energy extrapolation. *Theoretica chimica acta* **35**, 33–58 (1974).
26. BUNGE, A. V. & ESQUIVEL, R. O. Simple correlated wave functions for accurate electron densities : An application to neon. *Physical Review A* **34**, 853–859 (août 1986).
27. BUNGE, C., BARRIENTOS, J. & BUNGE, A. Roothaan-Hartree-Fock Ground-State Atomic Wave Functions : Slater-Type Orbital Expansions and Expectation Values for  $Z = 2-54$ . *Atomic Data and Nuclear Data Tables* **53**, 113–162 (jan. 1993).
28. BURKATZKI, M., FILIPPI, C. & DOLG, M. Energy-consistent pseudopotentials for quantum Monte Carlo calculations. *The Journal of Chemical Physics* **126**, 234105 (juin 2007).
29. BURKATZKI, M., FILIPPI, C. & DOLG, M. Energy-consistent small-core pseudopotentials for 3d-transition metals adapted to quantum Monte Carlo calculations. *The Journal of Chemical Physics* **129**, 164115 (2008).
30. CAFFAREL, M. *Quantum Monte Carlo Methods in Chemistry* (éd. ENGQUIST, B.) (Springer, 2012).
31. CAMUS, A. *Le mythe de Sisyphe : essai sur l'absurde* ISBN : 2070322882 (Gallimard, Paris, 1985).
32. CASULA, M., ATTACALITE, C. & SORELLA, S. Correlated geminal wave function for molecules : An efficient resonating valence bond approach. *The Journal of Chemical Physics* **121**, 7110 (2004).
33. CHAMPIN, P.-A. & CORDIER, A. *Introduction à GIT* <<http://liris.cnrs.fr/~pchampin/enseignement/intro-git/>> (2015).
34. CHROMIUM PROJET. *Gyp can Generate Your Projects* <<https://www.openhub.net/p/gyp/>> (2015).
35. CIMIRAGLIA, R. & PERSICO, M. Recent advances in multireference second order perturbation ci : The cipsi method revisited. *Journal of computational chemistry* **8**, 39–47 (1987).
36. CLAVERIE, P., DINER, S. & MALRIEU, J.-P. The use of perturbation methods for the study of the effects of configuration interaction. I. Choice of the zeroth-order Hamiltonian. *Int. J. Quantum. Chem* **1**, 751–767 (1967).
37. CLELAND, D. M., BOOTH, G. H. & ALAVI, A. A study of electron affinities using the initiator approach to full configuration interaction quantum Monte Carlo. *The Journal of Chemical Physics* **134**, 024112 (2011).



38. CLELAND, D., BOOTH, G. H. & ALAVI, A. Communications : Survival of the fittest : Accelerating convergence in full configuration-interaction quantum Monte Carlo. *The Journal of Chemical Physics* **132**, 041103 (2010).
39. CLELAND, D., BOOTH, G. H., OVERY, C. & ALAVI, A. Taming the First-Row Diatomics : A Full Configuration Interaction Quantum Monte Carlo Study. *J. Chem. Theory Comput.* **8**, 4138–4152 (nov. 2012).
40. CONDON, E. U. The Theory of Complex Spectra. *Phys. Rev.* **36**, 1121–1133 (7 oct. 1930).
41. CONIFER SYSTEMS LLC. What's Wrong With GNU make ? <<http://www.conifersystems.com/whitepapers/gnu-make/>> (2015).
42. CURTISS, L. A., RAGHAVACHARI, K., TRUCKS, G. W. & POPLE, J. A. Gaussian-2 theory for molecular energies of first- and second-row compounds. *The Journal of Chemical Physics* **94**, 7221 (1991).
43. DANOWITZ, A., KELLEY, K., MAO, J., STEVENSON, J. P. & HOROWITZ, M. CPU DB. *Communications of the ACM* **55**, 55 (avr. 2012).
44. DAVIDSON, E. R. The iterative calculation of a few of the lowest eigenvalues and corresponding eigenvectors of large real-symmetric matrices. *Journal of Computational Physics* **17**, 87–94 (jan. 1975).
45. De GROOT, F. Secret Opcodes <<https://web.archive.org/web/20090601101025/http://www.moyogo.com/blog/2005/09/secret-opcodes.html>> (2015).
46. DIGABEL, S. L. Algorithm 909. *ACM Transactions on Mathematical Software* **37**, 1–15 (fév. 2011).
47. DJANGO SOFTWARE FOUNDATION. Django : The web framework for perfectionists with deadlines <<https://www.djangoproject.com>> (2015).
48. DJOUDI, L. et al. MAQAO : Modular assembler quality Analyzer and Optimizer for Itanium 2 in Workshop on EPIC Architectures and Compiler Technology San Jose, California, United-States (mar. 2005).
49. DOCOPT ORGANISATION. docopt-language for description of command-line interface <<http://docopt.org>> (2015).
50. DONGARRA, J. J. *Applied parallel computing : computations in physics, chemistry, and engineering science : second international workshop, PARA '95, Lyngby, Denmark, August 21-24, 1995 : proceedings* ISBN : 3540609024 (Springer, Berlin New York, 1996).
51. DREPPER, U. What every programmer should know about memory. *Red Hat, Inc* **11** (2007).

52. DUNNING, J. & H., T. Gaussian basis sets for use in correlated molecular calculations. I. The atoms boron through neon and hydrogen. *The Journal of chemical physics* **90**, 1007–1023 (1989).
53. ENVIRONMENTAL MOLECULAR SCIENCES LABORATORY. *EMSL Basis Set Exchange*
54. EVANGELISTI, S., BENDAZZOLI, G. L., ANSALONI, R., DURÌ, F. & ROSSI, E. A one billion determinant full CI benchmark on the Cray T3D. *Chemical Physics Letters* **252**, 437–446 (avr. 1996).
55. EVANGELISTI, S., DAUDEY, J.-P. & MALRIEU, J.-P. Convergence of an improved CIPSI algorithm. *Chemical Physics* **75**, 91–102 (1983).
56. FELLER, D. The role of databases in support of computational chemistry calculations. *J. Comput. Chem.* **17**, 1571–1586 (oct. 1996).
57. FELLER, D. & PETERSON, K. A. Re-examination of atomization energies for the Gaussian-2 set of molecules. *The Journal of Chemical Physics* **110**, 8384 (1999).
58. FIALA, D. *et al. Detection and Correction of Silent Data Corruption for Large-scale High-performance Computing in Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (IEEE Computer Society Press, Salt Lake City, Utah, 2012), 78 :1–78 :12. ISBN : 978-1-4673-0804-5. <<http://dl.acm.org/citation.cfm?id=2388996.2389102>>.
59. FOULKES, W. M. C., MITAS, L., NEEDS, R. J. & RAJAGOPAL, G. Quantum Monte Carlo simulations of solids. *Rev. Mod. Phys.* **73**, 33–83 (1 jan. 2001).
60. FRACCHIA, F., FILIPPI, C. & AMOVILLI, C. Size-Extensive Wave Functions for Quantum Monte Carlo : A Linear Scaling Generalized Valence Bond Approach. *J. Chem. Theory Comput.* **8**, 1943–1951 (juin 2012).
61. FRANCE GRILLES. <<http://www.france-grilles.fr>> (2015).
62. FREE SOFTWARE FOUNDATION. *Make—GNU Projet* <<http://www.gnu.org/software/make/>> (2015).
63. FRISCH, M. J., TRUCKS, G. W., SCHLEGEL, H. B. & SCUSERIA, G. E. *Gaussian 09* Gaussian Inc. Wallingford CT 2009.
64. GENCI. *Rapport d'activités GENCI 2012* <[http://www.genci.fr/sites/default/files/RA2012-GENCI-Institution\\_0.pdf](http://www.genci.fr/sites/default/files/RA2012-GENCI-Institution_0.pdf)> (2015).
65. GERSHGORN, Z. & SHAVITT, I. An application of perturbation theory ideas in configuration interaction calculations. *International Journal of Quantum Chemistry* **2**, 751–759 (1968).
66. GINER, E. *Coupling Configuration Interaction and quantum Monte Carlo methods : The best of both worlds* Theses (Université de Toulouse, oct. 2014). <<https://hal.archives-ouvertes.fr/tel-01077016>>.

67. GINER, E., SCEMAMA, A. & CAFFAREL, M. Fixed-node diffusion Monte Carlo potential energy curve of the fluorine molecule F<sub>2</sub> using selected configuration interaction trial wavefunctions. *The Journal of Chemical Physics* **142**, 044115 (jan. 2015).
68. GINER, E., SCEMAMA, A. & CAFFAREL, M. Using perturbatively selected configuration interaction in quantum Monte Carlo calculations. *Canadian Journal of Chemistry* **91**, 879–885 (sept. 2013).
69. GODFREY, M. & HENDRY, D. The computer as von Neumann planned it. *IEEE Annals Hist. Comput.* **15**, 11–21 (1993).
70. GONZE, X. *et al.* First-principles computation of material properties : the ABINIT software project. *Computational Materials Science* **25**, 478–492 (2002).
71. GORDON, M. S. & MICHAEL, W. S. Advances in electronic structure theory : GAMESS a decade later (2005).
72. HACKMEYER, M. & WHITTEN, J. Configuration Interaction Studies of Ground and Excited States of Polyatomic Molecules II. The Electronic States and Spectrum of Pyrazine. *Journal of Chemical Physics* **54**, 3739–3750 (1971).
73. HAMMOND, B. L., LESTER, W. A. & REYNOLDS, P. J. *Monte Carlo methods in ab initio quantum chemistry* ISBN : 9789814317245 (World Scientific, Singapore River Edge, NJ, 1994).
74. HEWLETT-PACKARD & SCEMAMA, A. *Université Paul Sabatier saves space and energy with HP Moonshot* in (2015). <<http://www8.hp.com/h20195/v2/GetPDF.aspx/4AA5-9253ENW.pdf>> (visité le 12/08/2015).
75. HURON, B., MALRIEU, J. & RANCUREL, P. Iterative perturbation calculations of ground and excited state energies from multiconfigurational zeroth-order wavefunctions. *The Journal of Chemical Physics* **58**, 5745–5759 (1973).
76. iMATIX CORPORATION. *ZeroMQ : Code Connected* <<http://zeromq.org/>> (2015).
77. JANSSEN, C. *Parallel computing in quantum chemistry* ISBN : 9781420051643 (CRC Press, Boca Raton, 2008).
78. JOURNAL OFFICIEL DE LA RÉPUBLIQUE FRANÇAISE. *JORF no. 0129 du 6 juin 2010 page 10453 texte no. 42* (Direction des journaux officiels, Paris, 2010).
79. KALDERIMIS, J. & MEYER, M. *Travis CI – Test and deploy your code with confidence* <<https://travis-ci.org>> (2015).
80. KATO, T. On the eigenfunctions of many-particle systems in quantum mechanics. *Communications on Pure and Applied Mathematics* **10**, 151–177 (1957).

81. KLEIN, A. *Flexx : Python UI toolkit based on web technology* <<https://github.com/zoofIO/flexx>> (2015).
82. KNUPP, J. *Open Sourcing a Python Project the Right Way* <<http://jeffknupp.com/blog/2013/08/16/open-sourcing-a-python-project-the-right-way/>> (2015).
83. KOSKINEN, J., LAHTONEN, H. & TILUS, T. *Software Maintenance Cost Estimation and Modernization Support* rapp. tech. (University of Jyväskylä, 2003). <<http://users.jyu.fi/~%5C~%7B%7Dkoskinen/smcems.pdf>>.
84. KRAUT, R. E. & STREETER, L. A. Coordination in software development. *Communications of the ACM* **38**, 69–81 (1995).
85. KUTZELNIGG, W.  $l$ -Dependent terms in the wave function as closed sums of partial wave amplitudes for large  $l$ . *Theoretica Chimica Acta* **68**, 445–469 (déc. 1985).
86. KUTZELNIGG, W. & KLOPPER, W. Wave functions with terms linear in the interelectronic coordinates to take care of the correlation cusp. I. General theory. *The Journal of Chemical Physics* **94**, 1985 (1991).
87. L'ECUYER, P. Combined Multiple Recursive Random Number Generators. *Operations Research* **44**, 816–822 (oct. 1996).
88. LENTHE, E. V. & BAERENDS, E. J. Optimized Slater-type basis sets for the elements 1-118. *J. Comput. Chem.* **24**, 1142–1156 (mai 2003).
89. LEVESQUE, J. *High performance computing : programming and applications* ISBN : 978-1-4200-7705-6' (Chapman & Hall/CRC, Boca Raton, FL, 2011).
90. MARTINE, E. *Ninja is a small build system with a focus on speed* <<http://martine.github.io/ninja/>> (2015).
91. MATSEN, F. in *Advances in Quantum Chemistry* 59–114 (Elsevier BV, 1964). doi :[10.1016/S0065-3276\(08\)60375-5](https://doi.org/10.1016/S0065-3276(08)60375-5). <[http://dx.doi.org/10.1016/S0065-3276\(08\)60375-5](http://dx.doi.org/10.1016/S0065-3276(08)60375-5)>.
92. McVOY, L. & STAELIN, C. *Lmbench : Portable Tools for Performance Analysis* in *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference* (USENIX Association, San Diego, CA, 1996), 23–23. <<http://dl.acm.org/citation.cfm?id=1268299.1268322>>.
93. MELL, P. & GRANCE, T. The NIST definition of cloud computing (2011).
94. MILLER, P. Recursive make considered harmful. *The Journal of AUUG Inc* (1988).

95. NEESE, F. *In praise of CIPSI in European Workshop on Theoretical approaches of Molecular Magnetism : JUJOLS VIII* (juin 2015).
96. O'EILL, D. P. & GILL, P. M. W. Benchmark correlation energies for small molecules. *Molecular Physics* **103**, 763–766 (mar. 2005).
97. O'NEILL, M. *PCG, A Family of Better Random Number Generators* <<http://www.pcg-random.org/>> (2015).
98. OUSTERHOUT, J. Scripting : higher level programming for the 21st Century. *Computer* **31**, 23–30 (mar. 1998).
99. POU-AMÉRIGO, R., MERCHÁN, M., NEBOT-GIL, I., WIDMARK, P.-O. & ROOS, B. O. Density matrix averaged atomic natural orbital (ANO) basis sets for correlated molecular wave functions. *Theoretica Chimica Acta* **92**, 149–181 (sept. 1995).
100. PRESTON-WERNER, T., WANSTRATH, C. & HYETT, P. *GitHub* <<https://github.com>> (2015).
101. RAMANARAYANAN, R., MATHEW, S., KRISHNAMURTHY, R., GUERON, S. & ERRAGUNTLA, V. *Combined set bit count and detector logic* US Patent 8,214,414. Juil. 2012. <<http://www.google.com/patents/US8214414>>.
102. REITZ, K., BENFIELD, C. & CORDASCO, I. *Requests : HTTP for Humans* <<http://docs.python-requests.org/en/latest/>> (2015).
103. RÍOS, P. L., MA, A., DRUMMOND, N. D., TOWLER, M. D. & NEEDS, R. J. Inhomogeneous backflow transformations in quantum Monte Carlo calculations. *Physical Review E* **74**. doi :[10.1103/PhysRevE.74.066701](https://doi.org/10.1103/PhysRevE.74.066701). <<http://dx.doi.org/10.1103/PhysRevE.74.066701>> (déc. 2006).
104. RIVERBANK COMPUTING LIMITED. *Pyqt* <<https://www.riverbankcomputing.com/software/pyqt/intro>> (2015).
105. RONACHER, A. *Flask : web development one drop at a time* <<https://www.djangoproject.com>> (2015).
106. SCEMAMA, A., APPENCOURT, T., GINER, E. & CAFFAREL, M. Quantum Monte Carlo with very large multi-determinant expansions. *Preprint* —, (2015).
107. SCEMAMA, A., APPELCOURT, T., GINER, E. & CAFFAREL, M. Accurate nonrelativistic ground-state energies of 3d transition metal atoms. *The Journal of Chemical Physics* **141**, 244110 (déc. 2014).
108. SCEMAMA, A. & GINER, E. An efficient implementation of Slater-Condon rules. *ArXiv e-prints*. arXiv : [1311.6244](https://arxiv.org/abs/1311.6244) [[physics.comp-ph](https://arxiv.org/archive/physics)] (nov. 2013).
109. SCEMAMA, A. *IRPF90* <<http://irpf90.ups-tlse.fr>> (2015).

110. SCEMAMA, A., APPLENCOURT, T., CAFFAREL, M. & da COSTA, G. <[http://devlog.cnrs.fr/\\_media/jdev2015/jdev2015\\_t5\\_anthonyscemama\\_francegrilles\\_20150702.pdf](http://devlog.cnrs.fr/_media/jdev2015/jdev2015_t5_anthonyscemama_francegrilles_20150702.pdf)> (2015).
111. SCHMIDT, K. E. & MOSKOWITZ, J. W. Correlated Monte Carlo wave functions for the atoms He through Ne. *The Journal of Chemical Physics* **93**, 4172 (1990).
112. SEIBEL, P. *Coders at Work : Reflections on the Craft of Programming* ISBN : 1430219483 (Apress, 2009).
113. SIBAJA-HERNANDEZ, A. *et al.* Half-terahertz silicon/germanium heterojunction bipolar technologies : A TCAD based device architecture exploration. *Solid-State Electronics* **65-66**, 72–80 (nov. 2011).
114. SLATER, J. C. Atomic Shielding Constants. *Physical Review* **36**, 57–64 (juil. 1930).
115. SLATER, J. C. The Theory of Complex Spectra. *Phys. Rev.* **34**, 1293–1322 (10 nov. 1929).
116. SOFTWARE FREEDOM CONSERVANCY. *Git* <<https://git-scm.com>> (2015).
117. THE PLOTLY TEAM. *Collaborative data science*. <<https://plot.ly>> (2015).
118. THE PYTHON SOFTWARE FOUNDATION. *unittest — Unit testing framework* <<https://docs.python.org/2/library/unittest.html>> (2015).
119. THE QT COMPANY. *Qt - Build your world* <<http://www.qt.io/>> (2015).
120. TOWLER, M. D. in *Electronic Structure Approaches for Biotechnology and Nanotechnology* 117–166 (Wiley-Blackwell, juil. 2011). doi :[10.1002/9780470930779.ch4](https://doi.org/10.1002/9780470930779.ch4). <<http://dx.doi.org/10.1002/9780470930779.ch4>>.
121. TURING, A. M. On Computable Numbers, with an Application to the Entscheidungsproblem. *j-PROC-LONDON-MATH-SOC-2* **42**, 230–265. ISSN : 0024-6115 (1936).
122. 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**. doi :[10.1103/PhysRevLett.98.110201](https://doi.org/10.1103/PhysRevLett.98.110201). <<http://dx.doi.org/10.1103/PhysRevLett.98.110201>> (mar. 2007).
123. Van der DOES, P. *git-flow (AVH Edition)* <<https://github.com/petervanderdoes/gitflow>> (2015).



124. WHITTEN, J. & HACKMEYER, M. Configuration Interaction Studies of Ground and Excited States of Polyatomic Molecules. I. The CI Formulation and Studies of Formaldehyde. *The Journal of Chemical Physics* **51**, 5584–5596 (1969).
125. WIKIPEDIA. *Circle-ellipse problem* <[https://en.wikipedia.org/wiki/Circle-ellipse\\_problem](https://en.wikipedia.org/wiki/Circle-ellipse_problem)> (2015).
126. WOLFGANG, P. *Design patterns for object-oriented software development* (Reading, Mass. : Addison-Wesley, 1994).
127. WYNN, J. *A Hacker's Guide to Git* <<http://wildlyinaccurate.com/a-hackers-guide-to-git/#introduction>> (2015).
128. XKCD. *A webcomic of romance, sarcasm, math, and language*. (Creative Commons Attribution-NonCommercial 2.5 License) <<https://xkcd.com/>> (2015).

## RÉSUMÉ

L'objectif de ce travail de thèse est double : 1. Le développement et application de méthodes originales pour la chimie quantique ; 2. La mise au point de stratégies informatiques variées permettant la réalisation de simulations à grande échelle. Dans la première partie, les méthodes d'Interaction de Configuration (IC) et *Quantum Monte Carlo*, Monte Carlo quantique (QMC) utilisées dans ce travail pour le calcul des propriétés quantiques sont présentées. Nous détaillerons en particulier la méthode d'IC sélectionnée perturbativement, *Configuration Interaction using Perturbative Selection done Iteratively* (CIPSI) que nous avons utilisée pour construire des fonctions d'onde d'essai pour le QMC. La première application concerne le calcul des énergies totales non-relativistes des atomes de transition de la série 3d ; ceci a nécessité l'implémentation de fonctions de base de type Slater et a permis d'obtenir les meilleures valeurs publiées à ce jour. La deuxième application concerne l'implémentation de pseudo-potentiels adaptés à notre approche QMC, avec pour application une étude concernant le calcul des énergies d'atomisation d'un ensemble de 55 molécules. La seconde partie traite des aspects *High Performance Computing* (HPC) avec pour objectif l'aide au déploiement des simulations à très grande échelle, aussi bien sous l'aspect informatique proprement dit – utilisation de paradigmes de programmation originaux, optimisation des processus monocœurs, calculs massivement parallèles sur grilles de calcul (supercalculateur et *Cloud*), outils d'aide au développement collaboratif *et cætera* –, que sous l'aspect *utilisateur* – installation, gestion des paramètres d'entrée et de sortie, interface graphique, interfaçage avec d'autres codes. L'implémentation de ces différents aspects dans nos codes-maison *Quantum Package* et *QMC=Chem*) est également présentée.

## ABSTRACT

This thesis work has two main objectives : 1. To develop and apply original electronic structure methods for quantum chemistry 2. To implement several computational strategies to achieve efficient large-scale computer simulations. In the first part, both the Configuration Interaction (CI) and the *Quantum Monte Carlo*, Monte Carlo quantique (QMC) methods used in this work for calculating quantum properties are presented. We then describe more specifically the selected CI approach (so-called CIPSI approach, Configuration Interaction using a Perturbative Selection done Iteratively) that we used for building trial wavefunctions for QMC simulations. As a first application, we present the QMC calculation of the total non-relativistic energies of transition metal atoms of the 3d series. This work, which has required the implementation of Slater type basis functions in our codes, has led to the best values ever published for these atoms. We then present our original implementation of the pseudo-potentials for QMC and discuss the calculation of atomization energies for a benchmark set of 55 organic molecules. The second part is devoted to the *High Performance Computing* (HPC) aspects. The objective is to make possible and/or facilitate the deployment of very large-scale simulations. From the point of view of the developer it includes : The use of original programming paradigms, single-core optimization process, massively parallel calculations on grids (supercomputer and Cloud), development of collaborative tools , etc – and from the user's point of view : Improved code installation, management of the input/output parameters, GUI, interfacing with other codes, etc.