Prospectives IN2P3, GT09 : le point de vue du collectif Reprises

23 septembre 2019

Pierre Aubert[†], David Chamont[+], Hadrien Grasland[+], Bogdan Vulpescu[*], Vincent Lafage[⊥], Arnaud Beck[††], Luisa Arrabito[‡], Emmanuel Medernach[⊤]

LAPP : [†]
LAL : [+]
LPC : [*]
IPNO : [⊥]
LLR : [††]
LUPM : [‡]
IPHC : [⊤]

Table des matières

1 Introduction **7**

2 Pourquoi optimiser les programmes ? **9**
2.1 Qu'apporte l'optimisation des programmes ? . . . 10
2.2 Méthodes d'optimisation . . . 10
2.2.1 Préparer l'optimisation . . . 10
2.2.2 Optimisation du format de données . . . 11
2.2.2.1 Contiguïté des données . . . 12
2.2.2.2 Préchargement des données . . . 12
2.2.2.3 Localité des données . . . 13
2.2.2.4 Caches CPU . . . 14
2.2.2.5 Tableau, vecteur ou liste ? . . . 15
2.2.3 Optimisation du calcul . . . 16
2.2.3.1 Bibliothèques bas niveau . . . 16
2.2.3.2 Bibliothèques haut niveau . . . 16
2.2.3.3 Changer d'algorithme . . . 17
2.2.3.4 Effectuer moins de calculs . . . 17
2.2.3.5 Réduire l'empreinte mémoire . . . 17
2.2.3.6 Supprimer certain branchements . . . 18
2.3 Impact du cycle de développement sur les optimisations . . . 20

3 Pourquoi et comment vectoriser ? **23**
3.1 Principe de la vectorisation . . . 24
3.2 Pourquoi vectoriser ? . . . 24
3.3 Ce calcul est-il vectorisable ? . . . 25
3.3.1 Propriétés du format données . . . 25
3.3.1.1 Données contiguës . . . 25
3.3.1.2 Données alignées . . . 25
3.3.2 Indépendances des éléments . . . 28
3.4 Comment vectoriser ? . . . 29
3.4.1 Vectoriser avec le compilateur . . . 29
3.4.2 Utiliser des fonctions intrinsèques . . . 30
3.4.3 Les bibliothèques optimisées . . . 31
3.4.4 Les bibliothèques d'auto-vectorisation . . . 32
3.4.5 Les générateurs de code . . . 32

3.5 Ce code a-t-il été vectorisé par le compilateur ? . . . 33
3.6 Retours d'expériences . . . 33
3.6.1 Vectorisation de la simulation CORSIKA . . . 33
3.6.1.1 Introduction à la simulation CORSIKA . . . 33
3.6.1.2 Retour d'expérience . . . 34
3.6.2 Vectorisation de l'analyse de CTA . . . 35
3.6.2.1 L'expérience CTA . . . 35
3.6.2.2 Retour d'expérience . . . 35
3.6.3 Vectorisation dans Smilei . . . 36
3.6.3.1 Le code Smilei . . . 36
3.6.3.2 Retour d'expérience . . . 36
3.7 Recommandations pour le long terme . . . 37
3.7.1 Structure/Stockage de données . . . 37
3.7.2 Bonnes pratiques . . . 37
3.7.3 Vectoriser un programme existant . . . 38
3.7.4 Vectoriser un nouveau programme . . . 38

4 Parallélisation **41**
4.1 Introduction . . . 43
4.2 Considérations théoriques . . . 44
4.2.1 Granularité des tâches . . . 44
4.2.2 Passage à l'échelle . . . 44
4.2.2.1 Introduction . . . 44
4.2.2.2 Loi d'Amdahl . . . 45
4.2.2.3 Loi de Gustafson . . . 45
4.2.2.4 Applications réelles . . . 46
4.2.3 Parallélisation sur la Grille . . . 47
4.3 Formes de parallélisation . . . 48
4.3.1 Introduction . . . 48
4.3.2 Parallélisation multi-cœur . . . 48
4.3.2.1 Threads, cœurs et processeurs . . . 48
4.3.2.2 Mémoire partagée et cohérence de cache . . . 49
4.3.2.3 Abstractions d'exécution parallèle . . . 50
4.3.2.4 Synchronisation . . . 51
4.3.2.5 Abstractions d'ordre supérieur . . . 52
4.3.2.6 Paradigmes pour le parallélisme . . . 53
4.3.3 Parallélisme multi-nœud . . . 54
4.3.3.1 La fin des certitudes . . . 54
4.3.3.2 Réseau . . . 54
4.3.3.3 Du paquet au message . . . 55
4.3.3.4 Limite des messages . . . 56
4.3.3.5 Approches haut niveau . . . 57
4.3.3.6 Tolérance aux pannes . . . 58
4.4 Les aléas du parallélisme . . . 59
4.4.1 Passage à l'échelle . . . 59

4.4.2 Équilibrage de charge . . . 60
4.4.3 Algorithmie . . . 60
4.4.4 Débogage . . . 61
4.4.5 Profilage . . . 62
4.5 Retours d'expériences . . . 63
4.5.1 Introduction . . . 63
4.5.2 *Multi-threading* en physique des particules . . . 63
4.5.3 Calcul distribué dans LSST . . . 65
4.6 Conclusion . . . 66

5 Quels langages de programmation ? 69
5.1 Introduction . . . 70
5.2 C . . . 70
5.3 C++ . . . 71
5.4 Fortran . . . 72
5.5 Ada . . . 73
5.6 Rust . . . 74
5.7 Go . . . 75
5.8 Python . . . 76
5.9 Julia . . . 77
5.10 Conclusions . . . 78

6 Des FPGAs pour le calcul ? 81
6.1 FPGA : généralités . . . 82
6.1.1 Qu'est-ce qu'un FPGA ? . . . 82
6.1.2 Comment programmer un FPGA ? . . . 82
6.1.3 Programmer des calculs sur le FPGA . . . 83
6.1.3.1 Parallélisme sur GPU et sur FPGA . . . 84
6.2 Calcul sur FPGA avec OpenCL . . . 86
6.2.1 Le matériel . . . 86
6.2.2 La programmation . . . 89
6.3 Évolution des accélérateurs avec FPGA Altera/Intel . . . 92
6.3.1 Solutions Intel avec OpenVINO pour l'apprentissage automatique . . . 93
6.4 Accélération avec FPGA Xilinx sur la plate-forme ACP du Laboratoire Leprince-Ringuet . . . 94

7 Les GPUs... pour quoi faire ? 97
7.1 Introduction . . . 98
7.1.1 Structure d'un GPU . . . 100
7.1.2 Fonctionnement d'un GPU . . . 100
7.2 Omniprésence des GPUs . . . 101
7.3 Utilisation des GPUs . . . 102
7.4 Programmation . . . 103
7.4.1 Langages de programmation bas niveau . . . 103
7.4.2 Langages de programmation haut niveau . . . 103

7.5 Retour sur expériences . . . 103
7.5.1 Projet Sympatick_G . . . 103
7.5.2 Projet CMS-MEM . . . 105
7.5.2.1 Une méthode d'analyse pour CMS . . . 105
7.5.2.2 Contraintes et défis technologiques . . . 105
7.5.2.3 Production sur la plate-forme GPUs du CC-IN2P3 . . . 106
7.5.2.4 Remerciements . . . 107
7.6 Recommandations . . . 107

8 La problématique de la précision 109
8.1 Introduction . . . 110
8.2 Erreur de calcul . . . 110
8.3 Sécurité numérique et évaluation de la précision d'un calcul . . . 111
8.4 Reproductibilité des résultats . . . 112

9 Recommandation de production de logiciels open-source 115
9.1 Introduction . . . 116
9.2 Modularité . . . 116
9.3 Réutilisabilité . . . 116
9.4 Développement . . . 117
9.4.1 Convention de nommage . . . 117
9.4.2 Écrire des fonctions courtes . . . 117
9.4.3 Outils de développement . . . 117
9.4.4 Cycles courts . . . 118
9.4.5 Tests unitaires . . . 118
9.4.6 Tests de qualité d'exécution/fuites mémoire . . . 118
9.5 Chaîne de compilation . . . 119
9.5.1 Choix du compilateur . . . 119
9.6 Documentation . . . 120
9.7 Gestion des versions et sauvegarde des projets . . . 121
9.7.1 Utilisation de ces outils . . . 122

10 Conclusion 123
10.1 Enseignements à propos de la quête de performance . . . 124
10.1.1 Optimisation . . . 124
10.1.2 Vectorisation . . . 124
10.1.3 Parallélisation . . . 125
10.1.4 Langages de programmation . . . 125
10.1.5 Calcul sur FPGA . . . 126
10.1.6 Calcul sur GPU . . . 127
10.1.7 Problématique de la précision . . . 127
10.1.8 Recommandations de production de logiciel open source . . . 128
10.2 Thèmes techniques prioritaires . . . 128
10.2.1 Profilage du calcul et des entrées/sorties en contexte parallèle . . . 128
10.2.2 Structures de données performantes . . . 128

10.2.3 Modes de programmation PPP 128
10.2.4 Calcul sur FPGA 129
10.2.5 Calcul multi-précision 129
10.2.6 Reproductibilité numérique en contexte parallèle et hétérogène 129
10.2.7 Apprentissage automatique 129
10.3 Recommandations stratégiques 129
10.3.1 1. Recruter des ingénieurs pour le calcul 129
10.3.2 2. Former massivement 129
10.3.3 3. Avoir le matériel approprié pour une veille active 130
10.3.4 4. Développer une recherche appliquée interne 130
10.3.5 5. Constituer une force de frappe trans-laboratoires « IN2P3 Optimisation » 130

Appendices **131**

A Optimisation et prédicteur de branchements **133**

Chapitre 1

Introduction

Depuis que la fréquence de fonctionnement du matériel de calcul stagne, pour des raisons de dissipation thermique, les fabricants rivalisent d'imagination pour multiplier et spécialiser les cœurs à l'infini. La programmation de ce matériel devient une affaire de spécialistes et l'offre technologique explose. Nos physiciens ne peuvent plus écrire leurs applications de façon candide, en faisant mine d'avoir une simple machine de von Neumann, séquentielle avec une mémoire agréablement uniforme. Il faut prendre en compte le matériel hétérogène sous-jacent, combiner différentes approches parallèles, et arracher des gains de performances en optimisant le code de façon agressive. Car malgré la nouvelle donne, les concepteurs des instruments de physique persistent à réclamer un gain d'un ordre de grandeur à chaque nouvelle génération informatique.

Or, optimiser le code, c'est généralement nuire à sa lisibilité, à sa portabilité, et à sa maintenabilité à long terme. Il convient donc, en théorie, de le faire le moins et le plus tard possible, en s'appuyant sur des profilages fins permettant de circonscrire l'optimisation aux portions de code les plus impactantes sur la performance globale. Malheureusement, en pratique, certaines optimisations ne peuvent plus être faites si on n'a pas opté pour les bons choix dès le début, notamment en ce qui concerne les structures de données. Pire, ces bons choix en amont sont souvent assez différents selon les technologies d'optimisation finalement utilisées. Et nous parlons de « technologies » au pluriel, parce qu'aucune ne saurait tout faire, et qu'il faut absolument les cumuler au sein d'un même programme, avec une programmation dite « hybride ».

C'est un des objectifs principaux du groupe Reprises, que d'évaluer les technologies en concurrence, et de rechercher le juste équilibre entre performance, portabilité et facilité d'écriture et de maintenance, ce que l'on qualifie à présent de « *Performance, Portability and Productivity* » dans l'un des nouveaux ateliers de la conférence SuperComputing (https://p3hpc2018.lbl.gov/). À ces trois « P », nous ajoutons celui de la précision numérique. En effet, une des potentialités de gain de performance réside dans la réduction de cette précision, lorsqu'elle est inutilement élevée. Encore faut-il reprendre le contrôle de nos erreurs de calcul flottant, dont on fait trop peu de cas dans notre communauté.

Dans ce document, nous tentons de faire le bilan de notre expérience dans ces différents sujets, en mêlant des considérations générales sur l'optimisation, des chapitres dédiés aux thèmes clefs du moment (vectorisation, GPU, FPGA), ou des thèmes qui nous paraissent importants (précision). L'inévitable guerre des langages de programmation est également

abordée. En conclusion, nous essayons d'en tirer des enseignements à destination des chercheurs, et des recommandations à destination des décideurs de l'institut.

Chapitre 2

Pourquoi optimiser les programmes ?

Sujet du chapitre 2 –
L'optimisation est une méthode qui a pour but de réduire le temps d'exécution des programmes. Ce chapitre explique pourquoi l'optimisation des analyses est indispensable pour répondre aux besoins de calcul des expériences de physique actuelles et à venir, et expose les méthodes d'optimisation les plus utilisées.

Sommaire

2.1 Qu'apporte l'optimisation des programmes ? 10
2.2 Méthodes d'optimisation 10
2.2.1 Préparer l'optimisation 10
2.2.2 Optimisation du format de données 11
2.2.3 Optimisation du calcul 16
2.3 Impact du cycle de développement sur les optimisations 20

2.1 Qu'apporte l'optimisation des programmes ?

L'optimisation des analyses de données se traduit directement par une réduction du temps de calcul. L'implication immédiate est que les centres de calculs seront utilisés plus efficacement, car la durée moyenne des jobs diminuera. Le temps économisé permettra de traiter des jobs supplémentaires.

D'autre part, les physiciens attendront moins longtemps les résultats de leurs analyses. Ils pourront traiter plus de données dans le même temps ou utiliser une analyse de physique plus complète qui prendra en compte davantage de phénomènes ou qui utilisera des modèles plus fins avec plus de paramètres, *etc.* La qualité des analyses pourra donc aussi être améliorée.

L'optimisation des analyses de données réduit le coût brut des jobs (électricité, maintenance) puisqu'ils peuvent traiter plus de données dans le même temps.

Ce sera la seule méthode pour analyser les quantités de données colossales que produiront les nouvelles expériences de physique (CTA [1], SKA [2], FCC, , *etc.*).

2.2 Méthodes d'optimisation

2.2.1 Préparer l'optimisation

Avant d'optimiser un programme, il est nécessaire de vérifier s'il ne contient pas de bogues. Cette tâche est difficile à réaliser dans des analyses importantes mais est grandement facilitée par l'utilisation de tests unitaires [1] tout au long du développement.

Un niveau de vérification supplémentaire peut être mis en place en utilisant le programme *Valgrind* [3] pour détecter des opérations illégales au cours de l'exécution du programme qui sont sources de bogues cachés [2] (voir cours [4]).

Certains compilateurs fournissent des outils de vérification similaires, appelés « *sanitizers* », qui ont l'avantage de détecter plus de problèmes que *Valgrind* et de produire moins de faux positifs grâce à une connaissance plus précise du programme étudié. Malheureusement, ces outils ne sont pas disponibles pour tous les langages de programmation, et leur utilisation nécessite d'effectuer une compilation spéciale du programme et de toutes les bibliothèques tierces où l'on souhaite rechercher des erreurs.

Dès lors que tous les bogues (connus) du programme sont corrigés, il est important de vérifier si ce dernier ne produit pas de fuite mémoire. En effet, une fuite mémoire fera augmenter progressivement la quantité de mémoire RAM utilisée par le programme jusqu'à ce que la totalité de la RAM soit saturée. Certaines fuites peuvent passer inaperçues lors de tests sur une quantité de données réduite (des tests unitaires par exemple). Le programme

1. Les tests unitaires sont des programmes qui vérifient qu'une fonctionnalité se comporte bien comme prévu avec des jeux de paramètres définis. Quand une fonctionnalité ajoutée à un programme est accompagnée de tests unitaires vérifiant son bon fonctionnement, il est possible de les utiliser tout au long de la vie du programme pour vérifier que les mises à jour ne cassent pas cette fonctionnalité. Lors d'un cycle de développement standard, les tests unitaires sont lancés au moins une fois pas jour pour vérifier que le programme en question reste cohérent.

2. Les bogues cachés ne se produisent pas toujours durant l'exécution du programme, ce qui les rend extrêmement difficiles à corriger.

Valgrind [3] permet aussi un diagnostic de l'utilisation de la mémoire afin de détecter toute fuite mémoire. Il s'avère très efficace d'utiliser *Valgrind* sur les tests unitaires de manière automatique pour prévenir ce genre de problèmes.

Lorsque les bogues et les fuites mémoire ont été corrigés, il est nécessaire d'évaluer quelles fonctions devront être optimisées en priorité. Ce seront généralement les fonctions les plus fréquemment appelées. Là encore, il est important d'utiliser des outils adéquats. Les programmes *Valgrind* [3], *maqao* [5], *gprof* [6] et *perf* [7] permettent d'effectuer un profilage du programme afin de déterminer les fonctions les plus coûteuses, dite chaudes. Il faut néanmoins prendre garde à ce que le programme profilé ne s'exécute pas trop rapidement pour que les fonctions d'initialisation ne prennent pas le pas sur les fonctions qui seront effectivement les plus utilisées en production.

Le profilage donne le point de départ de l'optimisation mais quelques réflexes permettent de développer des programmes plus rapides.

Les programmes utilisent des bibliothèques tierces, fournies par d'autres développeurs tels que GNU, Khronos, KitWare, ou Apache. Il peut être difficile pour les développeurs d'une analyse de physique de physique de modifier ces bibliothèques, car elles sont parfois très complexes et un changement pérenne ne peut y être instauré qu'avec l'aide de leurs développeurs, qui ne sont pas toujours ouverts à cette possibilité. Si programme utilise une fonction lente que ses développeurs ne peuvent pas optimiser, il est possible de contourner le problème en appelant cette fonction le moins souvent possible.

Par exemple, si un tableau est nécessaire pour effectuer un calcul sur un événement. Il est intéressant de ne l'allouer qu'une seule fois pour analyser tous les événements à traiter et de le réutiliser. Les fonctions d'allocations sont lentes, mais l'utilisation de temporaires (voir section 2.2.3.5) permet d'amortir leurs appels. La lisibilité d'un programme peut être améliorée en regroupant tous les temporaires d'une même tâche dans une classe commune.

Dans un autre cas d'utilisation, si le programme doit appeler une base de données, il peut être intéressant de n'effectuer qu'une seule requête complexe plutôt que de nombreuses requêtes simples en stockant le résultat obtenu pour ne pas rappeler la base de données. De plus, les bases de données peuvent subir des latences très importantes. Un programme qui dépend d'une base de données devra donc être conçu pour être affecté le moins possible par ses temps de réponse.

Enfin, un programme peut être ralenti par l'utilisation d'un format de données inadapté pour le HPC. Dans ce cas, le format de données devra être optimisé avant les calculs (voir section suivante).

2.2.2 Optimisation du format de données

Comme nous l'avons vu dans la section précédente, le format de données joue un rôle crucial dans l'optimisation d'une analyse de données. Il doit donc être optimisé au même titre que les calculs. Cette section décrit les principales caractéristiques que doit satisfaire un format de données pour permettre une analyse HPC.

Note 1 –

Écrire de petites fonctions :

— Permet de cerner beaucoup plus facilement les calculs qui coûtent le plus cher en performances. Notamment à l'aide de programmes de profilage comme *valgrind* [3].
— Facilite le débogage (une fonction de quelques lignes sera plus simple à comprendre qu'une fonction de 1000 lignes).
— Facilite l'optimisation du compilateur, car les calculs effectués seront plus simples donc plus facilement optimisables.

2.2.2.1 Contiguïté des données

Les accès mémoire se font par blocs de données (64 octets pour les accès RAM des processeurs Intel actuels). Pour qu'ils soient efficaces, il faut que l'intégralité de chaque bloc obtenu soit utilisée. Cela implique l'utilisation de données contiguës.

La contiguïté des données s'obtient par l'utilisation de tableaux, matrices ou tenseurs (voir figure 2.1). Cela implique que toutes les données d'un même type (entier, flottant, *etc.*) et d'une même signification (signal, position, *etc.*) soient adjacentes en mémoire. Cette configuration améliore autant les performances du programme que sa lisibilité. De plus, l'utilisation de telles représentations permettra aux développeurs d'utiliser plus facilement des GPUs s'ils le souhaitent.

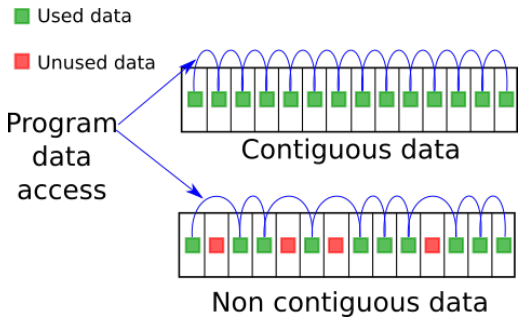

FIGURE 2.1 – **En haut :** L'utilisation de données contiguës améliore les performances d'accès aux données. **En bas :** Si les données ne sont pas contiguës, les accès aux données seront plus lents.

En revanche, l'utilisation de représentations non contiguës, telles que des listes, dégradera les performances. En effet, les unités de calculs devront attendre plus longtemps le rapatriement des données dans les registres du CPU.

2.2.2.2 Préchargement des données

Le préchargement des données est une technique d'optimisation qui consiste à rapatrier des données, de la RAM aux registres CPU par exemple, avant que le programme n'en ait besoin. En effet, la vitesse de lecture de la mémoire RAM est très lente par rapport à celle des registres et le temps de rapatriement est donc très important. Dans ces conditions il est préférable de rapatrier davantage de données afin d'amortir le coût de cette manipulation de données.

La section précédente montre qu'un stockage avec des données contiguës est essentiel.

Le préchargement des données implique également que ces données contiguës peuvent être lues dans un ordre optimal (voir figure 2.2). Comme le préchargement copiera des données adjacentes, le programme doit les lire de la même manière, sinon les données préchargées ne seront pas utiles et le programme ne sera pas accéléré.

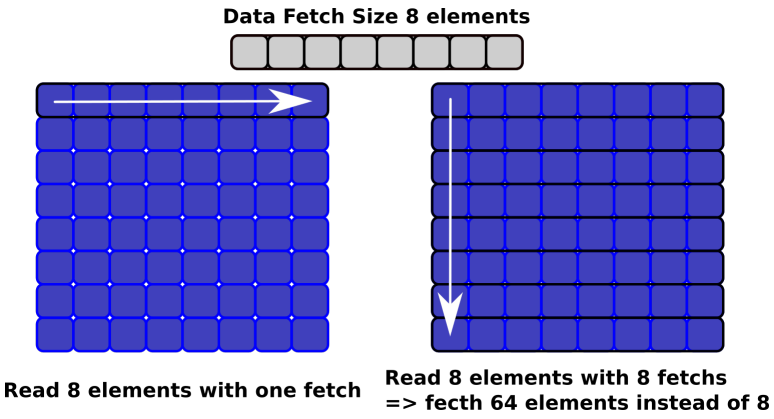

FIGURE 2.2 – Préchargement CPU des données en fonction de leur lecture par le programme.

2.2.2.3 Localité des données

La localité des données permet d'évaluer si le voisinage d'une donnée utile au programme en contient d'autres utilisées pour le même calcul (voir figure 2.3)

Ainsi, un stockage qui utilise des tableaux, des matrices ou des tenseurs verra la localité de ses données grandement améliorée ce qui optimisera naturellement les mécanismes de gestions des données propres au CPU.

FIGURE 2.3 – Localité des données en mémoire.

2.2.2.4 Caches CPU

Le mécanisme de cache a été introduit pour optimiser la vitesse de lecture des données de la RAM au CPU. Les caches sont classiquement au nombre de 4 (voir figure 2.4) :

— Le cache L1d de capacité 32 kB : contient des données et est au plus près du CPU.
— Le cache L1i de capacité 32 kB : pendant du cache L1d pour les instructions.
— Le cache L2 de capacité 256 kB.
— Le cache L3 de capacité 8 MB : le plus près de la RAM.

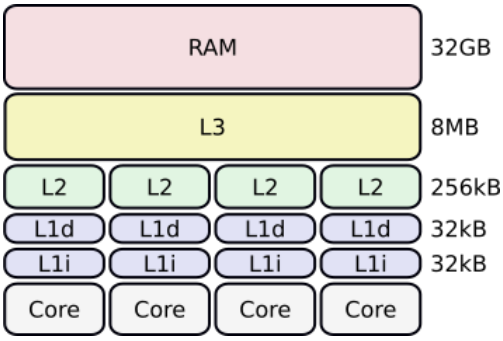

FIGURE 2.4 – Caches dans un CPU.

Généralement les caches L1i, L1d et L2 sont dédiés par cœur tandis que le cache L3 est commun, au même titre que la RAM. Certain CPUs ont des caches L2 et L3 plus grands, respectivement de 1024 kB et 16 MB.

Le temps nécessaire pour accéder à une donnée est également différent. Typiquement, une donnée dans le cache L1d est lue en 1 cycle[3], dans le cache L2 en 6 cycles, dans le cache L3 en 10 cycles et dans la RAM en 25 cycles ou plus.

Sans mécanisme de cache, un processeur ayant une fréquence d'horloge de 2 GHz ne pourrait donc pas calculer à plus de 80 MHz.

Selon le positionnement des données dans la RAM, des conflits peuvent apparaître (voir figure 2.5). En simplifiant, les adresses des données dans les caches seront obtenues par leurs adresses respectives dans la RAM modulo la taille du cache. Par conséquent, certaines données dont les adresses RAM sont différentes peuvent avoir des adresses identiques dans les caches.

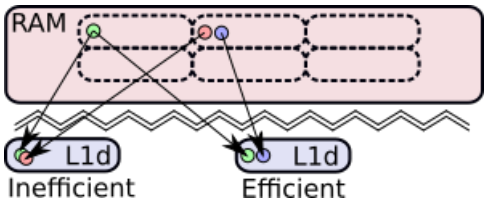

FIGURE 2.5 – Conflits d'accès au cache.

Lorsque de tels conflits se produisent, la dernière donnée rapatriée chasse la donnée précédente. Or, si un calcul a besoin de ces deux données en même temps, le CPU devra chercher de nouveau la donnée perdue en RAM, ou dans un autre cache (L2 ou L3).

Ce phénomène est d'autant plus préoccupant que les constructeurs de CPUs ont ajouté un mécanisme dit de coloration de cache pour améliorer la vitesse de lecture de ces derniers. Mais il peut engendrer des conflits supplémentaires, car les caches sont découpés en N couleurs afin d'accéder plus rapidement aux données qu'ils contiennent.

Par conséquent, les analyses ne doivent pas utiliser trop de tableaux dans le même calcul (idéalement moins que de couleurs de cache) et doivent utiliser une politique de pagination adéquate pour éviter ces problèmes qui dégradent fortement les performances de calculs.

Les OS ont différentes manières d'allouer de la mémoire aux applications et il peut arriver que certains fournissent des blocs de mémoire systématiquement alignés en RAM. Cela

3. Unité de temps de référence basée sur l'horloge des processeurs.

détériorera les performances d'un programme qui utilise de nombreux tableaux en même temps. Certains outils ont été développés pour atténuer ce problème en réimplémentant un allocateur [8].

Note 2 –
Il n'existe pas à proprement parler de programmes capables de mesurer précisément l'utilisation du cache ou la localité des données.
Mais des outils comme *perf* [7] ou *Valgrind* [3] permettent d'évaluer le nombre de défauts de cache (*cache-miss*), et ainsi de mettre en évidence une mauvaise organisation des données.
Certains programmes permettent aussi de visualiser les accès aux données dans la RAM, mais l'utilisateur doit alors déterminer lui-même quels emplacements mémoire correspondent à quelles données.

2.2.2.5 Tableau, vecteur ou liste ?

Les tableaux, les std::vector et les std::list utilisent différemment l'espace mémoire (voir figure 2.6).

Les tableaux sont très indiqués pour stocker des données de types simples et vectorisables (float, int, *etc*). Leur compréhension est intuitive car tous les éléments se traitent de la même façon. Le CPU et le compilateur sont bien plus efficaces pour traiter des tableaux à une seule dimension. Or, les analyses de physique utilisent des matrices et des tenseurs.

Si l'analyse a besoin d'une matrice $N \times M$, il est préférable d'allouer un tableau de taille $N \times M$ et de le traiter comme une matrice. Ainsi, un élément à la i-ème ligne et à la j-ème colonne sera obtenu avec une adresse $i \times M + j$ si les données sont stockées en lignes (à la manière du C) ou $j \times N + i$ si les données sont stockées en colonnes (à la manière du Fortran).

Certaines propriétés, indispensables pour une vectorisation efficace (voir chapitre 3), peuvent également être ajoutées.

Il faut cependant être vigilant lorsque l'on utilise des tableaux en C ou en C++ afin d'éviter les fuites mémoire. Des outils comme Valgrind [3] permettent de les détecter.

Les std::vector stockent des données quelconques sur une seule dimension. Les données stockées dans des std::vector en C++ 98 sont contiguës par bloc, les données des std::vector à partir de C++ 0 sont contiguës à la manière d'un tableau. Ils sont particulièrement efficaces lorsque de nouvelles données doivent être concaténées aux précédentes (une classe événement par exemple). Mais l'insertion de données au centre du tableau coûte bien plus cher que leur concaténation. Il est également difficile de les utiliser pour vectoriser des calculs (voir chapitre 3).

Les std::list offrent une très grande souplesse de manipulation de données au prix d'un surcoût important dans les tâches de recherche et de lecture. Elles sont efficaces pour de

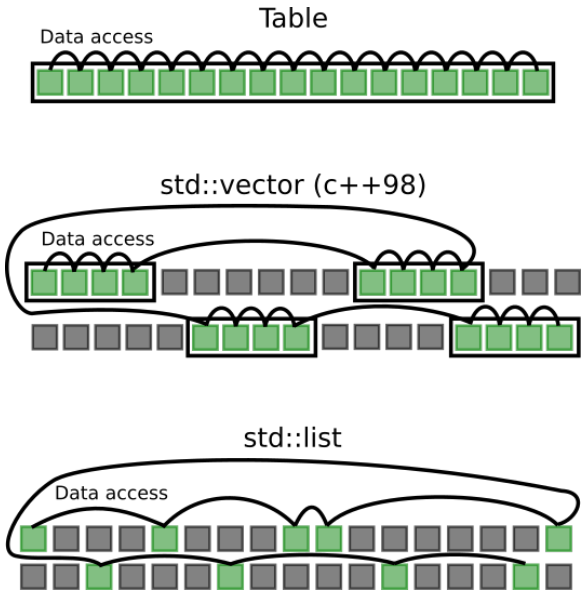

FIGURE 2.6 – Différences de stockage entre les tables, les vectors (C++98) et les lists.

petits volumes de données (comme des configurations) mais leur performance se dégrade rapidement quand le volume de données augmente.

2.2.3 Optimisation du calcul

Les différentes optimisations décrites dans cette section ne sont efficaces que si le format de données est adapté au HPC (voir section 2.2.2). Une dernière optimisation est à ajouter à la liste suivante : la vectorisation, qui sera traitée dans le chapitre 3.

2.2.3.1 Bibliothèques bas niveau

Certaines bibliothèques telles que MKL [9], BLAS [10], ATLAS [11], LAPACK [12], LAPACK++ [13] ont été avant tout conçues pour être très rapides, au prix d'une interface peu ergonomique. Il peut être intéressant de les utiliser dans des cas appropriés (voir section 3.4.3).

2.2.3.2 Bibliothèques haut niveau

D'autres bibliothèques C++ comme EIGEN [14], ARMADILLO [15] ou HPX [16] fournissent des objets mathématiques simples à manipuler comme les vecteurs, les matrices ou les tenseurs (voir section 3.4.4). Cela permet d'optimiser directement les calculs du programme sans passer explicitement par des fonctions bas niveau, ce qui facilite l'écriture et la lecture de ces programmes.

2.2.3.3 Changer d'algorithme

Si l'algorithme utilisé est lent, ou qu'il n'est pas possible de l'optimiser davantage, il faut envisager de changer d'algorithme.

Dans ce cas, les chercheurs doivent travailler en collaboration avec les ingénieurs. Les ingénieurs devront trouver un nouvel algorithme plus performant ou une variante plus rapide d'un algorithme existant et les chercheurs devront vérifier que ce nouvel algorithme produit bien des résultats de physique cohérents.

2.2.3.4 Effectuer moins de calculs

Une variante de la situation précédente consiste à garder l'algorithme de départ tel quel, mais à modifier le nombre de fois qu'il est appelé.

Par exemple, en astrophysique, on peut trier des données décrivant des sources galactiques par ascension droite et déclinaison, afin de n'appliquer l'algorithme utilisé qu'aux endroits de la carte du ciel les plus pertinents.

Un algorithme de départ peut aussi être raffiné en plusieurs sous-algorithmes, appelés chacun leur tour sur un sous-ensemble des données d'entrées filtré selon un critère de pertinence (quantité de signal reçu, forme de la source à étudier, *etc.*).

Il peut aussi arriver qu'une expression mathématique contienne de nombreux zéros qui pourraient tout simplement être ignorés. Lorsque c'est le cas dans des matrices, on parle de matrices creuses.

Enfin, il peut aussi arriver qu'un programme effectue un calcul coûteux plusieurs fois de suite, auquel cas on gagnera à conserver le résultat original plutôt que de recalculer. Cependant, cette approche augmente aussi l'empreinte mémoire du programme, il faut donc équilibrer entre ce coût et le bénéfice attendu.

2.2.3.5 Réduire l'empreinte mémoire

Une consommation mémoire trop importante peut fortement ralentir l'exécution d'un programme. Dans les cas extrêmes, l'ordinateur peut être amené à utiliser fortement l'espace d'échange, ou SWAP [4], car la mémoire RAM est saturée. Mais comme cet espace de stockage est situé sur le disque dur, les accès aux données qui y sont sauvegardées sont extrêmement ralentis.

Note 3 –

Il suffit qu'un programme consomme trop de mémoire sur un nœud de calcul pour ralentir tous les autres jobs. Ce problème est donc étendu à tous les programmes utilisés sur des centres de calcul. Pour éviter ce ralentissement généralisé, certains centres de calcul interrompent brutalement les calculs dont la consommation de mémoire dépasse un certain seuil.

4. L'espace d'échange, ou SWAP, est une région du disque dur de l'ordinateur qui est utilisée en substitut de la RAM pour pouvoir continuer les calculs lorsque celle-ci est pleine.

On peut réduire l'empreinte mémoire en évitant de stocker des données dont on n'a pas besoin, ou bien qui sont imposantes mais rapides à recalculer lorsqu'on en a besoin.

Il faut aussi s'assurer qu'on ne stocke pas accidentellement plusieurs exemplaires de la même donnée, ce qui arrive facilement dans des langages basés sur la sémantique de copie comme C++. Si c'est le cas, il vaut mieux désigner un exemplaire maître de la donnée et transmettre des pointeurs/références aux parties du programme qui en ont besoin.

Un autre problème, qui survient notamment dans les langages de programmation où la mémoire est gérée manuellement, est la « fuite mémoire », où l'on oublie de libérer de la mémoire préalablement allouée qui n'est plus utilisée.

On doit également éviter d'effectuer un grand nombre de petites allocations (ex : une allocation pour un seul nombre flottant) car chaque allocation est associée à des données de gestion pour le système, qui peuvent dans ces cas extrêmes peser plus lourd en mémoire que le contenu de l'allocation. Mieux vaut mutualiser une grande allocation en regroupant les données sous forme de structures et/ou de tableaux. Cela permettra aussi de diminuer le temps passé dans les fonctions d'allocations et d'améliorer la localité des données.

En regroupant plusieurs tableaux au sein d'une classe, qui exposera des fonctions permettant de les manipuler ensemble, il sera possible à la fois de mutualiser les allocations et d'augmenter la clarté du programme.

2.2.3.6 Supprimer certain branchements

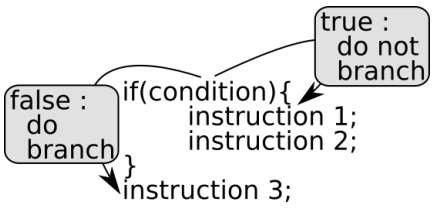

FIGURE 2.7 – Branchement dû à une condition dans un programme C/C++.

Un branchement est une instruction qui fait sauter le programme à un autre endroit du code. On parle de branchement conditionnel lorsque le branchement s'effectue ou non selon la valeur d'un booléen, appelé condition.

Le mot-clé **if** des langages de programmation se traduit généralement par un branchement conditionnel au niveau matériel, voir figure 2.7. Si la condition est vraie, alors le programme continue d'exécuter les instructions internes au bloc derrière le **if**. Si la condition est fausse, en revanche, le programme sautera aux instructions correspondantes et poursuivra l'exécution du programme après le bloc.

Ce mécanisme n'est pas anodin, car il interfère avec une optimisation de performance interne du CPU, le *pipelining*.

L'exécution de chaque instruction CPU se décompose en plusieurs étapes, pour simplifier on peut considérer 5 étapes (voir figure 2.8) :

1. **IF, Instruction Fetch** : Aller chercher l'instruction à exécuter en RAM.
2. **ID, Instruction Decode** : Décoder l'instruction.
3. **EX, EXecute** : Exécuter l'instruction.
4. **MEM, MEMory** : Échanger avec la mémoire.
5. **WB, Write Bytes** : Écrire le résultat.

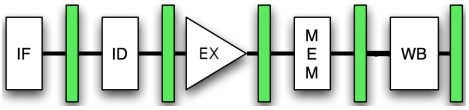

FIGURE 2.8 – Différentes étapes d'un pipeline CPU.

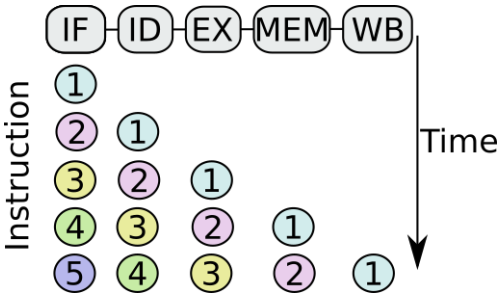

FIGURE 2.9 – Utilisation du pipeline CPU en fonction du temps.

Les premiers CPU n'exécutaient qu'une étape à la fois. Mais dans la mesure où chaque étape est gérée par un composant différent du processeur, les fabricants de matériel ont rapidement commencé à faire fonctionner ces composants en même temps.

L'idée était de traiter les différentes étapes de traitement comme des parties d'une chaîne de montage industrielle, un *pipeline* (voir figure 2.9).

Les instructions entrent dans le pipeline à tour de rôle. Pendant que la première instruction est en train d'être décodée, la deuxième est rapatriée. Pendant que la première instruction est exécutée, la deuxième est en train d'être décodée et la troisième d'être rapatriée. Et ainsi de suite.

Une fois que le pipeline a traité les 5 premières instructions, il fonctionne à plein régime et fournit des résultats en sortie au même rythme que les instructions arrivent en entrée.

Si l'on suppose que chaque étape de traitement d'une instruction prend le même temps, cela représente une accélération d'un facteur 5 par rapport au cas où le CPU attend qu'une instruction soit complètement traitée avant d'aller chercher la suivante.

Toutefois, les choses se compliquent lorsque le CPU rencontre une condition. En effet, le CPU ne peut pas, *a priori*, précharger l'instruction suivante, car il ne peut pas deviner si la condition va être vraie ou fausse, et ne peut donc pas savoir quelle est l'instruction suivante à charger.

Comme l'utilisation du pipeline CPU a permis d'accélérer considérablement les performances, une solution a été développée pour atténuer le problème des branchements : le prédicteur de branchement.

Un prédicteur de branchement est un automate à quatre états qui détermine si un branchement a une probabilité plus importante d'être pris ou non (voir figure 2.10).

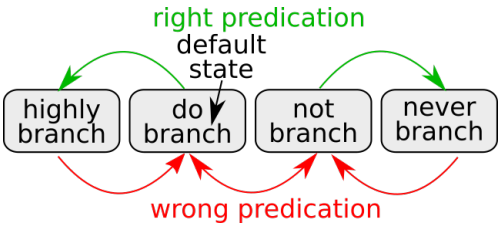

FIGURE 2.10 – Fonctionnement d'un prédicteur de branchement.

Le prédicteur prédit initialement que les branchements auront lieu, ce qui est vrai dans 60% des cas à cause des boucles.

À chaque fois qu'un branchement conditionnel est effectué, l'état de l'automate va vers une plus grande certitude que le branchement suivant sera effectué. À chaque fois qu'il n'est pas effectué, il va vers une plus grande certitude que le branchement suivant ne sera pas effectué.

L'existence d'états extrêmes « *highly branch* » et « *never branch* » permet au prédicteur de prendre en compte des exceptions, où un branchement a presque toujours lieu ou bien n'a presque jamais lieu.

Pour chaque condition, le CPU poursuit l'exécution selon l'hypothèse prédite, tout en se préparant à annuler les opérations effectuées s'il s'avérait que la prédiction était fausse.

De nos jours les prédicteurs de branchement sont très performants, ils ont un taux de

bonnes réponses supérieur à 95%. Mais les quelques pourcents restant peuvent être problématiques pour les performances.

Lorsque le prédicteur de branchement donne une bonne réponse, le CPU va chercher les bonnes instructions et le programme n'est pas ralenti. En revanche, lorsqu'il se trompe, les instructions préchargées ne sont pas les bonnes. Il faut donc vider le pipeline et le remplir de nouveau.

Or, les pipelines des CPUs actuels sont composés de 40 à 80 étapes, chacune prenant plusieurs cycles d'horloge. Vider et remplir le pipeline coûte environ 300 cycles.

Si cela se produit trop fréquemment, on peut y dépenser une fraction conséquente du temps total de calcul. C'est ce qui s'est produit dans le traitement d'image historique des analyses de type CTA.

Un exemple de performance est donné dans l'annexe A.

2.3 Impact du cycle de développement sur les optimisations

Le développement de fonctionnalités optimisées dans un programme implique que des développeurs spécialistes doivent passer plus de temps sur certaines fonctions. L'impact de ce processus sur le temps de développement du programme peut être réduit en focalisant cet effort sur les fonctions les plus coûteuses en temps (dite fonctions chaudes), qui sont généralement peu nombreuses.

Il est généralement admis qu'il n'est préférable d'optimiser un programme qu'une fois que ses résultats sont cohérents (donc après la phase de débogage), mais cette affirmation doit être nuancée dans la mesure où la conception du programme doit aussi permettre son optimisation.

En effet, si de mauvais choix ont été faits trop tôt dans la phase de développement, le programme ne pourra pas être optimisé, ou ne pourra être optimisé qu'au prix d'une réécriture quasi-totale. Dans ces conditions, le travail à fournir sera bien plus important que le temps de développement de la première version, ce qui n'est généralement pas envisageable vu la taille des programmes utilisés par les nouvelles expériences.

De plus, certaines modifications peuvent être lourdes de conséquences. Par exemple, le choix d'un format de données inadapté (voir section 2.2.2), ne pourra être corrigé que par l'écriture d'un nouveau code d'analyse utilisant un format de données adapté, accompagné de convertisseurs qui serviront à mettre à jour les fichiers de données stockés dans l'ancien format. Dans ce cas, non seulement le temps de développement est conséquent, mais le temps de calcul lié aux conversions peut lui aussi être gigantesque.

Dans un autre cas de figure, si l'analyse dépend d'un algorithme qui ne passe pas à l'échelle[5], toute la partie qui dépend de cet algorithme devra être développée de nouveau en utilisant un nouvel algorithme plus performant. Cela peut conduire à des impasses.

5. On dit qu'un algorithme ne passe pas à l'échelle quand son temps de calcul ne diminue pas de façon proportionnelle à la quantité de ressources matérielles (ex : cœurs processeurs) qui sont allouées au calcul.

L'échelle croissante des expériences a conduit naturellement les physiciens à travailler avec des informaticiens.

Les physiciens agissent en utilisateurs d'analyses et d'infrastructures afin d'obtenir des résultats de physiques, loin des considérations matérielles, logicielles, ou en rapport avec le stockage des données de leur analyse. Ce sont des utilisateurs exigeants, et c'est leur droit le plus strict.

Les informaticiens, quant à eux, doivent intervenir là où la complexité des problèmes à résoudre est importante, sans rendre le système incompréhensible pour ses utilisateurs.

Mais l'augmentation exponentielle du volume de données à traiter dans les expériences de physique implique que les informaticiens doivent répondre à des problématiques extrêmement complexes, en devenant de plus en plus experts de leurs domaines respectifs. Masquer la difficulté de la tâche aux yeux des utilisateurs devient donc plus difficile, mais aussi plus nécessaire que jamais.

De ce point de vue, le travail des informaticiens est bien plus important que précédemment et le sera davantage aux vu des défis de calcul liés aux nouvelles expériences.

Cette responsabilité va au-delà du fait de fournir des analyses efficaces et optimisées. En effet, le physicien qui doit lui-même développer une partie de son analyse de données s'inspirera des programmes qui lui sont fournis par les informaticiens. Donc, ces derniers doivent fournir des exemples irréprochables afin que les physiciens puissent partir d'une base saine.

Cette responsabilité pourrait s'étendre aux physiciens eux-mêmes, qui sont tenus de fournir des programmes corrects et performants à leurs pairs, par exemple, dans le cadre de la vérification d'un résultat ou de la poursuite d'une idée prometteuse.

Cette préoccupation de viser, non pas la solution en apparence simple et peu efficace à court terme, mais bien la solution la plus simple et efficace à long terme est le sujet de la formation IN2P3 « Qualité : faire simple et utile » [17].

Le processus d'optimisation amène aussi à une relecture approfondie du code, qui peut amener à corriger des erreurs dans celui-ci et à entamer une réflexion approfondie sur la précision numérique requise. Par conséquent, contrairement à une idée reçue, un programme optimisé ne sera pas forcément moins précis que le programme optimisé (voir chapitre 8).

La prise en compte de ces différents conseils permettra aux physiciens et aux informaticiens de travailler plus efficacement ensemble.

Les physiciens pourront, sans s'engager dans un processus d'optimisation extrêmement poussé, obtenir d'ors et déjà des analyses plus rapides qu'ils n'en écrivent actuellement.

Les informaticiens spécialisés en optimisation seront chargés d'accélérer ces analyses d'un facteur 10 à 20 en se concentrant sur des problèmes difficiles, et non plus d'obtenir des programmes plusieurs centaines de fois plus rapides par une réécriture quasi-totale du code, car le code produit par les physiciens sera plus efficace d'emblée.

L'essentiel du chapitre 2 –

L'optimisation des programmes d'analyse est indispensable pour contenir la quantité de données des nouvelles expériences de physique additionnée à celles des expériences actuelles. Il existe de nombreuses manières d'optimiser une analyse : utilisation de bibliothèques HPC, changement d'algorithme, vectorisation et bien d'autres. Cependant, cette optimisation ne peut être efficace que si le format de données choisi permet le HPC.

Chapitre 3

Pourquoi et comment vectoriser ?

Sujet du chapitre 3 –

De nos jours, la vectorisation est une méthode incontournable afin d'utiliser au mieux les centres de calculs. Ce chapitre expose les enjeux, montre les différentes manières de vectoriser et termine sur un retour d'expérience ainsi que sur une recommandation pour le long terme.

Sommaire

3.1 Principe de la vectorisation . . . 24
3.2 Pourquoi vectoriser ? . . . 24
3.3 Ce calcul est-il vectorisable ? . . . 25
3.3.1 Propriétés du format données . . . 25
3.3.2 Indépendances des éléments . . . 28
3.4 Comment vectoriser ? . . . 29
3.4.1 Vectoriser avec le compilateur . . . 29
3.4.2 Utiliser des fonctions intrinsèques . . . 30
3.4.3 Les bibliothèques optimisées . . . 31
3.4.4 Les bibliothèques d'auto-vectorisation . . . 32
3.4.5 Les générateurs de code . . . 32
3.5 Ce code a-t-il été vectorisé par le compilateur ? . . . 33
3.6 Retours d'expériences . . . 33
3.6.1 Vectorisation de la simulation CORSIKA . . . 33
3.6.2 Vectorisation de l'analyse de CTA . . . 35
3.6.3 Vectorisation dans Smilei . . . 36
3.7 Recommandations pour le long terme . . . 37
3.7.1 Structure/Stockage de données . . . 37
3.7.2 Bonnes pratiques . . . 37
3.7.3 Vectoriser un programme existant . . . 38
3.7.4 Vectoriser un nouveau programme . . . 38

Instruction Set	Release Date	First CPU	Nb **float** per instruction	Nb **int** per instruction
SSE2	2003	2004	2	1
SSSE3	2004	2005	2	2
SSE4.1	2006	2007	4	2
SSE4.2	2007	2008	4	4
AVX	2008	2011	8	4
AVX2	2012	2013	8	8
AVX 512.1	2013	2016	16	8
AVX 512.2	2015	2018	16	16

TABLE 3.1 – Évolution des capacités de vectorisation des processeurs Intel [18].

3.1 Principe de la vectorisation

Les processeurs actuels peuvent, depuis 2004 (voir table 3.1), effectuer un même calcul sur plusieurs nombres flottants ou entiers en même temps (voir figure 3.1).

Techniquement, la vectorisation est permise par la duplication des unités de calcul au sein des CPU. Sur certains programmes, cette capacité à effectuer N calculs en même temps (où N dépend de l'architecture) peut être exploitée pour rendre les programmes N fois plus rapides.

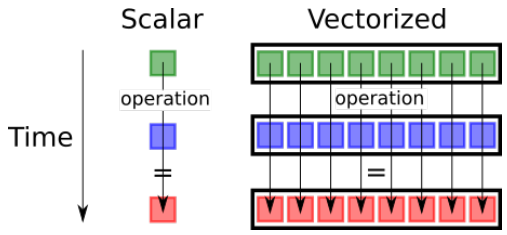

FIGURE 3.1 – **À gauche :** opération scalaire. **À droite :** opération vectorielle.

3.2 Pourquoi vectoriser ?

De nos jours, la quasi-totalité des ordinateurs qui équipent les centres de calcul permettent la vectorisation. Cette vectorisation permet des accélérations allant d'un facteur 2 à un facteur 16, suivant les architectures et les types de données utilisés. On peut en tirer deux conclusions.

D'une part, les centres de calculs actuels ont d'ors et déjà la possibilité d'effectuer les analyses de physique plus rapidement, sans qu'il y ait lieu d'acheter davantage de matériel. Mais pour cela, un effort devra être fourni pour mettre à niveau les analyses qui en ont besoin.

D'autre part, les analyses non vectorisées utilisent moins de 25% des capacités des processeurs des centres de calcul pour les processeurs les plus anciens encore utilisés (SSE 4.2), et moins de 6.25% de celles des processeurs les plus récents. Cette inefficacité n'est pas prise en compte par les algorithmes de calcul de consommation CPU couramment utilisés, qui ne font qu'observer le pourcentage du temps pendant lequel un programme s'exécute sur le CPU, ce qui rend l'utilisation réelle des ressources CPU très difficile à estimer.

On pourrait objecter à cette analyse que tous les algorithmes ne se prêtent pas à la

vectorisation. Mais c'est le cas de tous les calculs qui manipulent de façon régulière des images, des vecteurs, des matrices, des tenseurs... soit une grande majorité des traitements effectués lors des analyses de physique !

Le cas échéant, il est également possible d'étendre astucieusement la vectorisation à des calculs où son utilisation n'est habituellement pas considérée comme viable, comme la recherche par dichotomie ou les parcours d'arbres.

3.3 Ce calcul est-il vectorisable ?

3.3.1 Propriétés du format données

Comme pour les optimisations en général, la vectorisation n'est efficace que sur des formats de données qui satisfont à quelques propriétés.

3.3.1.1 Données contiguës

Nous avons vu dans la section 2.2.2 que l'utilisation de données contiguës est nécessaire pour que les échanges entre le CPU et la RAM soient efficaces. La contiguïté des données joue aussi un rôle crucial en calcul vectoriel.

En effet, les instructions vectorielles ne peuvent généralement être appliquées qu'à des données jointives en mémoire. Ce qui pose problème lorsqu'on stocke les données sous formes de tableaux de structures, comme il est intuitif de le faire en C ou en C++.

Par exemple, dans un tableau de vecteurs tridimensionnels, les coordonnées des vecteurs se retrouveront stockées en mémoire dans l'ordre X_1, Y_1, Z_1, X_2, Y_2, Z_2, *etc.* Il ne sera donc pas possible d'effectuer des opérations vectorielles sur les coordonnées X des vecteurs, car celles-ci ne sont pas stockées de façon contiguë en mémoire.

Pour éviter ce problème, il est souvent nécessaire de remplacer les tableaux de structures par des structures de tableaux (voir figure 3.2).

On conçoit aisément que cette organisation des données va à l'encontre des pratiques usuelles de programmation orientée objet, et qu'un format de données qui a été développé avec trop de classes imbriquées ne permettra pas une vectorisation directe des calculs.

Dans ce cas, les données devront être copiées dans des structures qui permettent la vectorisation. Or, cette copie a un coût, aussi bien en temps qu'en mémoire, ce qui amoindrira l'accélération due à la vectorisation voire rendra celle-ci inutilisable.

3.3.1.2 Données alignées

Lorsqu'on manipule de petits tableaux, en plus d'être contiguës, les données sur lesquelles on effectue un calcul vectoriel doivent aussi être convenablement alignées.

En effet, les registres vectoriels du processeur se comportent de manière similaire aux types scalaires (float, int, *etc.*), à cette différence près qu'ils sont composés de N types scalaires contigus.

Or, un processeur ne peut lire des données efficacement qu'à des adresses mémoires qui sont multiples de leur taille. Par exemple, pour un vecteur de N entiers 32-bits (4 octets chacun), l'adresse mémoire du vecteur doit être multiple de $4N$. Selon les processeurs, violer

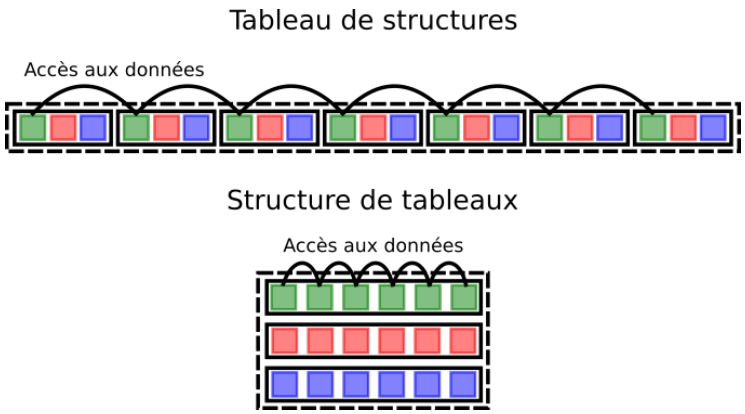

FIGURE 3.2 – Accès aux données dans les structures de tableaux et tableaux de structures.

cette règle aura pour effet soit d'empêcher totalement la vectorisation, soit de la rendre moins efficace [1].

Malheureusement, les allocateurs mémoires comme *malloc* alignent les tableaux sur 16 octets, ce qui convient pour des vecteurs **SSE4**, mais pas pour des vecteurs **AVX** (32 octets) ou **AVX 512** (64 octets).

Dans ces conditions un autre allocateur est nécessaire : *memalign* (ou *posix_memalign* sur MacOS).

Il est également possible d'utiliser des bibliothèques comme ***mallocproxy*** [19], qui change les propriétés de la fonction *malloc* dynamiquement. Dans tous les cas les formats de données doivent donc s'adapter.

Pour illustrer ces problématiques d'alignement, considérons un calcul sur un tableau de 23 éléments (voir figure 3.3).

Un traitement scalaire de ce tableau aurait besoin de 23 calculs. Nous allons supposer dans la suite que la vectorisation permet 8 calculs simultanés (**AVX** pour des float ou **AVX 512** pour des double).

Si le tableau n'est pas aligné, deux calculs vectoriels peuvent être effectués sur des sous-ensembles alignés des données au centre du tableau. Mais les premiers et les derniers éléments du tableau, qui ne peuvent pas remplir un registre vectoriel complet du fait de leurs adresses mémoire, sont traités séparément.

Les premiers éléments du tableau sont traités un par un avec des calculs scalaires : c'est le **peel**. Les derniers éléments du tableau sont également traités un par un avec des calculs scalaires, car il n'y a pas assez de données pour remplir un vecteur : c'est le **tail**. Dans notre exemple, le **peel** est constitué de 4 calculs, et le **tail** de 3. Ce qui implique un nombre de calculs total de 9 dans le cas non-aligné.

Lorsque le tableau est aligné, il n'y a plus de **peel**, 2 calculs vectorisés et 7 calculs de **tail**. Ce qui revient toujours à 9 calculs au total. Dans ces deux cas, l'accélération n'est donc

1. Il existe des instructions intrinsèques (voir section 3.4.2) qui permettent d'initialiser des registres vectoriels avec des données non alignées. Cependant elles sont plus lentes que leurs équivalentes alignées et le sont pas utilisées par le compilateur lorsqu'il vectorise.

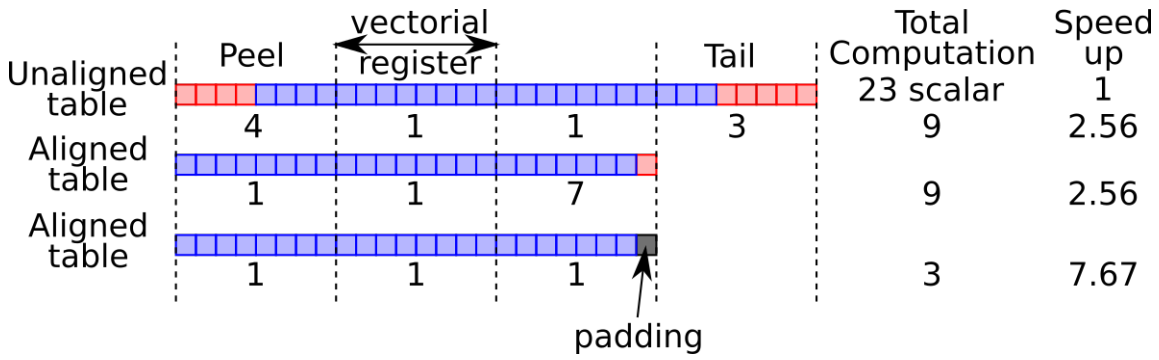

FIGURE 3.3 – Différence d'accélération attendues en fonction de l'alignement des données utilisées et de l'utilisation d'un padding sur un tableau de 23 éléments.

que de 2.56 alors qu'elle devrait idéalement tendre vers 8.

L'utilisation d'un padding [2] consiste à ajouter des valeurs supplémentaires à l'allocation du tableau de données pour que le nombre total d'éléments soit un multiple de la taille des registres vectoriels (dans note cas : 8). Les valeurs des éléments qui composent le padding dépendent du calcul à effectuer. S'il s'agit de multiplication, d'addition ou de soustraction, le padding vaudra 0 mais s'il s'agit d'une division le padding devra valoir 1 [3].

Dans notre cas, comme le tableau contient 23 valeurs, une seule valeur de padding est nécessaire. Cette astuce permettra, au prix d'une allocation légèrement plus grande, de n'utiliser que des calculs vectorisés. Dans notre cas, 3 instructions suffisent désormais, ce qui représente un facteur d'accélération de 7.67.

Certains générateurs de formats de données permettent de créer des tableaux, des matrices ou des tenseurs qui sont alignés en mémoire avec un padding si l'utilisateur le demande [20].

2. Le padding est nommé zero-padding dans la bibliothèque MKL.

3. Les calculs vectorisés ne se limitent pas aux seules opérations de base, mais peuvent utiliser toute sorte de fonctions trigonométriques, exponentielles, *etc.* Dans tous les cas, la valeur du padding doit être une valeur neutre pour ces calculs.

3.3.2 Indépendances des éléments

La vectorisation d'un calcul sur plusieurs éléments d'un tableau, implique que ces éléments doivent être indépendants. Sinon le calcul sur un élément n devra attendre le résultat du calcul sur l'élément $n-1$ (voir figure 3.4).

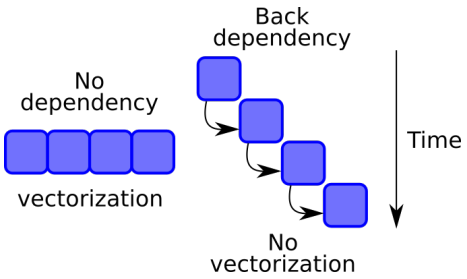

FIGURE 3.4 – **À gauche :** opération complètement vectorisée. **À droite :** opération non-vectorisée due à la présence d'une dépendance arrière entre les éléments.

Dans la plupart des cas, ce problème de dépendance arrière peut être résolu en changeant l'organisation des données ou l'ordre des opérations.

Par exemple, si l'on souhaite effectuer la somme des éléments d'un vecteur, le calcul du résultat pour l'élément n dépend de celui sur l'élément $n-1$. Mais cette dépendance arrière apparente peut être contournée en changeant l'ordre selon lequel les éléments seront additionnés. La figure 3.5 montre comment vectoriser complètement la somme des éléments d'un tableau en utilisant plusieurs accumulateurs différents.

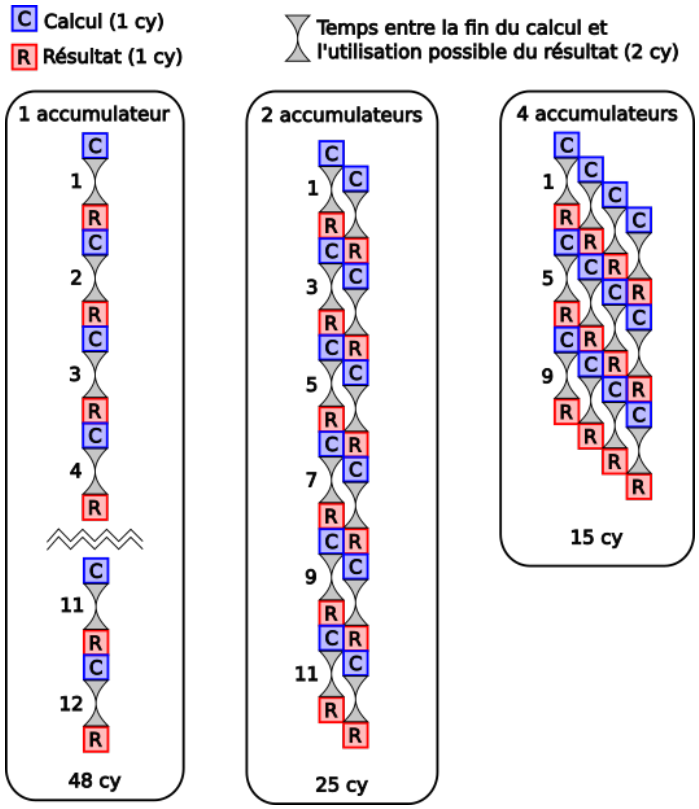

FIGURE 3.5 – Suppression de la dépendance arrière liée à la somme des éléments d'un vecteur (image tirée de [21]).

3.4 Comment vectoriser ?

Il existe de nombreuses méthodes pour vectoriser un calcul.

Cette section décrit ces différentes méthodes et liste leurs avantages ainsi que leurs inconvénients, de la méthode la plus rapide à la plus portable en passant par les différents compromis qu'il est possible de faire. Cette question a été abordée en détail dans le cours [22].

3.4.1 Vectoriser avec le compilateur

Les compilateurs, comme *GCC* et *G++* [23], sont très conservateurs et ne vectoriseront pas les calculs par défaut.

La vectorisation par le compilateur nécessite quelques indices.
Tout d'abord, les tableaux utilisés doivent être alignés, sinon le compilateur ne vectorisera pas les calculs. Pour cela leurs allocations doivent être effectuées avec *memalign* ou *posix_memalign* sur MacOS (voir section 3.3.1.2). De plus, les pointeurs utilisés doivent explicitement être déclarés comme alignés avec la fonction *builtin_assume_aligned*.

Une autre difficulté rencontrée par le compilateur est que les pointeurs C/C++ [24], servent de base aux tableaux dans ces langages, peuvent désigner n'importe quel espace mémoire. Aucune garantie ne permet donc au compilateur de conclure que des tableaux passés en entrée d'une fonction ne se chevauchent pas. Or, cette ambiguïté interdit au compilateur de vectoriser, car en cas de chevauchement la vectorisation change le résultat du calcul.

Le mot clé **__restrict__** garantit au compilateur que les pointeurs désignent des espaces mémoire différents (voir figure 3.6). Dans ce cas, l'utilisateur ne doit pas se tromper car le programme produirait alors un résultat faux.

Certaines fonctions de la bibliothèque standard du C, comme *memcpy* ou *memset*, sont définies avec le mot clé **__restrict__** car le tableau qu'elles utilisent en entrée doit être différent de celui qu'elles utilisent en sortie pour éviter des comportements non souhaités.

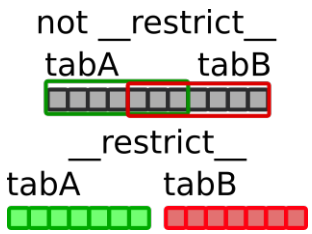

FIGURE 3.6 – **En haut :** si le mot clé **__restrict__** n'est pas utilisé, les espaces mémoires peuvent se chevaucher. **En bas :** si le mot clé **__restrict__** est utilisé, les espaces mémoire sont indépendants.

Pour finir, la vectorisation doit être explicitement activée avec les options de compilation :

— **--ftree-vectorize** : active la vectorisation (inclus dans **-O3**).
— **--march=native** : utilise tous les jeux d'instructions supportés par la machine hôte.
— **--mtune=native** : optimise les performances du code sur la machine hôte.
— **-mEXTENSION** : active un jeu d'instructions vectorielles spécifique.
Où **EXTENSION** : **sse2**, **sse4**, **avx**, **avx2** ou **avx512** [4].

Avantages : les compilateurs deviennent de plus en plus performants pour vectoriser et optimiser des programmes en général. L'analyse pourra voir sont temps d'exécution diminuer rien qu'en changeant de version de compilateur (cependant les accélération seront faibles à chaque fois). Le code en lui-même est facilement portable entre différents matériels, même si le programme final devra être compilé pour chaque architecture cible afin d'avoir les meilleures performances sur des architectures fortement hétérogènes comme la Grille WLCG [25]. Cette procédure est une des plus simples pour vectoriser.

Inconvénients : les compilateurs sont très efficaces pour vectoriser des fonctions simples mais atteignent vite leurs limites lorsque les fonctions gagnent en complexité. Par exemple, le calcul de barycentre à deux dimensions est très mal vectorisé et optimisé par le compilateur GCC [23] (voir cours [22]). Il n'est pas possible d'utiliser de vieux compilateurs comme *GCC 4.8* ou antérieurs car leurs capacités de vectorisation sont trop limitées.

3.4.2 Utiliser des fonctions intrinsèques

Les fonctions intrinsèques représentent la limite entre le langage C et l'assembleur. Ce sont des fonctions C qui contiennent de l'assembleur. Cela permet de programmer quasiment en assembleur (mais en évitant certains inconvénients d'un vrai assembleur) dans un programme C ou C++. Ces fonctions sont très utiles pour effectuer manuellement des optimisations que le compilateur ne sait pas effectuer automatiquement.

Utilisation des fonctions intrinsèques bas niveau

Il est possible d'utiliser directement les fonctions bas niveaux fourni par Intel [18].

Avantages : la programmation (experte) en fonctions intrinsèques permet d'obtenir de meilleures performances que celles atteintes par les compilateurs, car il est possible d'optimiser des algorithmes que le compilateur ne pourra pas optimiser [5].

Inconvénients : le code écrit directement en fonctions intrinsèques ne pourra être exécuté que sur les architectures qui utilisent le même jeu d'instruction (ou un jeu étendu). Il n'est

4. Il existe une vingtaine d'extensions **avx512**, comme **avx512f** pour les calculs flottant par exemple.
5. Ceci est dû à la mécanique interne des compilateurs. Ils utilisent une représentation interne qui décrit le programme comme un graphe et utilisent la théorie des graphes pour effectuer leurs optimisations. Or, ce graphe ne peut être modifié, car cela garanti le fait que le programme fonctionnera de la même manière que celle décrite par le développeur. Ce qui interdit certaines optimisations aux compilateurs.

donc pas complètement portable. Il est nécessaire d'être un expert du sujet pour écrire de telles fonctions.

Note 4 –
Bien que l'écriture de fonctions optimisées à l'aide des fonctions intrinsèques soit difficile, il est possible pour un physicien averti de lire et de comprendre de telles fonctions.

Utilisation des fonctions intrinsèques dans une bibliothèque

Certaines bibliothèques, (***xsimd*** [27], ***PLIBS 9*** [28]) fournissent une abstraction des fonctions intrinsèques. Cela permet de programmer avec des fonctions quasi-intrinsèques qui seront transformées en fonctions intrinsèques réelles lors de la compilation. Les programmes qui utilisent cette méthode voient leur portabilité améliorée, puisqu'une seule version du programme permet de calculer sur toutes les architectures supportées par la bibliothèque utilisée.

Avantages : cette méthode atteint les mêmes performances qu'une utilisation des vraies fonctions intrinsèques, tout en améliorant la portabilité du programme.

Inconvénients : il se peut que la durée de la compilation augmente suivant les bibliothèques utilisées. Il est toujours nécessaire d'être un expert du sujet pour utiliser ces bibliothèques. L'utilisation d'une bibliothèque tierce nécessitera un maintien de la compatibilité entre celle-ci et le programme désiré.

Note 5 –
Comme précédemment, un physicien averti peut lire et comprendre de telles fonctions.

3.4.3 Les bibliothèques optimisées

Les bibliothèques MKL [9], BLAS [10], ATLAS [11], LAPACK [12], LAPACK++ [13] fournissent des fonctions de calcul d'algèbre linéaire très bien optimisées pour toutes sortes d'architecture.

Avantages : Elles permettent d'obtenir des performances proches de la performance crête des machines utilisées et sont portables. La nomenclature des fonctions est relativement standard pour toutes ces bibliothèques.

Inconvénients : Elles sont difficiles à utiliser du fait du nombre assez important de paramètres à passer à leurs fonctions et au fait que ces derniers ne sont pas très explicites.

Les fonctions fournies par des bibliothèques sont de très bas niveau et traitent des problèmes simples d'algèbre linéaire. L'optimisation d'une analyse de physique avec de telles bibliothèques ne sera pas aussi optimale qu'en optimisant les algorithmes spécifiques à cette analyse.

3.4.4 Les bibliothèques d'auto-vectorisation

Certaines bibliothèques (***Eigen*** [14], ***Armadillo*** [15], ***HPX*** [16], ***XTensor*** [29]) utilisent la flexibilité du langage C++ [6] afin de simplifier et vectoriser les calculs à effectuer. .

Avantages : ces bibliothèques permettent de manipuler des objets Vector, Matrix, Tensor qui facilitent l'écriture et la lecture des programmes. Cette simplicité leur permet de rivaliser en termes de performance avec les bibliothèques les plus rapides comme ***MKL*** [9], ***BLAS*** [10], ***ATLAS*** [11], ou ***LAPACK*** [12].

Inconvénients : ces bibliothèques ont leurs spécialités mais ne sont pas compatibles entre elles. ***Armadillo*** [15] est très performante pour l'utilisation de petites matrices, tandis que ***Eigen*** [14] sera plus efficace sur les grandes matrices et ***HPX*** [16] permettra un calcul distribué pour de très grandes matrices [7]. Le temps de compilation et la consommation mémoire pendant la compilation sont bien plus importants qu'avec des bibliothèques plus classiques :

— ***Eigen*** : 12 secondes pour compiler un fichier de 100 lignes, utilise 400 Mo de RAM
— ***HPX*** : 4 Go de RAM suivant les utilisations

3.4.5 Les générateurs de code

Les générateurs de code permettent, à partir d'un programme écrit dans un langage spécifique (*Domain Specific Language*, DSL) ou d'un langage donné (C++, C, Python), d'écrire du code C++, CUDA [8] qui sera parfaitement optimisé pour une architecture donnée. Les générateurs de code les plus utilisés partent du langage Python :

— TensorFlow [31]
— Loopy [32]

D'autres, comme ***PLIBS 8 Kernel Shadok*** [33], utilisent un langage spécifique.

Avantages : l'utilisateur peut écrire son programme sans se préoccuper de l'architecture cible et obtenir de bonnes performances.

6. Le C++ est un langage qui permet l'utilisation des **template** afin de créer des classes et des fonctions génériques. Cette fonctionnalité a été détournée pour créer les **expression template** [30], qui permettent de programmer d'une façon qui forcera le compilateur à optimiser et vectoriser naturellement.
7. Les tests de performances sont généralement donnés sur des matrices mais l'utilisation de vecteurs ou de tenseurs en général est possible.
8. Langage développé par NVidia pour le calcul sur leurs GPUs.

Inconvénients : le fonctionnement du programme dépend d'un autre, qui devra être suivi et maintenu. Cela rajoute une étape supplémentaire pour la maintenance.

Note 6 –
Cette approche est de plus en plus utilisée dans des domaines qui utilisent massivement des GPUs (comme le **Deep Learning**), ou dans le domaine de la simulation.

3.5 Ce code a-t-il été vectorisé par le compilateur ?

On peut estimer grossièrement si le compilateur a vectorisé un code avec des tests de performances. Des programmes comme *time* [34] donnent le temps d'exécution d'un programme cible avec les contributions globales, spécifiques à l'OS et à l'utilisateur. On peut utiliser le programme *Maqao* [5] pour analyser si le binaire contient des instructions de vectorisation, combien il en contient, et ce qu'il faudrait faire pour obtenir de meilleures performances. Les développeurs de LHCb ont également mis au point une version modifiée de *valgrind* permettant de mesurer le pourcentage des calculs qui sont vectorisés pendant l'exécution d'un programme.

3.6 Retours d'expériences

3.6.1 Vectorisation de la simulation CORSIKA

3.6.1.1 Introduction à la simulation CORSIKA

CORSIKA [35] (*COsmic Ray SImulations for KAscade*) est un logiciel pour la simulation Monte Carlo des gerbes atmosphériques induites par des rayons cosmiques de haute énergie. Initialement développé au début des années 90 pour l'expérience KAscade à KIT (*Karlsruhe Institute for Technology*), ce logiciel a été rapidement adapté pour les cas d'utilisation d'autres expériences. Aujourd'hui CORSIKA est le logiciel de référence pour les expériences dans les domaines des rayons cosmiques, astronomie gamma et neutrino (*e.g.* Auger, CTA, IceCube, *etc.*). En particulier, CORSIKA simule de manière détaillée le développement des cascades électromagnétiques et hadroniques en prenant en compte les différents processus d'interaction, perte d'énergie, production de particules secondaires ainsi que la propagation des particules dans l'atmosphère.

CORSIKA fournit un programme principal, écrit en Fortran 77, qui gère la pile des particules à propager, le transport des particules et la description de l'atmosphère. Les processus physiques sont ensuite implémentés dans des modules externes. Dans le cas d'utilisation de l'astronomie Cherenkov, un module supplémentaire IACT/*atmo* [36], écrit en C, implémente la géométrie 3D des télescopes Cherenkov, une description détaillée de l'atmosphère ainsi que la propagation des photons Cherenkov jusqu'aux télescopes. Au total CORSIKA contient plus de cent mille lignes de code.

CTA (*Cherenkov Telescope Array*) [37], le futur observatoire de rayons gamma, utilise également CORSIKA pour ses simulations. Pendant sa phase de préparation, le consortium CTA a effectué plusieurs campagnes de simulation sur la grille de calcul EGI afin de définir le design global de CTA (nombre et dimension des télescopes), puis de guider certains choix stratégiques comme la sélection des meilleurs sites d'implantation [38] et la définition de la géométrie optimale des réseaux [39]. Ces productions utilisent typiquement 8000 cœurs simultanément, distribués sur une vingtaine de sites de la grille. En phase d'opération, des simulations seront nécessaires afin de calculer, pour chaque intervalle de temps étudié, les fonctions de réponse des télescopes. Le temps de calcul associé aux simulations est et restera très important, i.e. environ 200 millions d'heures CPU HS06 par an selon les estimations actuelles.

Dans ce contexte, l'optimisation de CORSIKA permet de produire des fonctions de réponse dans un délai plus court (intéressant pour la science des évènements transitoires), ou de réduire les incertitudes statistiques et systématiques sur chaque fonction de réponse en augmentant la statistique des évènements simulés et en élargissant l'espace des paramètres considérés ainsi que leur granularité.

3.6.1.2 Retour d'expérience

Afin de déterminer la meilleure stratégie d'optimisation, un profilage du programme a été effectué en utilisant plusieurs outils [40] : *Linux perf*[9], GNU *gprof*[10], *Callgrind*[11] et gdb[12] ainsi que différents outils de visualisation : *gprof2dot*[13], *Flame Graph*[14]. Les résultats de ces profilages sont compatibles entre eux et montrent que la plupart du temps CPU (82%) est utilisé par la fonction *cerenk* en charge de la production et propagation des photons Cherenkov. En outre, la majorité du CPU dans cette fonction est consommé par des appels à des fonctions mathématiques (exp, sin, cos, *etc.*).

Grâce au profilage, un informaticien n'ayant pas de connaissance spécifique du programme est capable d'identifier les parties du programme qui consomment le plus de CPU et sur lesquelles il faut potentiellement focaliser l'effort d'optimisation. Néanmoins l'interaction entre informaticiens et physiciens experts de CORSIKA a permis d'identifier très rapidement les pistes d'optimisation les plus prometteuses. En particulier, le profilage indique qu'on devrait obtenir un gain de performance en optimisant la fonction *cerenk*. Ensuite, l'observation que *cerenk* effectue des calculs sur chaque photon de manière indépendante indique qu'il serait éventuellement possible de vectoriser cette fonction.

Pour la vectorisation de *cerenk*, nous avons choisi l'approche de l'auto-vectorisation. Cette approche consiste à réécrire certaines boucles et à transformer le code de manière que le compilateur soit capable de détecter les parties du code vectorisables. Un des avantages de cette approche réside dans le fait d'obtenir un code portable, sans dépendance de bibliothèques externes et relativement lisible. Enfin, un code vectorisé s'exécute d'or et déjà efficacement

9. https://perf.wiki.kernel.org/index.php/Main_Page
10. https://sourceware.org/binutils/docs/gprof/
11. http://valgrind.org/docs/manual/cl-manual.html
12. https://poormansprofiler.org/
13. https://github.com/jrfonseca/gprof2dot
14. http://www.brendangregg.com/flamegraphs.html

sur la grille de calcul, car la plupart des processeurs dans les centres supportent les instructions vectorielles. Compte tenu du temps CPU utilisé dans les appels à des fonctions mathématiques (exp, sin, cos, asin, *etc.*) une autre optimisation très efficace a été obtenue en utilisant des bibliothèques mathématiques vectorisées [41] dans la fonction *cerenk*. Le gain de performance globale obtenu grâce à la vectorisation est un facteur d'accélération 1.52.

Un point crucial pour les physiciens concerne la validité des résultats obtenus avec la version du code optimisé. En particulier, l'utilisation des bibliothèques mathématiques vectorisées peut en principe produire des résultats numériques légèrement différents. Or, dans ce cas spécifique, nous avons vérifié que la version optimisée produit les mêmes résultats que la version originale. En outre, il a été montré dans [42] que dans le cas des applications typiques en physique de particules, l'impact sur la précision numérique dû à l'utilisation de bibliothèques mathématiques vectorisées est négligeable. Enfin, des tests préliminaires montrent que la version de CORSIKA vectorisée s'exécute efficacement sur la grille. Des productions à plus grande échelle sont également prévues afin de confirmer ce résultat.

3.6.2 Vectorisation de l'analyse de CTA

3.6.2.1 L'expérience CTA

Le projet *Cherenkov Telescope Array* (CTA [43]) sera le plus grand observatoire d'astronomie gamma au sol en 2021. Il aura pour but d'améliorer d'un ordre de grandeur la sensibilité aux rayons gamma et d'étendre l'intervalle de détection en énergie par rapport aux expériences précédentes. Il sera composé de deux sites, un site Nord à La Palma en Espagne composé de 19 télescopes et un site sud dans le désert du Paranal au Chili composé de 100 télescopes.

Une fois terminés, ces télescopes produiront une quantité inégalée de données pour une expérience d'astronomie gamma. Le flux de données brute sera théoriquement de 192 GO/s et sera réduit à 27 GO/s pour le site sud et 17 GO/s pour le site nord, du fait de leur déclenchement par stéréoscopie.

Ce flux de 210 PO par an sera réduit sur place à 4 PO par an qui seront envoyés dans quatre centres de calcul et de stockage Européens.

3.6.2.2 Retour d'expérience

L'optimisation et plus particulièrement la vectorisation de l'analyse de données de CTA a permis d'obtenir un facteur d'accélération final supérieur à 700 par rapport aux analyses des expériences précédentes (voir [21]).

Ce facteur d'optimisation, inattendu, a réduit drastiquement les besoins en calculs pour l'analyse en temps réel de cette expérience. Si bien qu'initialement plusieurs centaines d'ordinateurs étaient prévus pour répondre à la demande en calcul, là où maintenant seules deux machines partagées avec l'acquisition de données suffisent pour analyser le flux de données en temps réel.

Cette analyse en temps réel n'aura pas non plus besoin d'être simplifiée à outrance, et fournira donc des résultats de physique plus pertinents qu'une analyse en temps réel classique.

De plus, l'utilisation d'un nombre réduit de machines implique une forte simplification de la gestion des fautes. En effet, l'utilisation de centaines de machine implique une surcouche

logicielle adaptée pour garantir une tolérance aux fautes, ce qui complique énormément les programmes d'analyse. La réduction de cette couche logicielle simplifie grandement le développement de cette analyse [15].

Tous les cœurs de calcul économisés pourront servir à la réduction des données durant la journée. Des simulations par observations ont mêmes été envisagées. L'analyse, sur site mais non temps réel, pourra également utiliser plus de cœurs, ce qui améliorera l'échéance des résultats, avec une physique plus précise que celle de l'analyse en temps réel.

D'un point de vue plus technique, l'optimisation et la vectorisation de cette analyse ont été grandement facilitées par l'utilisation de générateurs de code, même si la version finale du programme d'analyse n'utilise plus ces générateurs. Ils nous ont permis d'explorer très efficacement la vectorisation par des fonctions intrinsèques, et ont aidé à déterminer quelle instruction de préchargement des données était la plus efficace pour nos calculs.

Des générateurs de format de données [20] ont également facilité l'exploration des propriétés indispensables d'un format de données HPC.

3.6.3 Vectorisation dans Smilei

3.6.3.1 Le code Smilei

Smilei est un code *Particle-In-Cell* (PIC) explicite écrit en C++. Il sert à simuler la physique cinétique des plasmas. Les domaines d'applications sont très vastes et vont de l'astrophysique (jets relativistes, supernovae, vents solaires...) aux plasmas de laboratoires. À l'IN2P3 il est utilisé pour la simulation d'expériences d'interaction laser-plasma. Il permet de préparer et d'interpréter les expériences mais aussi de mieux comprendre la physique en jeu. C'est un code massivement parallèle (***MPI*** + ***OpenMP***) destiné en premier lieu aux super-calculateurs.

Smilei utilise d'une part des schémas aux différences finies pour résoudre les équations de Maxwell sur un maillage et d'autre part des macro-particules échantillonnent l'espace des phases du plasma et sont libres de se déplacer dans tout le domaine de simulation. Le maillage est un ensemble de points immobiles et parfaitement ordonnés. Les particules, en revanche, sont a priori complètement désordonnées. Néanmoins, le déplacement des macro-particules génère les courants électriques qui sont les termes sources des équations de Maxwell, couplant ainsi les quantités définies sur le maillage et les particules. La difficulté de la vectorisation des codes PIC réside dans ces interactions entre deux structures très différentes.

3.6.3.2 Retour d'expérience

Smilei est un code open source utilisé par une communauté internationale assez vaste et contrastée. Les types de cas physiques simulés et les systèmes utilisés pour faire tourner le code peuvent être très différents, les problématiques de portabilité sont donc une priorité. Il faut également prendre garde à ce que les optimisations apportées au code pour une certaine

15. Une analyse simple contiendra naturellement moins de bogues et sera plus facile à corriger et à améliorer sur le long terme, par un plus grand nombre de personnes.

gamme de paramètres ne soit pas pénalisantes pour d'autres régimes. Ainsi, pour la vectorisation, nous avons décidé d'utiliser principalement l'auto-vectorisation (vectorisation par le compilateur voir 3.4.1).

Adopter des structures de données appropriées et donner quelques indices au compilateur(#pragma openmp simd, *memalign*,...) pour quelques boucles récalcitrantes suffit à avoir de bonnes performances dès lors que l'algorithme se prête bien à la vectorisation. Malheureusement, ce n'est pas le cas pour les parties les plus coûteuses d'un code PIC optimisé standard, même en 2019. Le gros du travail a donc été pour nous de réécrire les algorithmes afin de les rendre auto-vectorisables. Ces algorithmes sont a priori moins performants sur des machines complètement scalaires (sans vectorisation), mais adossés à une technique de tri de particules adéquate, ils permettent néanmoins un gain d'un facteur 2 environ sur les machines qui possèdent des instructions vectorielles.

Le choix a été fait de vectoriser sur les particules, car ce sont elles qui portent la plus grosse charge de calcul dans la plupart des cas. Cependant, il peut arriver que le nombre de particules soit trop faible pour remplir un registre vectoriel, les performances devenant alors moins bonnes qu'avec l'algorithme scalaire standard. Pour palier à ce problème, nous avons mis au point une technique de choix local de méthode de calcul. À l'exécution, et dynamiquement dans le temps et l'espace, le logiciel est capable d'évaluer quelle méthode sera la plus efficace et l'applique.

Ces travaux ont été publiés dans [44].

3.7 Recommandations pour le long terme

3.7.1 Structure/Stockage de données

Comme nous l'avons vu dans la section 3.3.1, le format de données est crucial pour garantir une vectorisation efficace. Cela inclut les données stockées sur disque dur ou bande magnétique. Un format de données non adapté au HPC détériorera les vitesses de lecture et d'écriture de ces données, ce qui ralentira globalement le programme, sans que la vectorisation améliore ses performances.

Le format de données HDF5 [45] est particulièrement efficace et permet un stockage de données standard et aussi modulaire qu'on le souhaite. Ce format est aussi auto descriptif, ce qui implique que n'importe quelle bibliothèque compatible HDF5 peut lire ces fichiers.

Les fichiers peuvent être partiellement lus s'ils sont trop volumineux pour la RAM. Cette fonctionnalité est à la discrétion du développeur qui doit utiliser un découpage efficace de ses données.

3.7.2 Bonnes pratiques

Les bonnes pratiques liés à la vectorisation sont surtout liées à l'utilisation de tableaux alignés, et de structures de données qui permettent la vectorisation. D'une manière générale, les bonnes pratiques de développement (voir chapitre 9) sont d'un très grand secours lorsqu'un programme doit être vectorisé ou optimisé en général. La simplicité et la clarté sont les meilleures garanties pour qu'un programme puisse être optimisé efficacement.

3.7.3 Vectoriser un programme existant

Un programme déjà développé, voire déjà en production, est potentiellement ancien, d'autant que les technologies d'optimisation ont beaucoup évolué ces dernières années.

Dans tous les cas, même pour un programme assez récent, l'optimisation ou la vectorisation de celui-ci va se heurter aux choix qui ont déjà été pris par les développeurs (qu'ils soient toujours présents ou non dans le projet).

Si ces choix ne prenaient pas en compte la vectorisation ou l'optimisation de ce programme, cette tâche va potentiellement s'avérer extrêmement difficile. Il est entendu par là que l'effort apporté pour la vectorisation de ce programme devra être important, sans la garantie d'une bonne accélération en fin de compte.

En effet, si les structures de données ont été conçues avec des classes sans un découpage adéquat [16], la vectorisation sera impossible avec le compilateur et inefficace manuellement, sauf à changer complètement l'ordre des données pour une vectorisation optimale.

Ces erreurs de développement imposent souvent de réécrire une nouvelle version en partant de zéro afin de maîtriser les calculs, les structures de données et leurs optimisations.

Il est cependant possible d'utiliser des tableaux temporaires pour recueillir les données de ces « structures inefficaces » et ainsi vectoriser efficacement ces calculs. Si les accélérations obtenues sont acceptables, elles seront loin de celles obtenues en réécrivant totalement le programme en question

3.7.4 Vectoriser un nouveau programme

Si le programme est nouveau, il devra être conçu pour permettre son optimisation (voir chapitre 2), sa vectorisation, et sa parallélisation (voir chapitre 4).

Sa conception devra prendre en compte les caractéristiques de performance du matériel moderne et faciliter l'optimisation des composants les plus critiques pour la performance du programme.

Il est aussi possible de choisir, un compilateur spécifique, une bibliothèque ou un générateur de code pour vectoriser (voir section 3.4). Cette tâche sera alors déléguée à une dépendance extérieure du programme qui devra être suivie avec le plus grand soin.

Le choix du compilateur est particulièrement important, puisque ceux-ci évoluent rapidement et ils optimisent et vectorisent de mieux en mieux. Ainsi, forcer l'utilisation d'un compilateur ancien pour des raisons de sûreté des résultats empêchera une utilisation efficace des architectures récentes sans pour autant être un gage réel de stabilité (voir section 9.5.1).

16. Les années 90 ont vu apparaître la programmation objet ainsi qu'une mode consistant à utiliser des objets le plus possible pour écrire du « beau code ». Malheureusement, les structures de données qui émergent de l'application intuitive de ce modèle ont de mauvaises performances d'accès mémoire et empêchent une vectorisation efficace.

L'essentiel du chapitre 3 –

Depuis une quinzaine d'années, les processeurs sont capables au sein d'un même cœur de calcul d'effectuer plusieurs calculs simultanément. Ne pas utiliser cette fonctionnalité revient à passer à côté de 75 à 93.75 % de la puissance de calcul brute de la machine utilisée. Il n'est pas envisageable de passé à côté d'une telle puissance de calcul.
Pour ce faire, les différentes techniques de vectorisation doivent être partagées. Si les langages C++ ou Rust sont trop bas niveau pour les non-experts, ils sont indispensables pour explorer de nouvelles possibilités d'optimisations ou de vectorisation. Les langages plus haut niveaux comme Python ou Julia permettent à des utilisateurs moins avancés une expressivité importante de leur calcul.
Là où Python offre un système de développement relativement simple, mais une exécution lente, Julia offre des temps d'exécution comparables au C mais avec des latences importantes durant le développement.

Chapitre 4

Parallélisation

Sujet du chapitre 4 –

Depuis quelques années, la fréquence des processeurs n'augmente plus et le temps d'attente de l'utilisateur ne peut être réduit que par parallélisation des calculs. De cet exercice émerge de nouvelles problématiques : gestion des communications entre les tâches de calcul, latence du réseau, mémoire distribuée bien plus complexe qu'une mémoire partagée, asynchronisme des tâches, tolérance aux pannes, et enfin des difficultés sans précédent pour déboguer les applications. Le développeur qui parallélise doit avoir une excellente connaissance des différents paradigmes de programmation parallèles, ainsi que des architectures toujours plus nombreuses et complexes que son application devra utiliser.

Sommaire

4.1 Introduction . . . 43
4.2 Considérations théoriques . . . 44
4.2.1 Granularité des tâches . . . 44
4.2.2 Passage à l'échelle . . . 44
4.2.3 Parallélisation sur la Grille . . . 47
4.3 Formes de parallélisation . . . 48
4.3.1 Introduction . . . 48
4.3.2 Parallélisation multi-cœur . . . 48
4.3.3 Parallélisme multi-nœud . . . 54
4.4 Les aléas du parallélisme . . . 59
4.4.1 Passage à l'échelle . . . 59
4.4.2 Équilibrage de charge . . . 60
4.4.3 Algorithmie . . . 60
4.4.4 Débogage . . . 61
4.4.5 Profilage . . . 62
4.5 Retours d'expériences . . . 63
4.5.1 Introduction . . . 63

4.5.2 *Multi-threading* en physique des particules 63
4.5.3 Calcul distribué dans LSST 65
4.6 Conclusion 66

4.1 Introduction

Dans un monde idéal, il serait possible d'accélérer indéfiniment les programmes sans les modifier, en augmentant simplement la fréquence d'horloge du processeur qui dicte le rythme auquel les instructions sont traitées. Malheureusement, les limites physiques de cette approche ont été atteintes autour de 2005, et l'on ne peut aujourd'hui augmenter cette fréquence qu'à un prix prohibitif en termes de consommation d'énergie et de dissipation de chaleur.

À la place, les processeurs modernes tentent donc d'effectuer plus de calculs par cycle d'horloge. La vectorisation, que nous avons traitée au chapitre 3, est une approche très économe en silicium pour parvenir à ce résultat. Mais tous les algorithmes ne s'y prêtent pas, et il est souvent préférable de pouvoir exécuter plusieurs sous-programmes composés d'instructions indépendantes simultanément. On parle alors de calcul parallèle.

Une première forme de parallélisation consiste à faire travailler ensemble plusieurs ordinateurs, appelés nœuds de calcul. Cette forme de parallélisation, appelée calcul distribué, est la seule qui soit capable de passer à l'échelle indéfiniment à technologie informatique constante. Mais l'écriture d'un programme de calcul distribué performant est très difficile, car les nœuds ne peuvent communiquer que par le biais d'un réseau dont la bande passante est bien plus basse que celle des échanges CPU-RAM, donc la latence est très élevée par rapport aux temps de calcul, et où l'ordre de distribution des messages ne peut être que partiellement garanti.

Pour cette raison parmi d'autres, un parallélisme à grain plus fin a aussi été introduit au sein des CPU, le multi-cœur. Un processeur multi-cœur est capable d'exécuter plusieurs sous-programmes (*threads*) simultanément, en permettant à ceux-ci d'échanger des données très rapidement via la RAM et la hiérarchie de caches. Convenablement utilisée, cette interconnexion rapide facilite grandement la parallélisation de calculs nécessitant des communications fréquentes entre sous-programmes. Mais même dans ce cadre favorable, la communication reste bien plus onéreuse que le calcul, et l'on ne peut donc pas en abuser.

Le calcul distribué et les processeurs multi-cœur sont omniprésents au sein des centres de calculs comme le CC-IN2P3 [46], car ce sont les seules infrastructures capables de traiter les volumes de données titanesques et toujours croissants de la Grille WLCG [25] sous des délais raisonnables. Un programme qui n'est pas capable d'exploiter ces ressources ne pourra donc pas contribuer aux analyses physiques à grande échelle couramment pratiquées à l'IN2P3.

Pendant de nombreuses années, les experts de la Grille ont su cacher l'existence du parallélisme multi-cœur et distribué aux physiciens et aux autres informaticiens, en tirant parti du fait que les analyses physiques manipulaient un grand nombre de données indépendantes. L'approche utilisée était de décomposer les données à traiter en blocs, et de lancer un exemplaire du programme d'analyse par cœur CPU du centre de calcul en ne lui confiant que ce sous-ensemble des données à traiter.

Mais malheureusement, cette approche atteint aujourd'hui ces limites. Comme le rapport capacité/prix et les performances de la RAM augmentent bien plus lentement que le nombre de cœurs des CPUs, il est de plus en plus souvent nécessaire de prendre explicitement en compte la nature multi-cœur des CPUs dans un programme pour faire bon usage des ressources mémoire. Et de nouvelles problématiques comme le traitement de données du télescope LSST ne se prêtent plus non plus à une distribution des calculs par découpage automatique des données d'entrée, car certaines étapes du calcul doivent prendre en compte une sous-partie des données trop grande pour un nœud de calcul individuel.

Par conséquent, les expériences de physique ont de plus en plus souvent besoin d'écrire des programmes qui prennent explicitement en compte le parallélisme du matériel de calcul. Dans ce chapitre, nous verrons donc quelles sont les difficultés techniques qui sont couramment rencontrées lors de l'écriture d'un programme parallèle, et quels outils peuvent être utilisés pour simplifier l'écriture de tels programmes.

4.2 Considérations théoriques

4.2.1 Granularité des tâches

Pour paralléliser un travail, il faut avant tout le décomposer en un certain nombre de tâches pouvant s'exécuter en parallèle.

Les centres de calculs actuels pouvant exécuter jusqu'à plusieurs millions de tâches en parallèle, il n'est pas viable pour un programmeur de définir celles-ci individuellement. À la place, on fait appel à des méthodes automatisables telles que la décomposition de domaine, qui consiste à découper le jeu de données d'entrée et confier à chaque tâche un sous-ensemble de données [1].

La granularité des tâches, c'est-à-dire la quantité de calcul qu'elles effectuent, est importante. Créer une tâche parallèle (*thread* ou processus) a toujours un certain coût, et il faut que le temps passé à exécuter la tâche amortisse largement ce coût pour que la parallélisation conduise à un gain net de performance. Par corollaire, l'utilisation d'une machine massivement parallèle pour effectuer un calcul n'a de sens que si le volume de calcul à effectuer est suffisant pour utiliser durablement les capacités de calcul de cette machine. Ceci pose une limite basse sur la taille des tâches parallèles.

Fixer une limite haute sur la taille des tâches peut aussi parfois avoir un sens. Évidemment, dans le cas extrême où les tâches sont si grosses qu'on a moins de tâches que de cœurs processeurs, on ne tire pas pleinement bénéfice du calcul parallèle. Mais par ailleurs, l'utilisation de tâches plus petites que nécessaire, de façon à avoir plusieurs tâches par cœur de calcul, facilite grandement l'équilibrage de charge entre les cœurs de calcul, notion que nous aborderons plus loin.

4.2.2 Passage à l'échelle

4.2.2.1 Introduction

Un programme parallèle ne s'exécutera qu'exceptionnellement n fois plus vite ou mieux [2] sur une machine à n processeurs. Ce passage à l'échelle parfait avec utilisation optimale des ressources de calcul n'est qu'un idéal vers lequel on doit tendre.

Une des raisons est que le programme parallèle n'utilise pas toutes les ressources de calcul disponibles pendant l'intégralité de son exécution. Par exemple, un schéma de fonctionnement classique est de commencer par une initialisation séquentielle sur un seul cœur CPU, puis de lancer le calcul sur les autres processeurs du système, poursuivre le calcul parallèle jusqu'à

1. Les méthodes de parallélisation historique de la Grille sont une forme de décomposition de domaine.
2. Il arrive rarement d'observer une accélération supérieure à n en parallèle quand l'algorithme parallèle utilise des ressources processeur partagées comme le cache L3 mieux que son homologue séquentiel.

ce qu'il soit terminé, et enfin terminer par une phase de finalisation séquentielle exécutée sur un seul cœur telle qu'un envoi des résultats à l'utilisateur. Durant les phases séquentielles, le programme sera limité par la performance d'un processeur unique.

D'autres phénomènes peuvent limiter le passage à l'échelle, par exemple le surcoût lié aux communications entre les tâches parallèles ou l'épuisement de ressources matérielles partagées entre les cœurs de calcul telles que la bande passante de la mémoire, du stockage, et de l'interconnexion réseau.

Une figure de mérite couramment utilisée est la loi de passage à l'échelle, qui relie l'accélération obtenue dans une exécution à n cœurs par rapport à la version séquentielle, au nombre de cœurs n alloués. Cette loi peut être prédite théoriquement dans certains cas idéalisés, que nous allons voir maintenant.

4.2.2.2 Loi d'Amdahl

La loi d'Amdahl donne l'accélération théorique α d'un programme dont l'exécution se compose de phases d'exécution séquentielles et de phases d'exécution parfaitement parallèles, en fonction de la fraction p de son code qui peut être parallélisée et du nombre de cœurs n sur lequel cette fraction est exécutée dans la version parallèle.

$$\alpha = \frac{1}{1 - p + \frac{p}{n}} \tag{4.1}$$

La figure 4.1 montre différents exemples d'accélérations que l'on peut attendre suivant le niveau de parallélisation du programme utilisé.

On voit que plus le nombre de cœurs n d'un système parallèle est important, plus il est difficile d'accélérer un traitement donné d'un facteur même approximativement égal à n en l'exécutant en parallèle sur ce système.

4.2.2.3 Loi de Gustafson

La loi d'Amdhal montre qu'on ne peut pas accélérer indéfiniment la résolution d'un problème donné en augmentant le niveau de parallélisme. Ce faisant, elle fixe des limites au degré auquel l'utilisation du parallélisme peut être utilisé pour optimiser un calcul existant.

Mais par construction, elle ne prend pas en compte le fait que l'augmentation des ressources des centres de calcul n'est généralement pas motivée par l'accélération des calculs existants, mais plutôt par la volonté de résoudre de nouveaux problèmes de plus grande taille sans que le temps de calcul ne devienne déraisonnable.

Dans ces circonstances, il peut être aussi pertinent d'étudier ce qui se passe pour un calcul qui n'est plus de taille fixe, mais dont la taille du cœur de calcul parallélisable augmente avec le nombre n de cœurs disponibles. On a alors une accélération par rapport au même calcul effectué sur un processeur séquentiel qui est donnée par la loi du Gustafson :

$$\alpha = n + (1 - n) \times s \tag{4.2}$$

Où s est la fraction du calcul qui est séquentielle à $n = 1$.

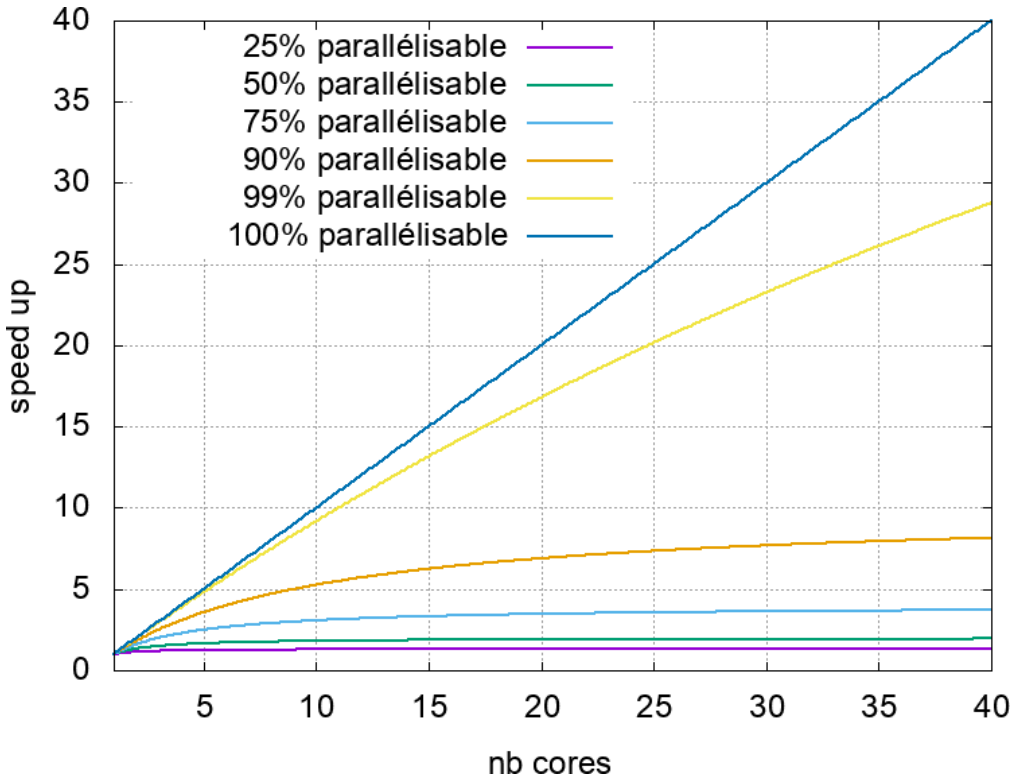

FIGURE 4.1 – Illustration de la loi d'Amdahl.

L'étude de cette loi (voir figure 4.2) montre que, sous certaines hypothèses, l'augmentation du parallélisme est toujours pertinente quand elle est motivée par la résolution de problèmes plus complexes. La réduction de la portion séquentielle du programme n'a alors pour objectif que d'utiliser plus efficacement les ressources de calcul ajoutées.

4.2.2.4 Applications réelles

Comme la loi d'Amdhal, la loi de Gustafson suppose une exécution parallèle parfaite de la section parallélisable du calcul, ce qui est une vision idéalisée.

Un calcul réel peut s'exécuter de moins en moins efficacement quand le parallélisme augmente, par exemple si le nombre de communications augmente quadratiquement avec n, ce qui est le cas lorsque chaque cœur communique ses résultats à tous les autres. Dans ce cas, l'interconnexion entre les cœurs sature inévitablement à grand n.

Il peut aussi arriver que contrairement à ce qui se passe dans le modèle de Gustafson, la taille de la portion séquentielle soit une fonction croissante de n. Cela se produit par exemple quand un seul cœur de calcul s'occupe du chargement des données d'entrées, du lancement des autres tâches parallèles, ou de l'accumulation des résultats. Dans ce cas, l'accélération de Gustafson se rapproche de celle d'Amdhal, avec les conséquences que l'on connaît.

Dans la littérature de programmation parallèle anglophone, on parle de *strong scaling* quand un calcul passe bien à l'échelle, c'est-à-dire fournit $\alpha \approx n$, selon le critère d'Amdhal, et de *weak scaling* quand un système passe bien à l'échelle selon le critère de Gustafson.

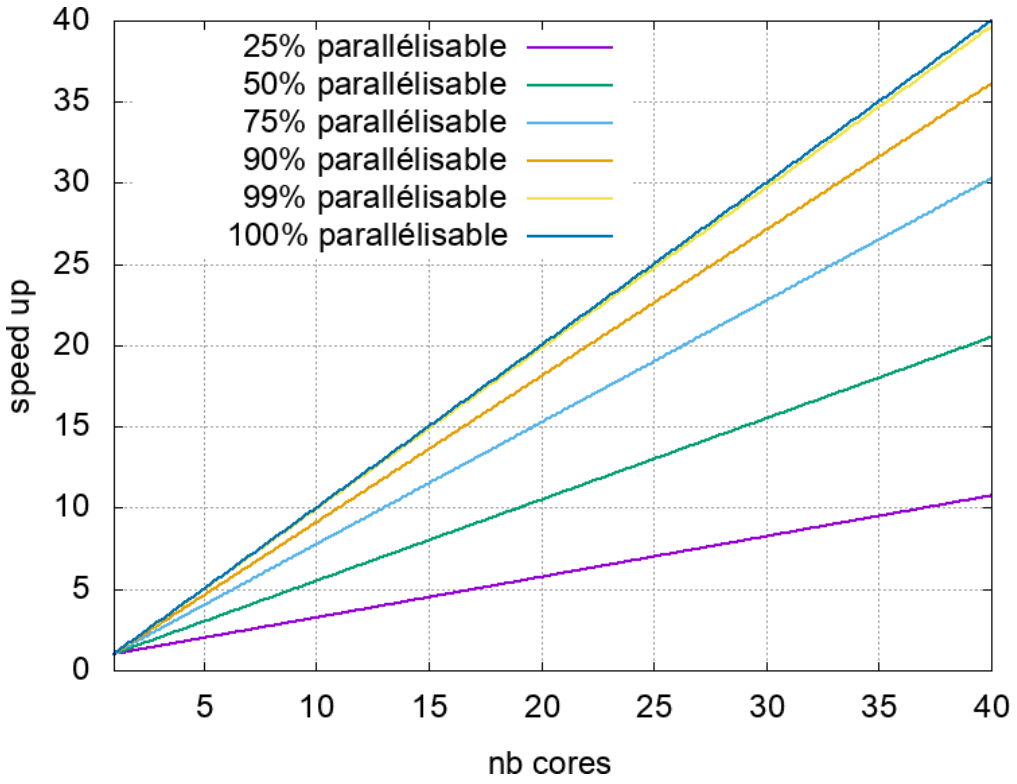

FIGURE 4.2 – Illustration de la loi de Gustafson.

4.2.3 Parallélisation sur la Grille

Il est important de souligner que l'utilisation de la programmation parallèle ne diminue qu'exceptionnellement le temps de calcul total du programme, accumulé sur tous les processeurs. Elle aura généralement plutôt tendance à l'augmenter. Ce n'est donc une optimisation du débit de données traité par unité de temps que lorsqu'on cherche à traiter un calcul unique le plus rapidement possible sur une machine qui n'a rien d'autre à faire. Ce qui est par exemple le cas en calcul intensif, ou lors d'une analyse interactive sur le poste de travail d'un physicien.

Si en revanche on souhaite traiter des jeux de données indépendants bien plus nombreux que le nombre de cœurs de calcul disponible, et dispose d'un système de parallélisation automatique entre jeux de données comme celui de la Grille, une parallélisation interne à chaque traitement n'améliorera pas le débit de données traité par seconde. Son seul intérêt sera de réduire la latence des calculs, c'est-à-dire le temps écoulé entre soumission d'un calcul et obtention du résultat. C'est la raison pour laquelle ce type de parallélisation n'a pas été beaucoup pratiqué dans la communauté IN2P3 historiquement.

La parallélisation des tâches individuelles sur la Grille ne doit donc pas être vue comme un substitut à l'optimisation du temps de calcul séquentiel, mais comme un complément de celle-ci qui vise à optimiser d'autres aspects du calcul que le débit de données traité par cycle processeur utilisé. Par exemple la latence, l'utilisation des ressources mémoire, la

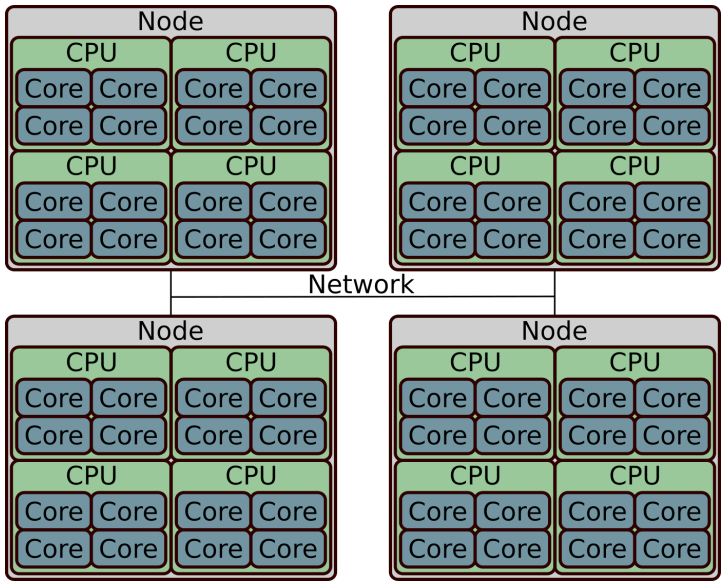

FIGURE 4.3 – Différents niveaux de parallélisme.

consommation d'énergie[3], ou la bande passante des entrées-sorties.

4.3 Formes de parallélisation

4.3.1 Introduction

La figure 4.3 illustre quelques formes de parallélisme couramment utilisées. Comme on peut le voir, il existe une hiérarchie du parallélisme, ce dernier étant présent à plusieurs échelles allant du cœur de calcul individuel au nœud complet.

Si certaines notions sont universelles, comme le souci de passer à l'échelle et de minimiser les communications par exemple, il existe aussi des aspects du matériel et du modèle de programmation qui changent selon l'échelle où on travaille. Nous allons maintenant étudier ces spécificités.

4.3.2 Parallélisation multi-cœur

4.3.2.1 Threads, cœurs et processeurs

Au sein d'un nœud de calcul, il existe plusieurs concepts matériels proches dont on pourra parfois ignorer les nuances, mais qu'il est important de pouvoir différencier :

— Un CPU *multi-thread* peut gérer plusieurs tâches concurrentes et basculer entre elles sans assistance du système d'exploitation.

— Un CPU multi-cœur possède la capacité d'exécuter plusieurs tâches simultanément.

3. Le parallélisme réduit la consommation énergétique des processeurs de plusieurs façons. Il permet d'obtenir une même puissance de calcul à une fréquence d'horloge plus faible, ainsi que de terminer plus rapidement un calcul pour ramener le système à un état de repos très économe en énergie.

— Un système multi-processeur possède plusieurs CPUs, généralement eux-mêmes multi-cœur et/ou *multi-thread*.

Ces trois concepts sont exposés par les systèmes d'exploitation via une abstraction unique, le *thread*, qui représente un sous-programme pouvant être associé à un *thread* du CPU ou s'exécuter sur un cœur CPU lorsque ces ressources matérielles seront disponibles.

Le *multi-threading* matériel n'est pas une forme de parallélisme, puisqu'il ne permet pas d'exécuter plus d'instructions en même temps. Mais il peut revêtir les apparences du parallélisme lorsqu'il est utilisé comme optimisation de performance, en permettant par exemple à un thread de prendre le relais d'un autre pendant que ce dernier attend un accès mémoire afin que les cycles CPU correspondants ne soient pas gaspillés.

Cette confusion est augmentée par le fait que les systèmes d'exploitation n'exposent la différence entre *thread* et cœur que dans leurs interfaces les plus bas niveau, et les fabricants de matériel abusent parfois de cette situation en entretenant la confusion pour pouvoir afficher des capacités multi-tâches supérieures dans leur communication.

Mais hélas, l'imposture est rapidement révélée quand on entreprend le processus d'optimisation. En effet, puisque le *multi-threading* matériel n'améliore les performances que quand un *thread* fait un usage inefficace du CPU (en attendant la mémoire par exemple), l'optimisation va avoir pour effet de diminuer l'impact de ce mécanisme sur les performances. Et sur un programme bien optimisé, le *multi-threading* matériel pourra même dégrader les performances, car il augmentera la pression sur les ressources CPU partagées entre les *threads*.

Face à cette situation, certains centres de calcul vont jusqu'à désactiver les fonctionnalités de *multi-threading* matériel au niveau OS. Sans cautionner cet extrémisme, qui force les utilisateurs à délaisser des ressources matérielles pouvant être utiles à certaines tâches comme la gestion des entrées-sorties, nous recommanderons de ne pas considérer le passage à l'échelle dans le régime de *multi-threading* matériel comme important, et de privilégier avant tout le parallélisme multi-cœur et multi-processeur.

La nuance entre multi-cœur et multi-processeur est plus subtile, et leur confusion est souvent plus pardonnable. Mais elle a une importance lors des accès mémoire et des communications :

— Les cœurs situés sur un processeur étant physiquement éloignés des cœurs situés sur un autre processeur, leurs communications ont plus de latence et doivent composer avec une bande passante plus faible.

— Chaque processeur possède généralement un accès privilégié à une partie de la RAM, les autres processeurs devant communiquer avec lui pour y accéder. Pour faire un usage optimal du bus mémoire, il faut alors privilégier l'accès aux données « locales ».

Alors que le nombre de cœur des processeurs augmente, ce type d'hétérogénéité des communications et des accès mémoires commence à apparaître même au sein des processeurs individuels. La frontière du processeur devient donc progressivement moins pertinente, et on tend à lui préférer aujourd'hui les concepts plus généraux de distance et de localité.

4.3.2.2 Mémoire partagée et cohérence de cache

Un principe de conception central des machines multi-cœur est que tous les cœurs de tous les processeurs ont accès à l'intégralité de la RAM, la manipulent via un système d'adressage universel, et peuvent utiliser cette mémoire partagée pour communiquer avec les autres cœurs

en écrivant et lisant dans des régions mémoires convenues à l'avance.

Nous avons déjà vu comment, même dans le cadre d'une mémoire en lecture seule, cette illusion d'une mémoire homogène n'est pas parfaite, car les coûts d'accès à la RAM peuvent varier significativement d'une région mémoire à une autre à cause des effets de localité.

Mais pour prendre en compte la possibilité d'écrire dans la mémoire partagée, la complexité matérielle doit encore augmenter. En effet, si l'on veut entretenir l'illusion d'une mémoire partagée, il faut garantir que tous les caches CPU s'accordent sur le contenu de cette mémoire. Par conséquent, la gestion des écritures nécessite pour le matériel l'utilisation d'un mécanisme de cohérence de cache qui garantit au moins deux propriétés :

— Toute écriture mémoire effectuée par un cœur finit tôt ou tard par être propagée dans les caches des autres cœurs.
— Il n'est pas possible pour un cœur d'écrire dans une région mémoire, ni de la lire en général, pendant qu'un autre cœur y effectue une écriture.

La cohérence de cache nécessite un grand nombre de communications entre tous les cœurs processeurs, et constitue donc un obstacle fondamental à l'augmentation du parallélisme intra-nœud. Pour cette raison, il est possible qu'à l'avenir, soit le parallélisme intra-nœud cessera d'augmenter, soit la notion de mémoire partagée cohérente sera partiellement ou totalement abandonnée au nom du passage à l'échelle, comme dans les GPUs.

Mais sur les processeurs actuels, la cohérence de cache a d'ors et déjà pour conséquence observable que si un *thread* accède à une région mémoire dans laquelle un autre *thread* est en train d'écrire ou vient d'écrire, cet accès mémoire sera beaucoup plus lent. La granularité à laquelle cet effet s'applique est la ligne de cache.

4.3.2.3 Abstractions d'exécution parallèle

L'abstraction principale fournie par le système d'exploitation pour permettre le calcul multi-cœur est le *thread*. Un processus en cours d'exécution peut créer autant de *threads* qu'il le souhaite, et le système d'exploitation se chargera d'organiser l'exécution de ceux-ci sur les différentes ressources de calcul disponible [4].

Sous les systèmes POSIX, comme Linux, l'API système principale pour créer et gérer des threads est PThread [47]. Comme beaucoup d'APIs POSIX, sa standardisation a malheureusement été un échec partiel, car son comportement varie tellement d'une implémentation à l'autre qu'il n'est pas vraiment possible de l'utiliser pour programmer plusieurs systèmes de façon portable. Par conséquent, lorsqu'une portabilité entre systèmes d'exploitation est désirée, on devra avoir recours à une couche d'abstraction.

Celle-ci est souvent fournie directement par la bibliothèque standard du langage de programmation utilisé, à condition toutefois que celui-ci supporte pleinement les *threads*. En effet, le support des *threads* compliquant l'implémentation, tous les langages de programmation ne supportent pas leur utilisation, et ceux qui le supportent ne permettent pas forcément leur exécution simultanée. Ainsi le langage Python par exemple permet de créer plusieurs tâches, mais pas de les exécuter simultanément.

4. Parfois, la logique d'ordonnancement par défaut du système d'exploitation n'est pas optimale. Dans ce cas, le programme peut le guider en lui indiquant la meilleure répartition des *threads* sur les ressources d'exécution disponibles.

Lorsque c'est le cas, une technique de contournement consiste à lancer n copies du programme, chacune tournant dans son propre processus système. Mais cette technique possède un grand nombre d'inconvénients :

— Le processus étant l'entité utilisée par le système d'exploitation pour cloisonner les ressources, il est très difficile de partager des ressources système (mémoire, fichiers, connexions réseau...) entre processus.
— Il est donc difficile de ne pas consommer n fois plus de RAM en dupliquant toutes les ressources partagées de l'application, problème qui n'existe pas en *multi-thread*.
— Les primitives fournies par le système d'exploitation pour la communication inter-processus sont souvent bien plus complexes et/ou bien moins performantes que celles utilisables pour la communication entre *threads*.
— Le processus étant l'entité atomique du crash, un programme multi-processus doit gérer à chaque instant la possibilité que l'un de ses sous-processus rencontre un problème, alors qu'en *multi-thread* le crash d'un thread emmène automatiquement le reste de l'application avec lui [5].

Quand on met tout cela bout à bout, la programmation d'un calcul multi-processus devient souvent aussi complexe que celle d'un calcul distribué, avec les mêmes difficultés de passage à l'échelle, et rend très difficile l'exploitation efficace de la mémoire partagée qui est l'élément central du parallélisme multi-cœur.

Par conséquent, les langages de programmation ne permettant pas l'utilisation de *threads* pour le parallélisme doivent être évités autant que faire se peut, et quand on n'a pas le choix ils doivent être interfacés à d'autres langages qui n'ont pas cette limitation pour tirer au mieux parti du parallélisme intra-nœud.

4.3.2.4 Synchronisation

Les garanties offertes par la cohérence de cache ne sont pas suffisantes pour la plupart des applications. Le fait qu'elles opèrent à une très petite granularité (accès mémoire de la taille d'un pointeur en général) signifie que :

— Si deux *threads* écrivent sur une même structure de données en même temps, ils peuvent laisser en mémoire un mélange binaire des deux versions de la structure, généralement invalide pour l'application.
— Si un *thread* lit une structure de données qu'un autre thread est en train d'écrire, il peut observer un mélange de l'ancien et du nouveau contenu de la structure, lui aussi généralement invalide.

De plus, pour ne rien arranger, les CPUs et les compilateurs s'autorisent à transformer les accès mémoire des applications de diverses manières, par exemple en ajoutant ou supprimant des lectures / écritures, ou en les réordonnant. Ces optimisations, initialement mises en place car elles étaient invisibles pour le code séquentiel, rendent très difficile tout raisonnement basé sur les accès mémoire dans un code parallèle.

Enfin, la mémoire partagée n'est pas un outil adapté pour un *thread* souhaitant attendre qu'un autre ait terminé son travail. On ne peut alors utiliser que l'attente active, qui est un

5. Cette propriété des conceptions multi-processus peut aussi être avantageusement utilisée pour augmenter la robustesse d'un programme face aux erreurs. Cependant, l'écriture de programmes robustes est très difficile, et les codes de calcul n'ont généralement pas besoin de cette garantie.

gaspillage de cycles CPU auquel aucun développeur soucieux de performances n'acceptera de se livrer pour de longues durées. Et les mécanismes matériels d'interruption inter-processeur permettant de faire mieux sont bien trop bas niveau et bien trop dangereux pour être exposés sans restriction à une application non privilégiée.

Les compilateurs de langages de programmation collaborent donc avec les systèmes d'exploitation pour offrir diverses réponses à ces problèmes. Par exemple, dans ***PThread***, le *mutex* permet à des *threads* de se mettre d'accord pour accéder à une donnée à tour de rôle, chaque thread attendant que le précédent ait fini avant d'accéder à la donnée. Et la *condition variable* permet à un *thread* de s'endormir en attendant qu'un autre *thread* ne le réveille.

Mais bien qu'ils simplifient l'écriture de code multi-cœur, ces mécanismes ne sont pas faciles à utiliser. Il n'est que trop facile d'oublier de les employer, de les utiliser de façon incorrecte, d'en faire un usage excessif nuisible à la performance, ou d'en avoir des attentes irréalistes comme d'espérer que synchroniser les accès à deux variables séparément conduise les autres *threads* à garder de ces deux variables une vision synchrone.

4.3.2.5 Abstractions d'ordre supérieur

L'écriture de code parallèle en manipulant directement des *threads* et des primitives de synchronisation bas niveau comme celles de ***PThread*** étant difficile, des modèles de programmation plus haut niveau simplifiant l'écriture de certaines catégories de programme ont été mis au point.

OpenMP [48] est une solution couramment utilisée en calcul intensif, qui se base sur des directives de compilation C/++ et Fortran standardisées. En sus d'offrir une abstraction des primitives du système d'exploitation, elle ajoute également une palette d'outils simplifiant grandement l'écriture et la validation de programmes parallèles composés de boucles **for** sur de grands tableaux d'éléments indépendants.

Toutefois, cette solution n'est pas proposée par tous les compilateurs, la qualité des implémentations varie grandement d'un compilateur à l'autre, et le vocabulaire d'abstraction ne permet pas facilement d'utiliser d'***OpenMP*** sur des programmes plus complexes, manipulant des structures de données contenant des pointeurs par exemple. De plus, comme OpenMP n'utilise pas des constructions standard du langage hôte, son utilisation rend certaines activités comme la méta-programmation relativement délicate.

Comme approche alternative, on peut également mentionner *Intel Threading Building Blocks* [**?**], dont le nom est souvent abrégé en « ***TBB*** », et qui est une bibliothèque C++ qui permet de rendre le parallélisme de *threads* plus composable.

En effet, dans la plupart des autres approches du parallélisme, une boucle parallèle dans une boucle parallèle est une mauvaise nouvelle, car elle signifie que n^2 *threads* seront créés au lieu de n, ce qui a un impact négatif sur les performances et l'utilisation des ressources. Ce problème complique souvent l'utilisation de bibliothèques de calculs écrites par des auteurs différents, qui ne peuvent se coordonner sur la question de la création des threads.

TBB règle ce problème en fournissant à l'ensemble de l'application un service commun d'exécution parallèle. Quand plusieurs bibliothèques utilisent ***TBB***, elles se coordonnent automatiquement pour exécuter leurs tâches sur les mêmes *threads* partagés, et l'on reste donc dans une configuration optimale à un *thread* par cœur CPU.

En dehors de ça, cette bibliothèque fournit une boucle **for** parallèle analogue à celle

d'***OpenMP***, diverses primitives de synchronisation, et d'autres mécanismes plus obscurs comme un moyen d'exécuter en parallèle des graphes de tâches simples.

Comme ***TBB*** est une bibliothèque, elle a l'avantage sur OpenMP de fonctionner de façon homogène sur tous les compilateurs C++ courants. En revanche, elle ne bénéficie pas d'une intégration au sein du compilateur, et ne peut donc pas offrir une assistance au programmeur qui cherche à contrôler les accès concurrents à ses variables comme OpenMP le fait.

C++17 [24] standardise une version parallèle des algorithmes de la STL. Dans le cas de *GCC* et *Clang*, celle-ci est basée sur ***TBB***. Ceci étend les capacités de ***TBB*** à toutes sortes de manipulation parallèle de conteneurs sur ces compilateurs, lorsque l'utilisation de C++17 et d'un compilateur récent est possible.

Avec cette évolution, C++ rejoindra la grande famille des langages de programmation disposant depuis quelque temps de fonctionnalités similaires (Rust, Java, Scala, C#...), soit en standard, soit via des bibliothèques matures et largement reconnues dans leurs communautés. En C++20, l'ergonomie de l'itération de collection en C++ devrait également enfin devenir comparable à celle des autres langages, grâce à l'introduction du concept de *range*.

4.3.2.6 Paradigmes pour le parallélisme

Si la disponibilité grandissante d'abstractions haut niveau pour le parallélisme multi-cœur est une excellente chose, car elle évite que certains types de codes parallèle soient réécrits sans cesse en faisant des erreurs, cela n'est pas une raison pour négliger les fondamentaux de la programmation parallèle.

En effet, même avec tous ces outils, il arrive encore trop souvent de se retrouver face à une situation où l'on doit gérer le parallélisme soi-même, et se retrouve alors face au chaos décrit dans la section 4.3.2.4. À tout instant, dans le parallélisme en mémoire partagée, on n'est qu'à une variable partagée de la catastrophe, et les erreurs peuvent ne pas être repérées pendant les tests si elles ne se manifestent que dans un ordre d'exécution bien précis.

Mais on peut faire mieux en restreignant volontairement le code que l'on écrit de façon à éviter l'existence d'état partagé et modifiable en mémoire, qui est la source de tous les maux, si possible avec l'assistance d'un langage de programmation conçu pour garantir le respect de cette restriction. Trois grandes stratégies peuvent ainsi être employées.

Dans une première stratégie, issue du monde de la programmation fonctionnelle, toutes les données qui sont partagées entre les threads sont en lecture seule. On distingue le style fonctionnel pur, où l'on s'interdit l'utilisation de données modifiables même lorsqu'elles ne sont pas partagées, du style impur où l'on s'autorise des modifications de données tant que celles-ci ne sont pas visibles du reste du programme.

L'utilisation d'un langage de programmation fonctionnel comme OCaml ou Haskell permet de faire vérifier le code par le compilateur pour s'assurer que ce style est bien suivi.

Dans une seconde stratégie, issue du monde des télécommunications, on décompose le programme en un grand nombre de sous-programmes (ne correspondant pas nécessairement à des *threads* du système d'exploitation) dont l'état interne est isolé et qui ne communiquent les uns les autres que par échange de messages. Selon que l'échange de message soit synchrone ou asynchrone, on parle de processus communiquants ou d'acteurs. Cette stratégie, qui se rapproche du calcul distribué, s'en distingue quand elle autorise l'existence d'état partagé en lecture seule commun, et la transmission de simples pointeurs vers un tel état partagé.

L'utilisation d'un langage de programmation conçu pour ce paradigme comme Go ou Erlang ne garantit malheureusement pas toujours une vérification parfaite du respect du paradigme (les garanties exactes dépendent du langage), mais augmente grandement son ergonomie et ses performances par l'introduction d'un mécanisme de processus légers moins coûteux que les *threads*.

Enfin, une troisième stratégie consiste à combiner ces deux approches. C'est ce que permet le langage Rust en vérifiant à la compilation qu'à chaque instant de l'exécution du programme une variable sera soit accessible en écriture soit partagée, tout en autorisant les variables à basculer d'un de ces états à l'autre au fil de l'exécution du programme.

4.3.3 Parallélisme multi-nœud

4.3.3.1 La fin des certitudes

À l'intérieur d'un nœud de calcul, il existe une vérité universelle accessible à tous les programmes en cours d'exécution : le contenu de la mémoire partagée.

Certes, cette vérité change tout le temps, et d'une manière qui peut être difficile à appréhender. Mais en maniant bien la synchronisation, il devient presque trivial de mettre deux threads d'accord sur quelque chose, et l'on peut entretenir le sentiment confortable d'avoir à chaque instant une vision d'ensemble du calcul, dont on peut facilement exposer chaque facette aux threads selon leurs besoins.

Malheureusement, l'accession au rang de calcul distribué, Graal du passage à l'échelle promettant à ses fidèles l'accès au nirvana du parallélisme inépuisable, implique le rite de passage du renoncement à ces conforts matériels.

Dans une infrastructure de calcul distribué, chaque nœud de calcul n'a accès qu'à une partie du jeu de données qui définit le problème. À chaque fois qu'il veut avoir accès à des données situées sur un autre nœud, il doit en discuter avec lui en lui envoyant un message par un réseau qui semble toujours trop lent, en latence comme en débit, et ayant la fâcheuse habitude de distribuer les communications dans un ordre différent que celui dans lequel elles ont été émises.

À la suite de quoi, avec un peu de chance, son interlocuteur répondra au bout d'un temps qui aura semblé durer une éternité. À moins bien sûr qu'il ne réponde jamais, bloqué dans l'attente d'un message d'un autre nœud, qui refuse de lui envoyer, car il attend d'abord un message de lui en raison d'un décalage mal géré entre la valeur de leurs horloges système respectives.

Vous l'aurez compris, il existe quelques différences conceptuelles entre le parallélisme en mémoire partagée (ou multi-cœur) et le parallélisme distribué (ou multi-nœud) qui doivent être prises en compte pour l'écriture de calculs distribués corrects et performants.

4.3.3.2 Réseau

Le réseau est l'élément le plus caractéristique d'un système distribué. Malheureusement, dans le domaine du calcul, c'est aussi celui qui est le plus variable.

D'un centre de calcul à l'autre, on peut en effet rencontrer toute une zoologie de couches physiques et de protocoles, allant de l'habituelle pile TCP/IP sur câble RJ45 à des intercon-

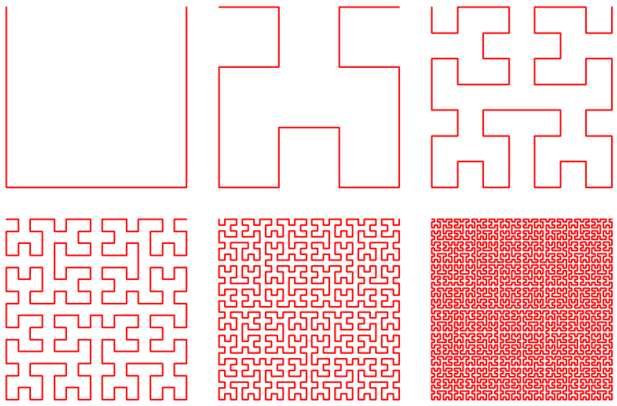

FIGURE 4.4 – Courbe de Hilbert.

nexions optiques dernier cri portant des protocoles réseaux exotiques dont la standardisation va de l'approximatif à l'inexistant.

Un calcul qui vise à être portable entre centres de calcul devra donc être capable de s'adapter à des interconnexions ayant une grande diversité de débits, de latences, et de protocoles natifs.

Le réseau relie les nœuds composant le système distribué selon des topologies parfois très éloignées des classiques hiérarchies de commutateurs câblés en étoile, tenant plutôt de la grille ou l'hypercube par exemple. Ces structures inhabituelles permettent de maximiser le nombre de premiers voisins de chaque machine, ce qui permet d'avoir des communications directes plus performantes.

Pour tirer parti de ces infrastructures, le code de calcul doit s'adapter en s'auto-organisant d'une façon qui maximise les communications entre premiers voisins, ce qui requiert d'avoir une connaissance minimale de la topologie du réseau. Mais comme la structure exacte est complexe et varie d'un centre à l'autre, des approximations sont souvent employés afin de simplifier l'implémentation du calcul.

Par exemple, une astuce parfois utilisée sur les réseaux en grille lorsque le calcul ne se prête pas naturellement à cette topologie est d'attribuer à chaque nœud une adresse logique unidimensionnelle attribuée aux machines en les numérotant selon une courbe de Hilbert (figure 4.4). Cela permet de ramener le problème de communication réseau à une seule dimension (communication avec le voisin d'avant et d'après dans le sens de la courbe) tout en conservant de bonnes propriétés de localité spatiale des communications.

4.3.3.3 Du paquet au message

La situation décrite précédemment donnera à tout programmeur le désir compréhensible de cacher la complexité et variabilité matérielle derrière une abstraction logicielle simplificatrice. Cependant, comme toujours, le premier niveau d'abstraction au-dessus du matériel doit être conçu avec prudence, et être avant tout une aide à la portabilité logicielle n'empêchant pas l'écriture d'un code performant.

Sur ce point, l'abstraction système classique du *socket* se révèle hélas peu adaptée au

calcul haute performance :

- Les modèles de communication proposés, comme le flux d'octets fiable continu, n'incluent pas des schémas de communication extrêmement répandus en calcul comme les messages fiables de taille variable. Ceux-ci doivent donc être réimplémentés par un protocole d'ordre supérieur, avec potentiellement un coût en bande passante réseau lié à une double encapsulation des données.
- Par construction, l'API *socket* nécessite un grand nombre de copies de données entre l'application et le stockage interne du système d'exploitation, qui ont un coût en capacité et bande passante RAM.
- Toutes les opérations nécessitent également l'utilisation d'appels systèmes et d'interruptions matérielles, dont le coût devient important lorsqu'ils se produisent un très grand nombre de fois par seconde.
- La communication asynchrone, essentielle à une bonne gestion de la latence réseau, est à la fois difficile d'utilisation et peu portable, non seulement entre systèmes d'exploitation mais aussi parfois entre versions d'un même système d'exploitation. En effet, les APIs associées évoluent encore vite par rapport aux versions du noyau Linux utilisées dans les centres de calcul.

Pour toutes ces raisons, il est souvent nécessaire de remplacer les abstractions système standard par d'autres plus adaptés aux besoins de la communication haute performance. Un exemple de ce type d'abstraction est LibFabric [50].

Par-dessus cette première couche d'abstraction, on peut ensuite fournir un deuxième niveau d'abstraction répondant de façon plus directe à des besoins applicatifs courants, au prix d'une généralité moindre. Dans le milieu des super-calculateurs, l'abstraction la plus utilisée est l'interface standardisée ***MPI*** [**?**], pour *Message Passing Interface*.

Comme son nom l'indique, cette interface se concentre principalement sur la transmission ordonnée et fiable de messages de taille arbitraire entre nœuds de calcul. Elle permet à l'application d'échanger des blocs de données typés de différentes manières : communication bloquante ou non-bloquante, point à point ou collective, avec l'ensemble du système ou à un sous-groupe... mais elle fournit aussi d'autres fonctionnalités importantes comme un outillage pour la création et l'adressage de processus sur les nœuds de calcul, une introspection de la topologie du réseau, un support des débogueurs et profileurs, et une gestion entrées/sorties de fichiers adaptée aux systèmes de fichiers distribués.

En dehors du monde du calcul intensif, d'autres outils pour l'échange de messages sont plus répandus, par exemple la bibliothèque multi-langages ZeroMQ [52]. Si ces outils sont un peu moins adaptés aux besoins du calcul que ***MPI***, ils ont néanmoins le grand avantage de s'adapter bien plus facilement aux infrastructures de calcul autres que les super-calculateurs. En effet, l'installation et la maintenance d'une infrastructure ***MPI*** nécessite un savoir-faire administratif très spécifique qui est peu rencontré en dehors de la communauté HPC.

4.3.3.4 Limite des messages

Les interfaces basées sur les messages comme ***MPI*** constituent le niveau minimum d'abstraction du réseau avec lequel il est possible d'être productif dans un contexte de calcul. Mais elles demeurent très difficile à utiliser directement.

Utilisées naïvement, elles conduisent très facilement à un calcul inefficace :

— L'envoi de messages de petite taille a un coût par octet transféré très important par rapport à ce qu'il est possible de faire avec de gros messages.
— Les communications collectives (de type diffusion, réduction, ou pire *all-to-all*), quand elles sont appliquées à l'ensemble du système distribué, ont des coûts en ressource réseau gigantesques. Elles doivent être employées avec une extrême parcimonie, et ne seront probablement plus utilisables du tout sur des super-calculateurs futurs.
— La plupart des bibliothèques fournissent par défaut des garanties comme la fiabilité de distribution ou le respect de l'ordre des messages transmis entre deux pairs, qui conduisent à une utilisation inefficace du réseau lorsqu'elles ne sont pas requises.
— La communication synchrone bloque toute activité CPU pendant la transmission réseau, qui est très longue au regard des temps de calcul, alors que les interfaces réseau modernes n'imposent aucunement cette contrainte.

À l'inverse, l'utilisation experte des bibliothèques de messagerie est très difficile et conduit fréquemment à des erreurs subtiles dans le code :

— Les communications asynchrones requièrent une gestion mémoire complexe pour éviter de libérer ou modifier les données en cours d'envoi, ou de lire trop tôt les données en cours de réception.
— Les communications sans copie mémoire requièrent souvent l'utilisation de méthodes d'allocation mémoire spécialisées, très difficile dans les langages de programmation généralistes où les bibliothèques n'exposent que rarement un contrôle sur la méthode d'allocation mémoire.
— Les communications désordonnées requièrent un grand effort de réflexion de la part du programmeur, puisque celui-ci doit prendre en compte toutes les manières dont les messages peuvent être intervertis.
— Les communications non fiables requièrent une gestion complexe des erreurs, surtout quand elles sont asynchrones.
— Les communications collectives non globales requièrent souvent une profonde restructuration du calcul pour l'adapter à la hiérarchie des communications locales.

Ces erreurs peuvent rester invisibles durant le développement, si elles ne se manifestent que dans des conditions d'exécution inaccessibles à ce stade [6].

Pour toutes ces raisons, de nombreuses recherches sont en cours pour mettre au point des abstractions de communication offrant un meilleur compromis ergonomie / efficacité que la communication par message.

4.3.3.5 Approches haut niveau

Une première approche consiste à fournir au développeur une abstraction du système distribué se rapprochant d'un système à mémoire partagée, conceptuellement plus simple à appréhender et sur lequel il existe une abondante littérature. C'est l'approche des bibliothèques à espace d'adressage partitionné, comme HPX [16], et des langages de programmation à tableaux distribués comme Fortran 2018 et Julia.

6. Le centre de calcul de production étant rarement disponible à la demande pendant le développement, celui-ci doit être largement effectué sur des machines ayant un nombre de nœuds bien plus faible que la machine finale.

La principale difficulté lorsqu'on conçoit une telle abstraction est d'exposer juste ce qu'il faut de la nature distribuée du système pour que le programmeur puisse correctement la prendre en compte et optimiser son programme en fonction, sans réintroduire toute la complexité d'une communication par message ce faisant.

Une approche très prometteuse est fournie dans ce domaine par l'utilisation de futures, abstraction monadique représentant une donnée en cours de transfert réseau ou de calcul sur un autre nœud. Les futures fournissent un moyen plus intuitif d'interagir avec des traitements asynchrones, surtout quand elles sont combinées à un système de coroutines intégré au langage de programmation permettant d'écrire du code asynchrone comme s'il s'agissait d'un code séquentiel bloquant.

À plus haut niveau, il est possible de bâtir sur une telle infrastructure des bibliothèques d'algorithmes parallèles standardisés analogues à celles que nous avons discuté précédemment.

Une autre approche consiste à écrire son calcul selon le paradigme de la programmation fonctionnelle, dont les propriétés de transparence référentielle se prêtent bien à toutes sortes d'optimisations automatiques telles qu'une mise en cache automatique des données sur les nœuds et une migration automatique des calculs vers les nœuds où leurs données d'entrée résident. C'est l'approche utilisée par les *frameworks* ***Hadoop*** [**?**] et ***Apache Spark*** [54], qui sont très utilisés dans la communauté *Big Data* et commencent à être aussi utilisés à l'IN2P3 dans le cadre des développements LSST.

4.3.3.6 Tolérance aux pannes

Traditionnellement, les codes de calcul ne sont pas conçus pour être robustes face aux pannes matérielles. Si une partie du centre de calcul tombe en panne, les calculs en cours sont brutalement arrêtés et toutes leurs données non sauvegardées sont perdues.

Si cette approche simplifie grandement l'écriture des programmes, elle est de moins en moins acceptable à mesure que le calcul distribué est pratiqué à une échelle toujours plus grande. En effet, le temps moyen entre deux pannes décroît en proportion du nombre de nœuds. S'il se compte en années sur des systèmes simples comme un nœud de calcul unique, il peut descendre en dessous du jour dans les grands centres de calcul actuel, et certaines prédictions estiment qu'il sera inférieur à une heure dans la prochaine génération *exascale* de super-calculateurs.

Dans ces conditions, des approches simples et relativement automatisables comme le *checkpointing* (mise en pause du calcul et sauvegarde complète de son état à intervalle régulier) qui ont bien servi les centres de calcul par le passé ne seront plus applicables. Il faudra à la place fournir des garanties de tolérance aux pannes à grain plus fin, ce qui est très difficile dans un système possédant un état interne qui varie rapidement dans le temps comme un programme écrit dans un style de programmation impératif.

Ici encore, l'application du paradigme de la programmation fonctionnelle peut améliorer la situation, puisque la transparence référentielle permet de considérer chaque nœud de calcul comme un outil aisément remplaçable au service du calcul d'une fonction mathématique sur un bloc de donnée. Les systèmes ***Hadoop*** et ***Apache Spark*** précités fournissent ainsi une tolérance aux fautes aux utilisateurs par réplication des données et redémarrage automatique des calculs présents sur un nœud de calcul qui est tombé en panne.

Hadoop et ***Apache Spark*** favorisent à la base fortement l'utilisation des langages Scala

et Java, mais il est possible de les interfacer avec d'autres langages en utilisant des outils comme MR4C [55].

4.4 Les aléas du parallélisme

4.4.1 Passage à l'échelle

Le problème de performance le plus fréquemment rencontré dans les programmes parallèles est une inaptitude à tirer parti des ressources d'exécution disponibles.

En particulier, tous les mécanismes de synchronisation qui sont basés sur l'attente (*mutexes*, communications bloquantes...) réduisent intrinsèquement le niveau de parallélisme puisqu'ils amènent une tâche à attendre qu'une autre progresse avant de pouvoir continuer son exécution. Si ces mécanismes de synchronisation causent beaucoup d'attente sur un chemin critique pour la performance du programme, le programme ne passera pas à l'échelle.

Dans ce cas, les approches suivantes peuvent être considérées, étant entendu qu'elles sont triées par ordre de priorité décroissante :

- **Élimination de la synchronisation.** Si une donnée ne varie pas au cours de l'exécution parallèle, sa synchronisation n'est pas nécessaire. Si elle varie peu et est de petite taille, il peut être acceptable d'en conserver une copie au sein de chaque tâche qui ne sera synchronisée qu'occasionnellement.
- **Synchronisation à grain plus fin.** Par exemple, au lieu de synchroniser l'accès à l'ensemble d'un tableau, on peut synchroniser l'accès à des éléments individuels. Cette approche réduit les conflits lorsque chaque tâche n'accède qu'à un sous-ensemble presque disjoint des données synchronisées, mais elle augmente souvent le coût de synchronisation et n'est donc pas applicable quand les accès sont fréquents.
- **Synchronisation non bloquante.** Il existe des techniques permettant de synchroniser deux tâches sans qu'elles ne s'attendent les une les autres à aucun moment (algorithmes *lock-free*, communications asynchrones...). En dernier recours, ces techniques peuvent réduire ou éliminer la perte de parallélisme liée à la synchronisation, mais cela se fait souvent au prix d'un surcoût sur une autre ressource (mémoire, CPU...) et d'une augmentation de la complexité du code.

En dehors de l'attente, une autre classe de problèmes à éviter est l'excès de communication. Les interconnexions qui portent ces communications (qu'il s'agisse du bus interne portant la cohérence de cache CPU ou du réseau explicitement commandé en calcul distribué) ont une certaine latence et une bande passante finie. Toute communication introduit donc des surcoûts qui écartent le programme parallèle du passage à l'échelle parfait. Et lorsque la bande passante d'une interconnexion est épuisée, on ne peut plus augmenter le niveau de parallélisme associé.

Cela implique, en particulier, qu'il faut se méfier de tous les mécanismes qui permettent à un *thread* ou plus de communiquer avec tous les autres *threads* du système. Quand une communication collective s'avère nécessaire, elle doit si possible se produire entre des groupes de *threads* de taille réduite, quitte à produire ensuite un résultat global par plusieurs passes de communication hiérarchique.

Les synchronisations de type « barrière », où l'on attend qu'un groupe de *threads* ait

terminé un travail avant de continuer, combinent à la fois une synchronisation basée sur l'attente et une communication entre de nombreux threads. Elles ne doivent donc être employées qu'avec la plus extrême parcimonie.

Enfin, le passage à l'échelle en parallèle passera aussi souvent par une optimisation individuelle de chaque tâche séquentielle, afin de réduire l'empreinte de chacune d'entre elle sur les ressources matérielles partagées dont l'épuisement empêche tout passage à l'échelle. La parallélisation se situera donc idéalement après l'optimisation séquentielle dans le processus de développement.

4.4.2 Équilibrage de charge

Dans les calculs où la parallélisation s'effectue par décomposition de domaine, un problème de passage à l'échelle apparaît lorsque le découpage automatique du calcul en blocs ne produit pas des tâches dont la charge de travail est équivalente. On parle de déséquilibre de charge.

Si ce défaut n'est pas corrigé, certains *threads* ou nœuds termineront leur travail avant les autres, et la performance globale du programme deviendra limitée par l'ensemble moins parallèle des *threads* qui continuent à travailler. Dans le cadre d'un calcul en plusieurs passes, ce phénomène se reproduira potentiellement à chaque passe, et son effet sera donc décuplé.

Plusieurs approches classiques existent pour régler ce problème :

— Lorsque la répartition de la charge de travail est prévisible à l'avance, un algorithme de décomposition de domaine plus astucieux pourra être utilisé pour produire des tâches de difficulté comparable.

— Dans des systèmes à faible niveau de parallélisme, le découpage du travail en petits blocs et leur insertion dans une file d'attente commune à tous les *threads* fournira une forme primitive de répartition dynamique du travail entre les *threads*.

— À un niveau de parallélisme plus élevé, il faudra faire appel à des algorithmes de répartition du travail plus complexes mais moins riches en synchronisation comme le vol de travail. L'utilisation d'implémentations standardisées comme celle de ***TBB*** est alors fortement recommandée.

On le devine, certaines de ces approches sont beaucoup plus faciles à appliquer en présence d'une mémoire partagée que dans un contexte distribué. Les systèmes distribués devront en effet rivaliser d'ingéniosité pour adapter ces algorithmes d'une façon qui ne nécessite pas des communications globales.

Deux approches classiques pour l'équilibrage de charge en calcul distribué sont la nomination d'un réseau hiérarchique de nœuds « maîtres », qui détiennent seuls une vision globale de la charge de travail des nœuds « esclaves » sous leur commandement, et l'échange de travail localisé entre nœuds voisins dans la topologie du réseau utilisé.

4.4.3 Algorithmie

Tous les algorithmes ne se prêtent pas à la parallélisation, et ceux qui s'y prêtent ne sont pas forcément parallélisables à toutes les échelles. De même que la parallélisation d'un calcul sur une matrice 5×5 n'a pas de sens, alors que sa vectorisation en a, il peut être pertinent pour un programme parallèle d'utiliser une stratégie de parallélisation à l'intérieur d'un nœud très différente de sa stratégie pour la distribution à grande échelle.

Lorsqu'on se lance dans un processus de parallélisation, il est important d'étudier l'état de l'art des structures de données et algorithmes utilisés pour résoudre en parallèle des problèmes similaires. On découvrira alors potentiellement qu'un algorithme séquentiel bien connu, comme *mergesort*, se décline très facilement en une version parallèle, tandis qu'un autre habituellement considéré comme supérieur en séquentiel, comme *quicksort*, ne se parallélise que beaucoup plus difficilement.

Cela évitera de partir sur une fausse piste en début de développement, et dans certains cas on y découvrira même une implémentation utilisable de l'algorithme étudié qui économisera de longues heures de développement.

Lorsque cette recherche n'est pas très fructueuse, l'expérimentation sur les infrastructures de calcul les plus fortement parallèles dont on dispose permettra de vérifier si une approche dont on a des raisons théoriques de penser qu'elle passe à l'échelle le fait réellement. En effet, il n'est que trop facile d'oublier dans son raisonnement des coûts matériels cachés, tels que la communication *broadcast* inhérente à la cohérence de cache en mémoire partagée, qui font qu'une l'implémentation d'un algorithme s'écarte du comportement théoriquement attendu.

4.4.4 Débogage

La validation d'un calcul parallèle est extrêmement ardue. En effet, il est très courant que le résultat d'un calcul parallèle dépende du détail de l'exécution de chaque *thread*, qui peut lui-même changer d'une exécution à l'autre au gré des fantaisies des mécanismes de synchronisation utilisé.

Très souvent, ces changements de résultat sont parfaitement anodins, et uniquement le fruit d'une absence d'associativité des calculs flottants ou d'une génération de nombres aléatoires différente. Voir à ce sujet le chapitre 8.

Mais malheureusement, il est difficile en pratique de discerner une fluctuation liée aux réordonnancements d'une erreur de calcul liée à un ordre d'exécution des *threads* imprévu. Une reproductibilité parfaite des résultats par rapport au calcul séquentiel est donc parfois exigée, mais cette contrainte de conception peut impliquer un temps de développement beaucoup plus important, une utilisation des ressources matérielles à l'exécution plus forte, et produire à terme une code qui passe moins bien à l'échelle. Tout cela étant dû au fait que le code parallèle doit potentiellement se synchroniser beaucoup plus pour reproduire exactement certains aspects du comportement séquentiel.

De plus, la reproduction exacte des résultats du calcul séquentiel n'est en aucun cas un gage de plus grande précision, car les réductions de données parallèles qui causent ces fluctuations des résultats produisent souvent en réalité des résultats plus précis qu'une simple accumulation séquentielle[7]. Pour un autre exemple d'augmentation de précision liée à l'optimisation, mais cette fois dans un contexte de vectorisation, consulter le chapitre 3.

Quand un problème est trouvé, la reproduction et l'analyse de celui-ci peut être difficile :

7. Cet effet est dû à la nature relative de la précision en virgule flottante. Lorsqu'on effectue une accumulation séquentielle naïve d'un ensemble de nombres en virgule flottante, on finit souvent à terme par ajouter des petites quantités à un accumulateur dont la valeur est grande, ce qui fait que certaines décimales des données d'entrées sont ignorées. La réduction parallèle des données, qui utilise un plus grand nombre d'accumulateurs, est moins sujette à ce problème.

— Le problème peut se présenter sur une machine mais pas sur une autre, du fait de différences de niveau de parallélisme ou même de micro-architecture CPU.
— Il peut ne se produire que très rarement, et nécessiter de passer par des centaines d'exécutions justes avant d'observer un résultat faux. Quand l'application est interactive (ex : visualisation), on peut alors devoir ajouter un mode d'exécution non interactif.
— L'instrumentation manuelle du programme peut faire disparaître le problème en changeant des décisions du compilateur ou en ajoutant des petits délais d'exécution qui rendent la situation problématique beaucoup moins probable.
— Les outils d'investigation automatisés, comme les débogueurs, rencontrent une partie de ces problèmes et ont aussi une ergonomie bien moins bonne quand ils sont appliqués à des programmes parallélisés (lorsque la possibilité existe).

Des outils de débogage spécialement adaptés aux programmes distribués sont parfois développés par les équipes des super-calculateurs. Si l'intention est louable, ils sont souvent bien moins flexibles que leurs homologues pour codes séquentiels, difficiles à installer sur d'autres machines que le super calculateur pour lequel ils ont été conçus, et ils ne supportent généralement que la distribution de calculs via ***MPI***.

4.4.5 Profilage

La situation de l'analyse de performances des applications parallèles n'est guère plus reluisante que celle du débogage.

L'écriture et l'analyse de *benchmarks* parallèles est d'abord compliquée par une dépendance beaucoup plus grande vis-à-vis de l'activité de fond de la machine hôte qu'avec du code séquentiel. Même en dépensant une grande énergie à réduire cette activité, les temps d'exécution fluctuent beaucoup plus en parallèle, et il est donc plus difficile de juger de la pertinence d'une optimisation.

L'analyse des points chauds par un profileur est aussi plus complexe. Certains profileurs anciens, tels que *GProf*, sont incapables de mesurer un profil de l'exécution d'un code à plusieurs *threads*. D'autres, comme *Callgrind*, simulent l'exécution parallèle de façon séquentielle selon des hypothèses très discutables, et ne sont donc pas non plus à même de fournir une description réaliste de l'exécution parallèle d'un code sur un jeu de données proche des conditions de production. Seuls les profileurs basés sur une instrumentation matérielle assistée par le système d'exploitation, comme *Perf*, parviennent à fournir une mesure raisonnablement fiable du profil d'exécution d'un programme parallèle.

En calcul distribué, la situation est encore plus désespérée. Les profileurs capables d'analyser précisément l'exécution d'un calcul distribué, qui sont conçus par les mêmes équipes auxquelles ont doit des débogueurs distribués, sont globalement sujets aux mêmes contraintes de faible portabilité et forte dépendance envers ***MPI***. De plus, les volumes de données générés par ces mesures sont aussi gigantesques, ce qui complique leur analyse.

Parfois, on doit donc s'en remettre à la solution de contournement d'utiliser comme outil de profilage primitif le système de *monitoring* système du centre de calcul, qui décrit l'évolution temporelle de l'utilisation des différentes ressources matérielles et peut aider à dégrossir l'analyse des goulots d'étranglement. Mais ces outils restent à gros grain, tant spatialement (ils n'indiquent pas précisément la source du problème dans le code) que temporellement (les mesures ne sont pas fréquentes), et leur sortie n'est donc exploitable qu'en la complétant par

un processus d'instrumentation manuelle du code particulièrement laborieux.

4.5 Retours d'expériences

4.5.1 Introduction

On ne saurait faire un retour d'expérience du parallélisme dans un contexte IN2P3 sans revenir un instant sur l'une des plus grandes expériences de parallélisme à grande échelle jamais tentée – et réussie : la grille WLCG.

Celle-ci a prouvé par l'expérience qu'une très grande variété de calculs, opérant sur des volumes de données colossaux, peut être parallélisée par décomposition de domaine sans que les développeurs des codes de calcul associés n'aient à lever le petit doigt ou même effleurer de leur pensée le concept de *thread*. C'est là une belle réussite technique et ergonomique du parallélisme à grande échelle dont nos communautés ne peuvent que se féliciter.

Malheureusement, plusieurs de nos expériences de physiques atteignent aujourd'hui les limites du mécanisme de parallélisation automatique de la Grille et doivent s'y substituer, soit en partie en remplaçant simplement sa composante de parallélisme intra-nœud, soit en totalité en fournissant elles-mêmes leurs propres moteurs de calcul distribué.

Le LAL est idéalement placé pour témoigner de ces deux efforts, puisque son service informatique contribue à la fois à la parallélisation *multi-thread* de plusieurs *frameworks* de traitement d'événements de physique des hautes énergies et à la mise au point d'un nouveau système de calcul distribué pour le télescope LSST.

4.5.2 *Multi-threading* en physique des particules

L'évolution des ressources matérielles de calcul va dans le sens d'une réduction de la quantité de RAM disponible par cœur CPU [8], alors que l'évolution des expériences de physique des particules au LHC va dans le sens d'une augmentation de la quantité de données et de la complexité des traitements par évènements. Les expériences LHC travaillant déjà à la limite du budget RAM qui leur est alloué par les centres de calcul, il est rapidement devenu clair que cette évolution conjointe allait conduire à une impasse si rien n'était fait pour l'empêcher.

Après avoir tenté pendant plusieurs années de retarder l'échéance par des solutions de contournement exploitant de manière hasardeuse le mécanisme de *copy-on-write* du système d'exploitation Linux [9], les expériences LHC ont fini par accepter l'inévitable et engager un processus de parallélisation *multi-thread* de leur code afin de pouvoir plus facilement partager des ressources mémoires communes entre cœurs CPU et ainsi diminuer leur empreinte RAM.

Comme tout effort visant à paralléliser un code séquentiel de grande taille, cet exercice s'est révélé périlleux. En effet, un code séquentiel, qui n'a jamais été conçu pour une exécution

8. Sans parler des GPUs, dont la capacité mémoire par cœur est très inférieure.

9. Sous Linux, juste après un appel système *fork*, les deux processus résultants partagent l'intégralité de leurs ressources mémoire. Celles-ci ne seront dissociées qu'au fur et à mesure que des écritures RAM surviendront. Il est possible d'abuser de ce mécanisme pour obtenir une forme simple de partage mémoire entre processus, mais cela n'est efficace que si l'appel à *fork* survient très tard dans l'exécution du programme et si celui-ci minimise ses écritures mémoires par la suite. Cette contrainte invisible complique l'écriture et la maintenance du code.

parallèle va intégrer la supposition qu'il s'exécute de façon sérielle en profondeur, et peut avoir recours à un grand nombre de pratiques incompatibles avec le parallélisme.

Par exemple, un obstacle significatif a été l'utilisation intensive de variables globales ou de constructions conceptuellement équivalentes (pointeurs vers une donnée présents dans de nombreux endroits du code, variables statiques...), ainsi que le recours fréquent à l'initialisation paresseuse [10]. L'existence de mécanismes déclenchant silencieusement une entrée-sortie lorsqu'on se livre à ce qui semble être la lecture d'une donnée en RAM s'est aussi révélée problématique, puisque les entrées-sorties doivent être soigneusement encadrées dans un environnement parallèle.

Très rapidement, il est devenu apparent que l'exercice de parallélisation aurait pu être grandement simplifié par un meilleur respect des règles de l'art de l'ingénierie logicielle, qui tendent à décourager ce type de pratiques afin de garantir une plus grande intelligibilité du code. Hélas, la très grande majorité du code ayant été développé par des personnels contractuels (doctorants, post-docs...) aujourd'hui disparus, et la main d'œuvre disponible pour la maintenance et l'évolution du code est aujourd'hui très faible. Il a été initialement jugé nécessaire de s'en tenir à des modifications aussi minimales que possible, et de ne pas procéder à une consolidation des fondations du *framework* avant d'effectuer la parallélisation.

Par la suite, cependant, la stratégie de parallélisation de différentes expériences a divergé. Certaines expériences, comme LHCb, ont fait le pari d'entreprendre un changement majeur de leur modèle de traitement d'évènement pour le rapprocher du paradigme de la programmation fonctionnelle, en gageant que l'important coût de développement lié à cette migration serait amorti par une bien plus grande facilité d'écriture du code par les physiciens. D'autres expériences, comme ATLAS, ont choisi à la place d'effectuer la migration vers une exécution *multi-thread* de façon aussi itérative que possible, en se concentrant sur la parallélisation des points chauds et en sérialisant tout le reste du calcul à grand renfort de *mutexes*.

Le pari de l'expérience LHCb s'est révélé payant, puisqu'ils disposent aujourd'hui d'un traitement d'évènements parallèle passant suffisamment bien à l'échelle pour leurs besoins actuels, là où les traitements de données de l'expérience ATLAS souffre toujours de très graves problèmes de passage à l'échelle. Toutefois, ce pari, qui était indubitablement risqué, doit être replacé dans son contexte : l'échéance de LHCb pour l'obtention d'une bonne parallélisation était le *Run 3* du LHC, bien plus proche que le *Run 4* visé par ATLAS. Et il est à ce stade possible qu'ATLAS parvienne à atteindre le niveau de parallélisme souhaité via leur approche plus prudente d'ici le *Run 4*.

Quoi qu'il en soit, le processus de parallélisation a été pour ces expériences une occasion de faire le point sur les outils de débogage et de profilage parallèle, leurs conclusions étant malheureusement en accord avec celles de ce document.

Et pour les deux expériences, l'objectif initial de la parallélisation, c'est-à-dire la réduction de l'empreinte mémoire, est aujourd'hui d'ors et déjà atteint puisque celle-ci se situe désormais autour de 150 Mo par *thread* ajouté en complément du *thread* principal, au lieu de plus de 2 Go par processus précédemment.

En conclusion, nous pouvons tirer de cette expérience les conclusions suivantes :

10. Procédé consistant à n'initialiser une donnée que la première fois qu'un programme y accède. Dans un environnement parallèle, cette optimisation nécessite une synchronisation coûteuse en performance et n'est donc généralement pas souhaitable.

— À chaque fois que c'est possible, la parallélisation du code doit être envisagée dès sa conception, car elle pourra alors guider de nombreux autres aspects de son fonctionnement dans la bonne direction.
— La parallélisation du code peut être facilitée par l'adoption de paradigmes appropriés, comme elle peut être compliquée par un faible niveau de qualité initiale du code. Lorsque c'est possible, commencer par une refonte du code visant à le simplifier et le clarifier simplifiera grandement l'exercice ultérieur de parallélisation.
— S'il est facile de trouver de la main d'œuvre pour l'écriture d'un nouveau programme, le faible niveau de main d'œuvre stable alloué à la maintenance des infrastructures logicielles a des conséquences préoccupantes, au LHC comme ailleurs.
— Si le développement itératif a souvent meilleure presse dans nos communautés, il ne faut pas négliger les bénéfices d'une approche plus disruptive lorsqu'un grand changement doit de toutes façons être effectué dans un logiciel.

4.5.3 Calcul distribué dans LSST

Le télescope LSST manipulera des volumes de données sans précédent dans l'histoire de l'astrophysique, et les infrastructures logicielles de cette communauté ne sont pas prêtes à faire face à ce déluge. C'est la raison pour laquelle le LAL pilote une action de R&D autour du calcul distribué dans cette expérience, proposant une approche de calcul distribué basée sur le *framework* ***Apache Spark***.

Le choix d'utiliser ***Spark*** plutôt que les mécanismes de distribution de calcul standard de la Grille a été motivé par deux facteurs :

— D'une part, les volumes de données que l'on peut extraire par une décomposition de domaine automatique sont trop imposants pour être raisonnablement traités par un seul nœud, et a fortiori par un seul cœur. Alors que ***Spark*** a été spécifiquement conçu pour la manipulation de grands volumes de données ne tenant pas en RAM.
— D'autre part, le processus de calcul de LSST peut être aisément décrit par un graphe acyclique de tâches, une abstraction de calcul nativement fournie par ***Spark***.

Cependant, le choix de ***Spark*** a aussi impliqué l'interfaçage voire la réécriture d'un certain nombre de bibliothèques d'astrophysique, ***Spark*** étant basé sur le langage de programmation Scala là où la communauté d'astrophysique utilise traditionnellement plutôt les langages Fortran, C++, et Python. Si ***Spark*** fournit en théorie une interface vers d'autres langages, il s'est rapidement avéré que l'écriture directe de code Scala était nécessaire à l'obtention d'une ergonomie et de performances maximales.

Un aspect positif de cette réécriture est que la tentation d'une migration itérative du code ne s'est jamais présentée, puisque la réécriture était la manière la plus simple d'interfacer les traitements existants à ***Spark***. Cette réécriture est l'occasion de procéder à une revue des codes d'analyse, et de les migrer vers le paradigme fonctionnel de ***Spark*** qui se prête mieux à la parallélisation distribuée.

Si ce système est encore à un stade de développement jeune, les premiers tests de passage à l'échelle sont extrêmement encourageants, et l'équipe chargée de ce projet au LAL est assez confiante pour le proposer comme composant d'un *broker* de LSST, infrastructure haute performance chargée de traiter l'énorme flux de données sortant du télescope en direct pour alerter les groupes de physique étudiants les images du télescope à travers le monde des

événements astronomiques de leur choix.

Dans l'ensemble, cette R&D amont est donc un succès, et montre encore une fois que les innovations technologiques de rupture ne doivent pas être un sujet tabou lorsqu'on fait face à un problème technique difficile comme le traitement de péta-octets de données astrophysiques.

4.6 Conclusion

Pour répondre à des besoins de calcul en croissante exponentielle alors que les gains de performance séquentielle d'une génération de CPU à l'autre étaient en décroissance exponentielle, les infrastructures de calcul ont dû évoluer en devenant massivement parallèles.

Si l'infrastructure de la Grille a longtemps su assurer automatiquement l'intégralité du travail de parallélisation des calculs de physique, cette approche atteint aujourd'hui ses limites, et la parallélisation manuelle de certains calculs devient nécessaire.

Malheureusement, le calcul massivement parallèle repose sur une infrastructure matérielle hiérarchique extrêmement complexe, dont la programmation regorge de périls. De nombreux obstacles doivent être surmontés pour bien en tirer parti et assurer un bon passage à l'échelle des calculs :

— La latence élevée, la bande passante réduite, et les topologies complexes des interconnexions.
— Le réordonnancement des accès mémoires et des messages réseau.
— Les abstractions inadaptées qui facilitent l'écriture de code incorrect, non portable, ou peu performant.
— La nécessité d'adapter les algorithmes à l'échelle de parallélisme considérée.
— Le caractère stochastique de l'exécution, source d'erreurs intermittentes aussi difficiles à détecter qu'à analyser.
— L'existence de pannes matérielles qui doivent être prises en compte dès qu'on veut prendre la main sur la distribution du calcul.
— L'immaturité des outils de débogage et de profilage.

Cependant, l'existence de nombreuses applications parallèles montre que toutes ces difficultés peuvent être surmontées, et des nouveaux paradigmes, bibliothèques et outils apparaissent chaque année pour simplifier toujours plus l'écriture et le passage à l'échelle de programmes parallèles.

En 2019, le calcul parallèle et distribué reste donc un domaine en évolution rapide, que ce soit au niveau matériel ou au niveau logicielle, et une veille technologique extrêmement attentive est requise pour permettre l'intégration de ces nouvelles approches du parallélisme au modèle de calcul de l'IN2P3, ainsi que la formation des développeurs à ces techniques plus productives.

L'essentiel du chapitre 4 –

Le matériel de calcul parallèle n'a jamais été aussi complexe, et la prise en main de cette complexité passe souvent par un recours à des approches de développement toujours plus sophistiquées. Le renouveau de paradigmes délaissés comme la programmation fonctionnelle ou les objets distribués pourrait répondre à cette crise de complexité logicielle en rétablissant un peu d'ordre bienvenu dans le chaos de l'écriture d'applications parallèles.

Chapitre 5

Quels langages de programmation ?

Sujet du chapitre 5 –

Il existe de nombreux langages de programmation. Généralement, les langages bas-niveau, proches des instructions machines, sont réservés aux experts, tandis que les langages haut-niveau, manipulant davantage de concepts abstraits, sont plutôt utilisés par les non-experts.
Ce chapitre décrit le fonctionnement ainsi que les avantages et les inconvénients de quelques langages utilisés en calcul intensif ou en simulation au sein de l'IN2P3.

Sommaire

5.1 Introduction **70**
5.2 C **70**
5.3 C++ **71**
5.4 Fortran **72**
5.5 Ada **73**
5.6 Rust **74**
5.7 Go **75**
5.8 Python **76**
5.9 Julia **77**
5.10 Conclusions **78**

5.1 Introduction

Les langages de programmation ont une influence significative sur la performance des programmes. Ils déterminent, entre autres choses, le degré d'optimisation automatique possible, le contrôle fin qu'un expert peut exercer sur le comportement d'un programme à l'exécution, et le degré auquel un programmeur qui les utilise est encouragé à écrire du code efficace.

Nous commencerons par faire un état de l'art de langages établis et émergents en calcul scientifique, analysés sous l'angle de la performance logicielle mais aussi d'autres critères comme l'ergonomie et la maturité, ce qui nous permettra de produire des recommandations quant au choix d'un langage pour le calcul à l'IN2P3.

5.2 C

Le langage C a été créé dans le cadre du développement du système d'exploitation Unix, puis a graduellement étendu son influence jusqu'à devenir aujourd'hui le standard de fait pour le développement de systèmes d'exploitation et l'interaction directe avec ceux-ci. Pour cette raison, c'est aussi le langage qui a été porté sur le plus grand nombre d'architectures matérielles, au point qu'il est quasiment impossible de trouver un CPU pour lequel il n'existe pas de compilateur C. C'est un standard de fait pour la programmation bas niveau, très largement enseigné dans ce cadre.

Du fait de ses objectifs de conception, c'est un des langages actuels qui offre le contrôle le plus fin sur la machine, avec par exemple la possibilité d'accéder à des adresses mémoires arbitraires, de contrôler finement l'organisation de ses données en RAM et l'ordonnancement des accès mémoires au niveau CPU, ou de communiquer directement avec le système d'exploitation. Malheureusement, ce grand pouvoir implique aussi une grande responsabilité, car le C simplifie aussi terriblement l'écriture de code incorrect.

Par exemple, le type de données central du langage est le pointeur, abstraction minimale d'une adresse mémoire qui joue de très nombreux rôles allant de l'émulation de tableaux (ces derniers étant très mal supportés par le langage) à la gestion d'allocation mémoire en passant par la construction de structures de données de types graphe. Comme un pointeur est un objet très flexible, un programmeur faisant face à un programme utilisant intensivement ceux-ci aura souvent le plus grand mal à déterminer leur rôle exact, et ses erreurs d'interprétation seront une source fréquente de bogues. Cette flexibilité signifie aussi qu'un compilateur aura une grande difficulté à déterminer ce qui constitue une utilisation incorrecte d'un pointeur, et ne pourra donc pas aider les programmeurs en les avertissant lorsqu'ils se livrent à des opérations douteuses.

Le C offre au programmeur des moyens d'abstraction très limités, qui ne permettent notamment pas de construire des objets au rôle mieux défini que les pointeurs, donc plus facile à utiliser. Les limitations du C, comme l'absence de support des tableaux multidimensionnels, ne peuvent donc pas être facilement contournées par une bibliothèque. Plus généralement, ce vocabulaire d'abstraction limité rend très difficile d'exprimer des concepts complexes dans une interface C ou d'écrire de grosses applications en C.

En revanche, ce minimalisme a le mérite d'avoir permis au C d'acquérir une interface binaire stable et standardisée, ce qu'il est l'un des seuls langages à fournir. Pour cette raison,

et du fait de son rôle dans la programmation système, il est courant de voir des langages de programmation fournir une interface avec le C, et de voir des bibliothèques exposer une interface C même quand elles ne sont pas écrites en C.

En résumé, l'utilisation du C n'est intéressante que lorsqu'on souhaite interagir directement avec le matériel ou le système d'exploitation. Ne possédant même pas un support complet des tableaux, le langage n'a aucun intérêt particulier pour le calcul, et pour tout usage applicatif généraliste il est généralement avantageusement remplacé par des langages comme C++ ou Rust qui offrent un sur-ensemble plus expressif de ses fonctionnalités.

Le C est généralement compilé statiquement vers des binaires natifs.

5.3 C++

Le langage C++ se voulait initialement être une surcouche du langage C, apportant des capacités d'abstraction supplémentaires comme le support de la programmation orientée objet. Si la base commune des deux langages a légèrement divergé depuis, ils restent remarquablement compatibles, pour le meilleur (interfaçage extrêmement aisé des bibliothèques C) comme pour le pire (héritage de tous les défauts de conception du C).

La longue évolution du C++, à travers plusieurs décennies d'ajouts de fonctionnalités tout en préservant une grande compatibilité descendante, a conduit le langage à devenir très complexe. Le C++ moderne fournit ainsi, entre autres, la surcharge de fonction et d'opérateurs, un modèle de programmation objet basé sur l'héritage, des types génériques basés sur la génération de code spécialisé (les *templates*), un environnement d'exécution de code à la compilation, un support natif du parallélisme multi-thread, et un ensemble très complet quoique malcommode d'outils pour manipuler des types à la compilation.

Cette pléthore de fonctionnalités rend le langage extrêmement difficile à apprendre et maîtriser, d'autant plus que les fonctionnalités du langage ne sont pas conçues de façon irréprochable et interagissent de manière très complexe entre elles. En contrepartie, les experts C++ tirent parti de sa richesse pour produire des bibliothèques extrêmement expressives, offrant des fonctionnalités telles que l'optimisation automatique des calculs d'algèbre linéaire ou la vectorisation par intrinsèque indépendante de l'architecture matérielle cible.

Comme il est difficile de maîtriser la totalité du langage, les programmeurs C++ tendent à se spécialiser dans un sous-ensemble avec lequel ils sont à l'aise, ce qui complique l'interaction entre équipes indépendantes. Ainsi, il existe des équipes qui utilisent le C++ comme un C avec quelques fioritures, d'autres qui l'utilisent principalement pour ses fonctionnalités objet, d'autres encore qui l'utilisent principalement pour ses capacités de génération de code et d'exécution de travaux à la compilation et ces sous-communautés communiquent difficilement.

Le C++ permettant un contrôle matériel fin, il est possible, moyennant un certain effort, d'en tirer des bonnes performances. Cette combinaison de bonnes performances à de bonnes capacités d'abstraction en a fait un choix standard pour les grandes applications à fortes contraintes de performance. Mais sa complexité le réserve aux spécialistes qui sont prêts à lui dédier leur carrière, ce qui l'a aussi souvent conduit à être couplé avec des langages plus haut niveau comme Python ou Lua pour ouvrir certaines utilisations de bibliothèques C++ à des publics moins experts.

C'est une telle combinaison de « cœur de calcul » C++ avec une interface haut niveau

en Python qui est couramment utilisée à l'IN2P3 aujourd'hui.

Le C++ est généralement compilé statiquement vers des binaires natifs, même si ROOT permet aussi de le compiler à la volée comme Julia.

5.4 Fortran

Fortran (*FORmulae TRANslate*) est un langage dédié au calcul numérique, fortement optimisé pour la manipulation de tableaux et de nombres en virgule flottantes.

Ce langage vénérable, initialement conçu pour la programmation d'ordinateurs à carte perforées, a vécu très difficilement la transition vers les ordinateurs modernes ne possédant plus les mêmes contraintes de conception [1], et n'a intégré les innovations ergonomiques de sa surcouche MATLAB qu'après plusieurs décennies. Mais cette période de transition est désormais terminée, et les versions modernes de ce langage (à partir de Fortran 90) offrent des caractéristiques très intéressantes pour les codes de calcul pur.

Contrairement à la famille C, le langage Fortran marque une distinction stricte entre pointeurs et tableaux, ce qui lui permet de détecter automatiquement davantage d'erreurs de programmation et d'optimiser davantage les manipulations de tableaux (d'une façon analogue à l'utilisation de « **restrict** » en C), tout en évitant à ses utilisateurs les pièges des pointeurs lorsque l'utilisation de ceux-ci n'est pas vraiment nécessaire.

La sémantique des opérations en virgule flottante est aussi très différente. Contrairement au C, qui vise à donner au programmeur un contrôle précis sur les opérations machine, Fortran se veut être un traducteur de formules mathématiques et autorise donc le compilateur à réordonner les opérations en virgule flottante, ce qui change les résultats. Ce comportement complique l'écriture de calculs numériquement instables, en contrepartie de quoi un compilateur peut davantage optimiser les calculs automatiquement.

Ce langage montre ses limites dans les applications de grande taille (où ses faibles capacités d'abstraction se font sentir), dans les applications qui font autre chose que du calcul flottant (de la visualisation ou des interfaces graphiques par exemple), et lorsqu'on souhaite avoir recours à des bibliothèques tierce partie (qui exposent rarement une interface vers ce langage). Il est donc à réserver à des programmes de calcul pur de taille petite à moyenne, tels que des générateurs d'événements en HEP. Lorsqu'il est utilisé dans ce cadre, il demeure compétitif aujourd'hui.

Malheureusement, la transition vers le Fortran 90 fut si longue, et l'inertie du code ancien est si grande, que le langage a acquis une image désuète et perdu beaucoup de ses pratiquants dans ce processus. Autrefois un des principaux langages de programmations utilisés, il est aujourd'hui cantonné à la niche du calcul intensif, et l'on l'enseigne de plus en plus rarement, lui préférant d'autres langages comme Python. Ce manque de popularité pourrait devenir un problème à l'avenir, car il limite le flux des nouveaux programmeurs et l'écriture des nouvelles bibliothèques.

Le Fortran est généralement compilé en binaire natif pour un système d'exploitation et une machine cible donnée. On parle de compilation statique.

1. Fortran a longtemps conservé des contraintes de conception issues des machines à cartes perforées, par exemple une limitation à des lignes de texte de 80 caractères dont certains sont réservés pour la machine, ou des contraintes très strictes sur le nombre de caractère des identifiants.

5.5 Ada

Il est extrêmement difficile d'écrire du code C ou C++ correct. Les occasions abondent, lorsqu'on utilise ces langages, d'effectuer une opération invalide au comportement indéfini comme le débordement d'un entier signé, l'accès au-delà des bornes d'un tableau, ou l'utilisation d'un pointeur vers une région mémoire déjà libérée. Ce comportement indéfini conduit à un résultat laissé à la discrétion du compilateur, du matériel, et du système d'exploitation, qui peut être désastreux ou invisible selon les circonstances d'exécution. La validation de code écrit dans ces langages est donc très complexe.

La famille Pascal, dont le langage Ada hérite, tente de résoudre ce problème en fournissant des briques de bases plus faciles à utiliser correctement et plus difficiles à utiliser de façon incorrecte, grâce à une conception plus ergonomique et à un haut niveau de vérification du programme par le compilateur. Ce faisant, Ada tente également d'offrir une grande généralité et un vocabulaire d'abstraction étendu.

Pour atteindre ces objectifs, le langage fournit un système de type de complexité comparable à celui du C++, mais qui met plus souvent l'accent sur la vérification d'erreur à la compilation au détriment parfois du confort d'écriture du code. Ainsi, les programmeurs sont par exemple encouragés à définir des types entiers ou flottants incompatibles entre eux, bien que correspondant aux mêmes types machine, afin que ce soit une erreur de compilation d'utiliser par exemple des miles là où des kilomètres sont attendus. Et le code générique doit spécifier clairement quelles sortes de données qu'il attend en entrée, ce qui le rend plus robuste mais aussi plus difficile à écrire.

Le langage se place à un niveau d'abstraction intermédiaire entre le C/++ et les langages plus haut niveau comme Python et Java. Il décourage fortement l'utilisation des pointeurs (bien que ceux-ci soient présents), et il fournit en standard des abstractions très complexes comme un support standard de la programmation concurrente avec des algorithmes d'ordonnancement et primitives de synchronisation élaborées. En revanche, sa bibliothèque standard fournit un jeu de fonctionnalités comparable à celle du C++, et la programmation idiomatique en Ada implique un souci similaire de considérations machine comme l'allocation mémoire ou la dichotomie pile/tas, ce qui peut être une bonne chose quand on souhaite obtenir de bonnes performances.

Héritage du Pascal, le langage Ada possède une syntaxe relativement verbeuse et riche en mots-clés, qui sera inhabituelle pour les programmeurs habitués à la syntaxe C. Bien qu'il soit parfois utilisé dans des situations d'enseignement, son usage à grande échelle reste cantonné à des domaines d'application bien précis comme les systèmes militaires et critiques. Par conséquent, et du fait qu'il est également relativement difficile de l'interfacer avec C ou les autres langages en général, Ada souffre comme Fortran d'un grave manque de bibliothèques tierce partie. Hélas, contrairement à Fortran, il ne bénéficie pas non plus d'une large communauté d'utilisateurs pour des tâches de calcul.

Par conséquent, bien qu'il ait pu constituer un bien meilleur substrat que le C++ pour les tâches d'informatique scientifique bas niveau, du fait de son accent sur le parallélisme, sa plus grande ergonomie pour le non-expert, et son support complet des tableaux à N dimensions de taille fixe comme bornée, il reste difficile de recommander de lancer aujourd'hui une activité de prospective sur une plus grande utilisation d'Ada à l'IN2P3.

Ada est généralement compilé statiquement vers des binaires natifs.

5.6 Rust

Comme Ada, Rust tente de rendre l'écriture de code bas niveau plus ergonomique que le C++ en évitant les comportements indéfinis et en facilitant la vérification du code par le compilateur. Cependant, il approche le problème sous un angle différent.

Dans sa syntaxe comme dans son jeu de fonctionnalités, Rust est avant tout un hybride entre C++ et les langages fonctionnels de la famille ML (OCaml, Haskell, *etc*). Il tire par exemple du premier une proximité avec la machine (distinction pile/tas, types entiers et flottants natifs, opérations mémoires atomiques et volatiles) et une implémentation similaire de l'objet et de la généricité. Et il tire par exemple des seconds un fort accent sur l'utilisation de données non modifiables, une utilisation intensive de l'inférence de type, et une intégration native des types de données algébriques (analogues à std::variant en C++ moderne, mais beaucoup plus ergonomiques).

L'approche de Rust pour la vérification de code à la compilation consiste à fournir trois garanties aux programmeurs : sûreté mémoire (absence de pointeurs invalides), de typage (absence de données invalides) et concurrente (absence de recouvrement entre l'écriture d'une région mémoire et un accès mémoire parallèle par un autre fil d'exécution). Réunies, et en combinaison avec quelques autres choix de conception, ces garanties permettent de prouver l'absence de comportement indéfini à la compilation. Les programmes sont donc bien plus faciles à écrire et à valider.

Le langage fournit ces garanties automatiquement et avec vérification à la compilation dans de nombreux cas, et permet aux programmeurs d'étendre les capacités du langage en fournissant du code pour lesquels ces propriétés ont été prouvées manuellement, utilisant un sur-ensemble du langage de base (*Unsafe Rust*) possédant un pouvoir expressif et des risques d'utilisation analogues au C. L'utilisation de ce sur-ensemble, qui est rarement nécessaire en pratique, doit être signalée par le mot-clé **unsafe**, ce qui facilite l'audit de ces régions sensibles où les bogues sont plus probables.

Pour permettre la vérification automatique des propriétés ci-dessus, Rust impose un modèle d'accès mémoire plus restrictif que la plupart des autres langages : toute référence doit pointer sur une région mémoire dont la validité peut être prouvée à la compilation, et il ne peut exister à un instant donné qu'une référence en lecture/écriture ou un ensemble de références en lecture seule vers une région mémoire. Ce modèle, analogue à celui des tableaux Fortran et des pointeurs **restrict** du C mais dont le respect est vérifié à la compilation, peut être difficile à prendre en main initialement. Mais outre les propriétés utiles ci-dessus, il permet aussi de meilleures optimisations par le compilateur.

Contrairement à Ada, Rust offre une excellente interopérabilité avec le C et les bibliothèques qui fournissent des interfaces C, via son sur-ensemble **unsafe**. En termes de niveau d'abstraction, Rust se comporte essentiellement comme un cousin de C++ avec certaines restrictions visant à rendre moins périlleuse l'utilisation courante (ex : pas d'utilisation de pointeurs bruts sans **unsafe**), certaines évolutions nécessaires qui peinent à émerger en C++ du fait de l'âge et de l'inertie du langage (ex : système de modules, types et fonctions génériques qui fixent des bornes strictes sur ce qu'ils acceptent en argument, mécanisme d'itération ergonomique, *etc*), et certaines idées nouvelles pour la communauté bas niveau qui ouvrent la voie à de nouveaux styles de programmation (ex : types de données algébriques).

Comme C++ également, Rust est un langage relativement difficile à apprendre et plutôt

destiné aux experts qui souhaitent consacrer un effort conséquent à son apprentissage. Ce défaut semble indissociable de la volonté de concilier contrôle précis de la machine et grand pouvoir expressif dans un même langage de programmation.

À l'heure actuelle, Rust est probablement le concurrent le plus sérieux à C++ pour les nouveaux développements ou réécritures où l'on est prêt à lui pardonner une certaine immaturité (fonctionnalités avancées du langage encore en cours de développement, faible nombre de bibliothèques natives), en échange de quoi on gagne en confort d'utilisation, vérification automatique du code, et plus généralement en productivité.

Rust est généralement compilé statiquement vers des binaires natifs, bien qu'il existe des prototypes de systèmes permettant de l'interpréter à la manière de ROOT.

5.7 Go

Initialement conçu par la société Google pour ses développements internes, le langage Go tente lui aussi d'investir la niche très convoitée de la programmation proche du système d'exploitation, mais avec une approche très différente de celle des langages précédents.

En effet, là où C++, Ada et Rust tentent de se placer aussi près de la machine que le C, tout en offrant des outils complémentaires pour rendre ce type de programmation plus expressif et/ou plus facile, Go prend le parti de se placer à un niveau d'abstraction un peu plus élevé afin de gagner en simplicité d'utilisation.

Ainsi, le langage utilise par exemple une gestion automatique de la mémoire (« ramasse-miettes »), ce qui diminue le contrôle du programmeur sur l'utilisation des allocateurs mémoire et le placement des données en mémoire mais offre en partie un plus grand confort d'utilisation puisque le programmeur ne doit plus se préoccuper de libérer la mémoire qu'il a allouée.

Le langage emploie une approche de conception minimaliste où une fonctionnalité n'est ajoutée au langage que s'il est rigoureusement impossible de reproduire le même comportement avec des fonctionnalités existantes. Grâce à cela, il possède un très faible nombre de concept et est facile à apprendre. Il donne également beaucoup moins de travail au compilateur, ce qui accélère les compilations.

La contrepartie de ce choix de conception est que certains types d'abstractions (ex : indépendance de la précision flottante, interfaces dont la bonne utilisation est vérifiée à la compilation) ne peuvent être exprimés par une bibliothèque Go, et que certains types de code (ex : algèbre linéaire, gestion des erreurs) sont beaucoup plus verbeux en Go que dans d'autres langages.

Comme Ada, Go fournit un support natif de la programmation parallèle et concurrente, en encourageant fortement l'utilisation d'un modèle de processus communiquants (le programme se compose d'un certain nombre de tâches concurrentes qui ne communiquent qu'en échangeant des messages). Ce modèle est très approprié à certains types de programmes comme les serveurs web, mais moins à d'autres cas d'utilisation comme le calcul scientifique où le partage direct de mémoire est souvent nécessaire à l'obtention de performances optimales. Pour ce type d'utilisation, Go fournit un ensemble de primitives de synchronisation similaire à celui du C++ moderne.

Du fait de son niveau d'abstraction plus élevé que le C, et notamment de l'utilisation de

coroutines et d'un ramasse-miettes, le langage Go est relativement difficile à interfacer avec le C et avec les autres langages de programmation en général. Par conséquent, ses utilisateurs sont généralement contraints de réimplémenter en grande partie les bibliothèques de calcul établies en C ou Fortran pour obtenir une ergonomie optimale.

Le langage Go possède à l'IN2P3 une communauté d'utilisateurs modeste mais extrêmement active, qui voit en lui un bon compromis pour éviter de devoir combiner deux langages de programmation, un langage « pour informaticien » (comme C++ ou Rust) et un langage « pour physicien » (comme Python).

Go est généralement compilé statiquement vers des binaires natifs.

5.8 Python

Le langage Python est très différent de tous les langages précédents, car il est dynamiquement typé. Cela signifie que dans un programme Python, le type des données utilisées n'est généralement pas connu par le programmeur, et peut potentiellement ne pas être connu par l'implémentation Python elle-même avant l'exécution. Il est même possible pour un programme Python de modifier son propre code pendant l'exécution.

Cette conception a certains avantages ergonomiques. En effet, si le typage des données joue un rôle crucial dans la vérification d'un programme par le compilateur et dans la production de code optimisé, l'apprentissage d'un système de type représente un effort important pour le programmeur, et l'expression précise des types utilisés par un programme peut être fastidieuse quand ceux-ci sont complexes.

En n'exposant pas les types de données au programmeur, Python peut donc être plus facile à apprendre pour le débutant, et permettre un prototypage d'applications plus rapide pour le programmeur confirmé. En contrepartie, la flexibilité du typage dynamique signifie qu'il est presque impossible de prouver qu'un programme Python est correct pour tous les types de données qu'il peut recevoir en entrée, et elle rend aussi très difficile l'exécution efficace d'un programme Python.

Plus généralement, lorsqu'un choix de conception doit être fait entre performance et facilité d'apprentissage ou ergonomie, le langage Python ne choisit jamais la performance. Les types numériques sont très éloignés des opérations machines natives, la gestion mémoire utilisée est inflexible et très peu performante, il est quasiment impossible d'utiliser le parallélisme en mémoire partagée, et le programmeur ne possède aucun contrôle sur l'organisation des données en mémoire, vis-à-vis de laquelle le langage fait des choix automatiques très peu optimaux.

En revanche, le dynamisme de Python donne aux auteurs de bibliothèques une énorme flexibilité dans les interfaces qu'ils choisissent d'exposer, sans commune mesure avec ce qu'il est possible de faire en C++ par exemple. Le langage est donc souvent utilisé pour construire des interfaces haut-niveau (tels que des *notebooks* d'analyse interactive accessibles via un navigateur web) vers un calcul implémenté de façon plus efficace dans un autre langage (en C++ par exemple).

Malheureusement, le franchissement de ces interfaces étant très coûteux au niveau machine, elles doivent être conçues de façon à donner une grande quantité de travail à faire au code sous-jacent à chaque fois qu'il est appelé. Ce surcoût, ainsi que la difficulté intrinsèque

de faire communiquer deux langages de programmation aux logiques très différentes, rend très difficile la mise au point d'interfaces entre *Python* et des langages plus optimaux pour le calcul.

Aujourd'hui, le Python est très largement utilisé pour l'enseignement du calcul scientifique, car il est facile à apprendre. Hélas, comme on a pu le voir, cette simplicité n'est obtenue qu'au prix d'une très mauvaise efficacité d'exécution, qui ne permet l'écriture de calculs performants qu'au prix de l'utilisation de systèmes à deux langages dont l'implémentation et l'optimisation est extrêmement complexe.

Ce type de système à deux langages, combinant une interface Python (« pour physiciens ») et un cœur de calcul C++ (« pour informaticiens »), est néanmoins la solution standard aujourd'hui à l'IN2P3 pour les outils ayant besoin d'être à la fois bien optimisés et accessibles aux physiciens, comme les environnements d'analyse.

Python est généralement interprété [2], mais il existe également des compilateurs pour des sous-ensembles de Python (*Cython*, *Pythran*, ***Numba***, *etc*) qui permettent d'accélérer l'exécution de certains types de codes.

5.9 Julia

La popularité grandissante du Python dans le domaine du calcul scientifique, alors que ce langage est peu adapté à cette tâche, ne peut que laisser songeur quant à l'efficacité des codes de calcul futurs. Le langage Julia tente de fournir une réponse à ce problème.

Comme Python, Julia part d'une base dynamiquement typée. Mais contrairement à Python, Julia offre un mécanisme de typage graduel, c'est-à-dire qu'il est facile pour un programmeur d'imposer progressivement des types de données explicite dans son programme lorsque c'est utile pour vérifier que le code est correct et améliorer ses performances [3].

Grâce à cette technique, le langage Julia est presque aussi simple à prendre en main que Python pour le néophyte, tout en permettant aux programmeurs confirmés de prendre progressivement le contrôle du système de type pour produire des abstractions aussi performantes qu'en C++. Pour cette raison, les auteurs de Julia le voient comme une solution possible aux problèmes des systèmes à deux langages, la totalité d'une application scientifique (de l'interface utilisateur au cœur de calcul) pouvant être en principe écrite en Julia.

Contrairement à Python, Julia considère la production de programmes efficaces comme une de ses priorités, et lui donne un poids comparable à celui de l'ergonomie dans sa conception. Ses choix de conception (comme les types numériques disponibles ou l'organisation des données en mémoire) sont donc davantage orientés vers la performance, avec pour résultat que l'on peut parfois produire des calculs aussi efficaces qu'un code C ou Fortran équivalent,

2. Lors de l'exécution, le programme est lu par un traducteur qui des opérations machines précompilées en fonction de son contenu. Ce processus évite de passer par un processus de compilation, mais ses performances d'exécution sont très mauvaises.

3. Il existe actuellement un effort pour introduire des mécanismes de typage graduels en Python, via l'extension « mypy ». Mais ce mécanisme est à un stade de développement très préliminaire, fait face à une forte opposition d'une partie de la communauté Python, et sera difficile à étendre à la grande quantité de code Python existant. Julia, en tant que langage conçu dès le départ pour le typage graduel, ne rencontre pas ces problèmes.

bien que l'on doive pour cela composer avec quelques mécanismes « haut niveau » indésirables lorsqu'on cherche à obtenir un code optimal comme l'utilisation d'un ramasse-miettes.

Le modèle de programmation de Julia se veut plus orienté vers les opérations mathématiques que celui des langages orienté objets classiques basés sur l'héritage de classe. À la place, il repose sur un mécanisme fondé sur la séparation des données et des fonctions et la spécialisation desdites fonctions par surcharge. Cette conception, perturbante pour les programmeurs habitués à d'autres langages, facilite certaines optimisations comme la spécialisation des calculs d'algèbre linéaire pour des types de matrice particuliers. Mais elle encourage aussi une organisation du code moins compartimentée qui pourrait compliquer la maintenance de gros programmes. À ce stade, le langage est trop jeune pour pouvoir dire si ce risque se concrétise ou non en pratique.

Comme Fortran, et contrairement à Python, C++ ou Rust, Julia supporte nativement les types de données classiques du calcul numérique (tableaux multi-dimensionnels, matrices, *etc*). Ceux-ci sont donc mieux intégrés au langage et plus faciles à utiliser que dans des langages où ils sont fournis par des bibliothèques.

Dans l'ensemble, le langage Julia est donc aujourd'hui un candidat sérieux à la succession de Python comme langage phare du calcul scientifique. Mais comme tous les langages récents, il dispose de moins de bibliothèques que les langages établis et celles-ci sont moins matures. Le langage lui-même est aussi en cours de finalisation, par exemple le support du parallélisme *multi-thread* vient seulement d'être ajouté en septembre 2019. Ses utilisateurs doivent donc s'attendre à affronter quelques défauts de jeunesse.

L'utilisation d'un interpréteur pur étant incompatible avec l'obtention de bonnes performances, et la compilation statique se prêtant mal aux langages dynamiques, Julia utilise un modèle de compilation à la volée où les fonctions sont compilées au moment où elles sont appelées par le programme, un peu comme en Java.

Ce modèle permet de générer des versions des fonctions spécialisées pour leurs paramètres d'entrées, qui peuvent dans certains cas être encore plus efficaces que celles d'un programme compilé statiquement. En revanche, cette compilation est aussi source de latences indésirables pendant l'exécution d'un programme Julia, qui rendent l'utilisation interactive du langage désagréable et sont problématiques pour les applications temps réel en général. Ce problème devrait être soluble à l'avenir par des mécanismes de compilation et optimisation graduelle du code analogues à ceux utilisés par les implémentations JavaScript.

5.10 Conclusions

Comme on a pu le voir, le choix d'un langage de programmation est complexe et ne saurait se résumer à une affirmation dogmatique de la supériorité d'un langage sur tous les autres. La performance n'est qu'une facette du problème, et d'autres contraintes comme l'expressivité, l'ergonomie, la compatibilité avec le code existant éventuel, la facilité de prise en main initiale pour les programmeurs non-spécialistes, la disponibilité d'experts, et la maturité de l'écosystème (langage, bibliothèques, *etc*), doivent toutes être prises en compte.

Il n'est pas possible de recommander ou même d'imaginer un langage affichant la perfection sur tous ces plans, car plusieurs de ces objectifs de conception entrent en conflit. Le choix

d'un langage relèvera donc nécessairement en partie d'un arbitrage informé entre différentes priorités, et notre rôle se bornera à émettre des recommandations pour un petit nombre de situations classiques.

Tout d'abord, une première question à poser est celle du degré d'inertie du projet logiciel dans lequel on œuvre :

— Dans des projets établis possédant une grande base de code existant, il sera souvent plus pertinent de poursuivre avec la technologie existante, sauf si des défauts rédhibitoires du langage ou du code existant justifient une réécriture totale ou partielle de l'application. Dans ce cas, un changement de langage pourra quand même être envisagé.
— À l'inverse, pour de nouveaux projets peu dépendants d'infrastructures existantes, comme des analyses physiques, l'exploration de nouveaux langages est bien plus aisée, et peut permettre d'échapper à la stagnation des écosystèmes établis.
— Entre ces deux extrêmes, il existe toute une palette de situations où le choix de changer ou non de langage dépendra fortement de la manière dont les développeurs dosent différent critères.

Une autre question qui doit être posée est celle du profil des développeurs :

— Pour des personnes dont la programmation est le cœur de métier, l'apprentissage de langages complexes mais dotés d'une grande puissance expressive comme C++ ou Rust simplifiera l'écriture de bibliothèques puissantes au code performant.
— Pour des personnes pour qui la programmation est une activité annexe, on privilégiera plutôt des langages conçus pour être faciles à prendre en main comme Go, Python ou Julia.
— Lorsque des personnes dont l'expertise en programmation est très hétérogène doivent collaborer, on aura recours à des langages qui peuvent être utilisés à plusieurs niveaux de maîtrise, comme Julia, ou à des systèmes à deux langages comme la combinaison Python/C++ lorsque l'utilisation d'outils plus établis est nécessaire.

L'échéance du projet jouera aussi un rôle important :

— Il n'est pas envisageable d'adopter une technologie peu mature, qui nécessitera un nombre important de contributions à l'écosystème du langage, lorsqu'un logiciel doit être livré sous une échéance courte.
— À l'inverse, des échéances longues comme celles de FCC donneront aux développeurs la possibilité d'envisager du travail de fond sur leur code, y compris des migrations technologiques importantes comme un changement de langage.

Et enfin, le contexte technologique pourra aussi influer sur les choix de langages disponibles. Par exemples, certains matériels de calcul comme les FPGAs n'acceptent actuellement d'exécuter que des routines écrites en C ou parfois en C++. Interagir avec ces matériels depuis un code écrit dans un autre langage nécessitera donc une base de code hybride, avec tous les soucis que cela implique. Si ces monopoles de langages disparaissent progressivement aujourd'hui, avec par exemple l'ouverture de Julia à la programmation des GPUs NVidia, tous les projets ne pourront pas se permettre d'attendre que la situation se résolve d'elle-même,

ni de contribuer à ces efforts.

L'essentiel du chapitre 5 –

Le choix d'un langage de programmation relève d'un compromis difficile, et l'informatique scientifique ne fait pas exception.
Si la combinaison de C++ et de Python, bien établie à l'IN2P3, reste appropriée dans certains cas, de nouveaux langages comme Julia et Rust promettent d'offrir à l'avenir de bien meilleurs compromis entre performance et ergonomie, à la fois pour les physiciens, les informaticiens, et leurs collaborations logicielles.
Face aux défis calculatoires sans précédents auxquels l'IN2P3 fait face, l'exploration et le développement de ces alternatives pourrait donc être une composante importante de la stratégie technologique de long terme de l'Institut.

Chapitre 6

Des FPGAs pour le calcul ?

Sujet du chapitre 6 –

Les FPGA (*Field Programmable Gate Arrays*) sont des circuits intégrés programmables utilisé dans une large gamme de cartes électroniques pour le traitement des signaux numériques, en particulier ceux résultants de la lecture des détecteurs de particules. Les FPGA sont préférés aux ASIC (*Application Specific Integrated Circuit*, les FPGA sont parfois utilisés pour leur prototypage) quand le traitement dépasse un certain niveau de complexité, mais surtout si les fonctions implémentées nécessitent une reconfiguration sur-le-champ (*field programmable*) et à répétition. C'est cet avantage de pouvoir être reprogrammé et la variété des configurations obtenue avec un grand nombre de portes logiques interconnecté dans un réseau (*gate array*) qui recommandent les FPGA comme des possibles unités dédiées au calcul au sens plus large.

Sommaire

6.1 FPGA : généralités 82
6.1.1 Qu'est-ce qu'un FPGA ? 82
6.1.2 Comment programmer un FPGA ? 82
6.1.3 Programmer des calculs sur le FPGA 83
6.2 Calcul sur FPGA avec OpenCL 86
6.2.1 Le matériel 86
6.2.2 La programmation 89
6.3 Évolution des accélérateurs avec FPGA Altera/Intel 92
6.3.1 Solutions Intel avec OpenVINO pour l'apprentissage automatique . 93
6.4 Accélération avec FPGA Xilinx sur la plate-forme ACP du Laboratoire Leprince-Ringuet 94

6.1 FPGA : généralités

6.1.1 Qu'est-ce qu'un FPGA ?

Le FPGA est un composant électronique qui fait partie de la famille des circuits logiques programmables (c'est-à-dire programmables après leur fabrication). Un FPGA contient un grand nombre de portes (ou cellules) logiques qui constituent des briques élémentaires nécessaires à la construction de schémas logiques plus complexes. La particularité d'un FPGA est la possibilité de définir le routage entre les différents éléments logiques dans la phase de conception et créer ainsi un *processeur sur mesure*, adapté à une tâche de calcul particulière.

Un bloc logique est généralement constitué d'une table de correspondance (*look-up-table*, LUT) qui peut effectuer des opérations logiques simples sur 4 à 6 entrées. Ces blocs logiques en très grand nombre (jusqu'à 10 millions) sont connectés entre eux par une matrice de routage configurable, où chaque connexion peut être rendue fermée ou ouverte par le biais d'un circuit de type bascule (*flip-flop*, FF). La matrice de routage est gérée par une passerelle informatique spécifique (Avalon, pour les FPGA Altera/Intel), qui occupe une surface assez importante sur le silicium. Le circuit FPGA fonctionne comme une mémoire volatile, alors il est nécessaire de sauvegarder son design sur un support externe non-volatile (comme une mémoire flash), d'où la même configuration peut être à nouveau chargée sur le FPGA à la demande (équivalent à un *reset* chaud du FPGA). La communication de configuration avec le FPGA se fait par un protocole appelé JTAG (*Joint Test Action Group*), capable de lire un fichier *image* binaire et effectuer la configuration du routage et des blocs logiques.

6.1.2 Comment programmer un FPGA ?

La programmation d'un FPGA est d'abord une spécialité du domaine de la micro-électronique. Un projet FPGA commence avec la spécification exacte du type de FPGA : nombres d'éléments logiques, nombre et emplacement des connecteurs, *etc.* L'outil de programmation et de synthèse pour les FPGA fabriqués par Altera/Intel est *Quartus* et pour les FPGA de la marque Xilinx (la compagnie américaine inventeur du FPGA) *Vivado*. Dans ce document, référence sera faite principalement aux FPGA de la marque Altera (actuellement Intel), qui ont été testés pour des tâches de calcul, dans le cadre du projet Reprises.

Quartus s'installe également sous Windows et sous Linux (CentOS 7.4 et Ubuntu 18.1 pour ce rapport) et il est disponible sur le site Intel, sans (*lite*) ou avec licence (*standard* ou *pro*). Les licences Quartus sont mises à jour sur un serveur de licences flottantes au Centre de Calcul de l'IN2P3 et sont utilisé de cette manière dans tous les services de micro-électronique des laboratoires de l'IN2P3.

La programmation d'un FPGA va de pair avec la description du hardware sur lequel le code sera exécuté. Dans le design hardware d'un projet FPGA, on utilise aussi des bibliothèques de modules préfabriqués qu'on insère dans notre projet après une configuration et un paramétrage partiel (le catalogue de cœurs *Intellectual Property*, ou *IP cores*). Pour cette raison, ce type de langage s'appelle un langage de description de hardware (*Hardware Description Language*, HDL) : Quartus offre des implémentations de Verilog et VHDL. Dans la partie software, la modélisation abstraite de la circulation des signaux dans des circuits numériques ainsi réalisés est faite dans un langage appelé RTL (pour *Register Transfer Level*).

La compilation d'un projet comporte aussi une analyse temporelle du signal propagé dans le circuit et Quartus contient des outils de simulation de la réaction du système aux stimuli externes.

L'une des dernières étapes du design d'un projet FPGA est l'association des connecteurs externes du FPGA. Ainsi le FPGA communique avec d'autres périphériques, on injecte et on extrait des signaux, *etc.* en fonction de la destination du projet. Dans le cas d'un système d'acquisition pour un détecteur de particules, le FPGA sera toujours monté avec plusieurs autres composantes électroniques sur la même carte, qui assureront l'alimentation, l'injection et l'extraction des données, les signaux horloge de synchronisation, le câble pour la mise à jour du firmware, *etc.* La dernière étape, celle de la compilation, intègre toutes les étapes précédentes et synthétise l'image binaire qui sera transférée dans le FPGA (en général par une connexion JTAG) ou vers la mémoire flash attachée au FPGA. La compilation est un processus gourmand en CPU et mémoire vive et peut prendre plusieurs heures, principalement en fonction de la taille du FPGA. L'image est un fichier de plusieurs dizaines de méga-octets ou plus et on appelle ça le *firmware*, car c'est une combinaison de software et hardware.

6.1.3 Programmer des calculs sur le FPGA

Les fabricants de FPGA et leurs partenaires ont crée des catalogues de blocs IP d'une large variété, qui ont facilité, par exemple, le design de systèmes complets avec une architecture von Neumann, avec un processeur basique, beaucoup exploités dans une multitude d'applications embarqués (*System on Programmable Chip*, SoPC). La possibilité d'exécuter des tâches de calcul est évidente dans ce cas, mais c'est un autre avantage des FPGA qui est plus prometteur : la possibilité de parallélisation des opérations arithmétiques et celle d'exécution en *pipelines* plus profonds que sur les architectures CPU classiques. Dans ce cas, le FPGA doit être d'abord interfacé avec un système hôte, par des blocs IP, par exemple pour contrôler la mémoire où les données seront échangées, ou autres interfaces physiques. Ensuit nous avons les blocs DSP (*Digital Signal Processor*) qui intègrent des fonctionnalités plus proches de la représentation des opérandes et des opérations arithmétiques, par exemple avec les formats flottants. Le tout sera dans ce cas décrit dans le langage VHDL, qui peut être considéré, dans ces circonstances, comme un langage de programmation parallèle.

L'écriture d'un *programme* en VHDL ou Verilog pour un FPGA présente certains aspects de la programmation directe en assembleur d'un processeur classique. Elle ressemble aussi à la programmation d'un micro-contrôleur, mais elle n'est en aucun cas un langage de haut niveau directement utilisable par des programmeurs non-spécialisés en micro-électronique. L'utilisation des langages de type HDL serait difficilement mise en accord avec l'idée de programmation en tant qu'outil accessoire de recherche pour les chercheurs avec une formation en physique, accompagnée d'une formation plus ou moins approfondie en programmation *classique*, avec les langages de haut niveau connus sur le marché, plus proches de l'expression mathématique et algorithmique que des détails de fonctionnement hardware de la machine de calcul.

Pour faire du FPGA un candidat populaire pour les utilisateurs de ressources de calcul, il a fallu créer le niveau d'abstraction qui cache les détails d'implémentation hardware et un compilateur qui supporte une interface de programmation confortable (similaire au langage C ou C++). Une différence reste pourtant, dans le fait qu'il n'existe pas de set d'instructions

élémentaires pour un tel équivalent de processeur. Cela signifie que la compilation (ou la synthèse) de l'image FPGA est toujours basée sur une description (un projet) de support, où les *prémisses* de communication des données et de leur manipulation arithmétique ainsi que toutes les autres interfaces physiques nécessaires sont déjà synthétisées dans une image de base (une sorte de pilote). Cette image de base (*Board Support Package*, BSP) est spécifique au type de FPGA avec toutes ses caractéristiques ainsi qu'à l'environnement de montage du FPGA, car le FPGA est en général présenté sur une carte avec toute la connectique nécessaire et les périphériques adjacents. La réalisation de cette image de support reste à la charge des spécialistes, elle peut évoluer et s'améliorer avec les versions successives du logiciel de synthèse (Quartus pour les FPGA Altera/Intel) et représente un projet spécifique à un type précis de FPGA. Néanmoins, si certains FPGA seront mieux ciblés pour le calcul hybride (CPU + FPGA), une standardisation serait toujours possible et la production des BSP pourrait être organisée dans des conditions optimales.

Dans les sections suivantes nous allons détailler l'approche suivi dans le cadre du projet Reprises pour l'étude d'une carte équipée d'un FPGA utilisée comme accélérateur de calcul lorsqu'elle est connectée à un PC hôte.

6.1.3.1 Parallélisme sur GPU et sur FPGA

Le principe de combiner le parallélisme avec le pipeline de l'exécution des instructions et expliqué dans un article technique disponible sur le site Intel [56]. Les unités graphiques de calcul (GPU) sont bien connues et depuis longtemps utilisées pour faire des calculs (*General Purpose Graphic Processing Units*, GPGPU). Initialement dédiées aux calculs nécessaires pour l'affichage graphique à l'écran, leur architecture est orientée vers l'application de la même opération (une transformation géométrique, par exemple) à la fois sur plusieurs valeurs (coordonnées des points en espace 3D ou 2D). Ceci est connu sous le concept de *Single Instruction Multiple Data* (SIMD) et représente l'une des approches du calcul parallèle.

La spécificité des GPU fait que par cycle d'horloge les unités actives exécutent la même instruction. En conséquence, pour une donnée (couleur dans le diagramme), le calcul sera complet toujours après 5 cycles, correspondants aux 5 instructions à appliquer. Dans cet exemple on étudie le cas pour 6 données présentées au GPU et au FPGA (6 couleurs différentes). Le GPU peut exécuter 3 items en mode SIMD. Dans le FPGA, les items exécutés en parallèle appliquent des instructions différentes sur des données différentes. Avec cette configuration, le traitement des 6 données est complet après 10 cycles d'horloge.

Le FPGA a l'avantage que les différents noyaux (qui exécutent, dans cet exemple, une des 5 instructions) sont compilés dans des circuits hardware séparés. Par cet artifice, sur le premier cycle l'item 1 exécute l'instruction A sur la donnée *bleue*. Sur le deuxième cycle, l'item 1 continue avec l'instruction B sur la donnée *bleue* et l'item 2 exécute l'instruction A sur la donnée suivante, *verte*. Sur le troisième cycle, l'item 3 exécute l'instruction A sur la donnée *mauve*, pendant que l'item 2 peut exécuter l'instruction B sur la donnée *verte* (la deuxième du flux des données) et l'item 1 continue sur la *bleue* (la première du flux de données) avec la troisième instruction, C. Et ainsi de suite.

Même si dans cet exemple on a l'égalité entre le GPU et le FPGA au bout de 10 cycles, on observe qu'à partir du septième cycle d'horloge, les items du FPGA commencent progressivement à ne plus être utilisés. Si on continue le flux des données, le GPU va compléter tout

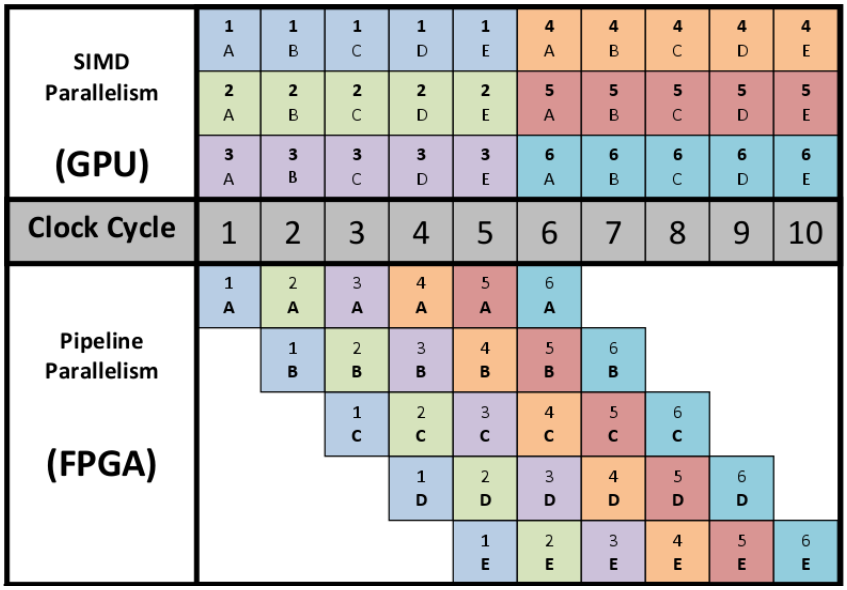

FIGURE 6.1 – Le parallélisme de type pipeline dans les FPGA : une comparaison avec le SIMD des type des GPU

les 5 cycles le calcul sur 3 données d'entrée, pendant que les items du FPGA exécuteront en moyenne toutes les 5 instructions sur le même cycle d'horloge.

Un autre avantage du FPGA par rapport au GPU devient visible quand une instruction (ou un noyau) produit un résultat qui implique une bifurcation de l'exécution suivante, en fonction de la valeur d'entrée. Par exemple, si l'instruction A est exécutée sur le même cycle pour trois données (bleue, verte, mauve) mais la séquence des trois instructions suivante et ramifié en (B_1, C_1, D_1), (B_2, C_2, D_2) et (B_3, C_3, D_3), étant donné que les items synchrones d'un GPU exécutent toujours la même instruction (on ne peut pas avoir B_1, B_2 et B_3), il en résulte que le temps total pourrait s'allonger. Si l'exécution sur la donnée bleue continue avec la séquence (1), celle sur la donnée verte avec la séquence (2) et la mauve avec la séquence (3), le nombre total de cycle d'horloge sera $3 \times 3 = 9$, au lieu de trois sans bifurcation (figure 6.1, voir [56]). Dans le FPGA chaque cycle est utilisé, car B_1, B_2 et B_3 seront compilés dans de circuits séparés et le routage des données à la sortie de A est fait selon le branchage spécifiée dans le code (*if*... *then*).

6.2 Calcul sur FPGA avec OpenCL

Ouvrir le monde du FPGA à l'utilisation d'un langage de programmation de haut niveau a été d'abord une décision des fabricants de FPGA. Altera a commencé en 2011 en choisissant OpenCL (*Open Computing Language*), un cadriciel de programmation parallèle crée par le groupe Khronos en 2008 [57], peu de temps après l'ouverture des GPU au calcul générique avec le langage CUDA par le fabricant de cartes graphiques NVidia. Beaucoup inspiré par ce dernier, OpenCL avait comme cible d'abord les processeurs multi-cœur, mais NVidia a aussi crée l'environnement nécessaire pour pouvoir exécuter du code OpenCL sur les GPU. De cette façon, très vite OpenCL a mis en évidence son ambition de langage de programmation des systèmes parallèles sur des ressources hétérogènes.

Le langage OpenCL [58] est un dérivé du langage C, avec une API (*Application Programming Interface*) en C++ à partir de la version majeure 2). L'environnement de programmation (*Software Development Kit*, SDK) et celui d'exécution (*Run Time Environment*, RTE) pour le CPU sont téléchargeables depuis le site Intel et s'installent facilement.

Un code OpenCL est composé d'une partie qui s'exécute sur le processeur hôte (s'il s'agît du OpenCL pour CPU, juste exécution en mode *natif*), la partie écrite en C (ou C++) standard, appelée partie hôte, qui appelle des fonctions de l'API OpenCL pour configurer, lancer et contrôler l'exécution d'une deuxième partie du code, écrite en langage OpenCL, qui est la partie *device*. Cette partie *device* contient un ou plusieurs noyaux de code qui peuvent s'exécuter en plusieurs copies identiques sur le même *device* (plusieurs *devices* peuvent être gérés en même temps, s'ils sont correctement insérés dans le système hôte).

La programmation OpenCL, avec des exemples bien choisis, est très bien présentée dans un livre largement connu [59]. Les codes utilisés dans le matériel du livre sont censés être exécutés sur CPU ou GPU, mais, dans les études de faisabilité menés au Laboratoire de Physique de Clermont, ces codes ont été adaptés pour l'exécution sur FPGA (pour la plupart des exemples, les modifications restent, toutefois, minimales).

6.2.1 Le matériel

Pour les premières études de principe, le FPGA choisi a été celui fourni par la carte de développement DE1-SoC [60]. Sur cette carte, le FPGA contient sur la même puce un vrai (*hard*) processeur ARM Cortex A9 avec 2 cœurs, formant ce qu'on appelle un *System on Chip* (SoC). Avec une mémoire permanente amovible sous la forme d'une carte Micro-SD, le tout peut accueillir un système d'exploitation Linux. Pour les applications OpenCL, il existe une passerelle directe entre le microprocesseur ARM et le FPGA.

Cette carte (figure 6.2) est un outil de laboratoire pour de nombreuses études qui impliquent l'utilisation d'un FPGA : des applications audio, vidéo, robotique, capteurs et autres périphériques (LEDs, interfaces de données) et le fabricant offre un nombre de projets pour démarrer dans la programmation HDL d'un FPGA. La carte s'alimente séparément et peut recevoir un écran, un clavier et une souris, afin de travailler sur le système installé dessus.

La compilation d'un projet OpenCL doit se faire sur un PC avec beaucoup plus de ressources que cette carte, car il faut d'abord installer Quartus (qui occupe à l'installation quelques dizaines de giga-octets). Ici il faut aussi installer le SDK OpenCL. Sur le système installé sur la carte DE1-SoC, nous avons besoin du pilote OpenCL pour le FPGA (compilé

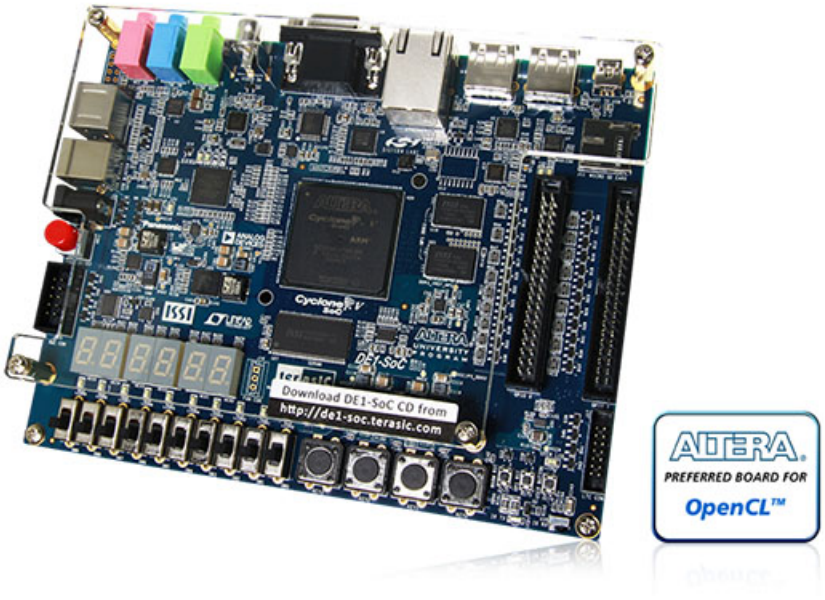

FIGURE 6.2 – La carte de développement DE1-SoC assemblée et vendue par l'entreprise Terasic, contient un FPGA Cyclone V SE 5CSEMA5F31C6N

comme module dans le noyau Linux) et du RTE OpenCL. La compilation de la partie hôte du code (pour une architecture ARM) peut se faire avec un compilateur croisé (*cross compiler*) et à la fin l'exécutable et l'image FPGA seront transférés vers le système sur la DE1-SoC à travers le réseau Ethernet. Le RTE du OpenCL offre aussi un outil pour charger l'image binaire sur le FPGA de la DE1-SoC.

Cette première version de travail a été très utile pour valider la méthode, mais elle reste inadaptée comme solution d'accélération sur une architecture de calcul habituelle. L'idée est d'avoir accès au FPGA directement depuis le PC hôte et ceci est possible avec un autre type de cartes, qui sont montée sur la passerelle PCIe (*Peripheral Component Interconnect*). Cette connexion assure l'alimentation de la carte, l'échange des données entre la mémoire du PC hôte et la mémoire accessible par le FPGA et même le chargement de l'image binaire sur le FPGA.

Il a été précisé auparavant que pour développer un projet OpenCL sur un FPGA nous avons besoin d'une image de base (BSP) qui contient déjà les circuits de base et d'interfaçage pour pouvoir configurer les noyaux OpenCL selon les spécifications du code. Pour la carte DE1-SoC, le fabricant met à disposition un BSP pour la version 16.0 de Quartus (et du SDK OpenCL), la dernière avant l'acquisition d'Altera par Intel. Ça signifie qu'on ne peut pas exploiter cette carte avec des versions ultérieures du SDK. Pour la carte Cyclone V GT (figure 6.3), montable sur le PC hôte par un connecteur PCIe, le seul BSP trouvé a été celui crée par un développeur FPGA (Sethuraman Kuppuswamy [61]), mis en ligne en mai 2017, pour la version 17.0 du SDK OpenCL (la version la plus récente est 18.1). Depuis, la situation a évolué et dans une autre section nous allons montrer quelles sont les configurations actuellement privilégiées par Intel, en tant que fabricant de FPGA, par son *Programming Solutions Group*.

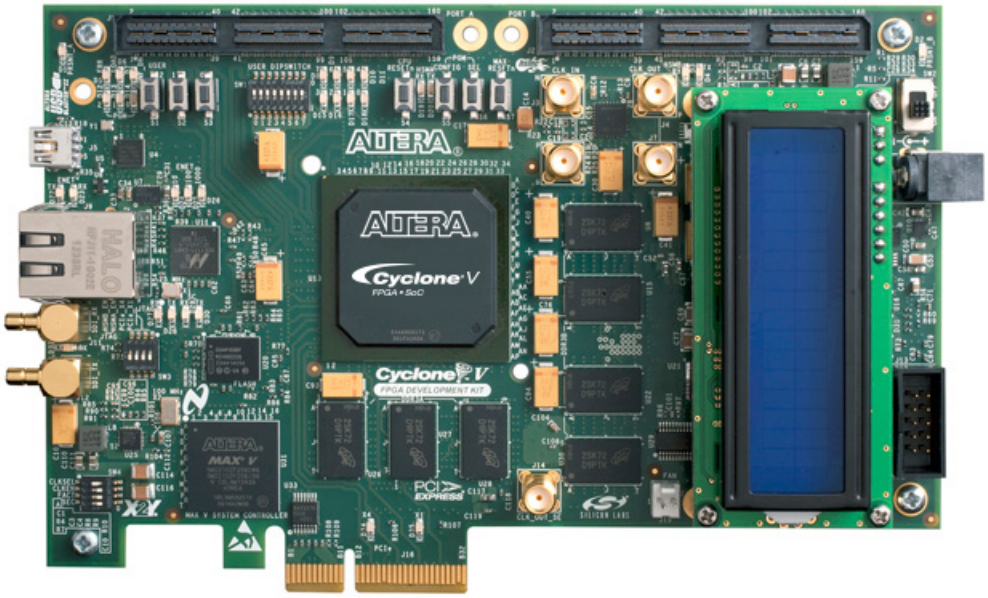

FIGURE 6.3 – La carte Cyclone V GT, avec un FPGA 5CGTFD9E5F35C7N

Le FPGA de la deuxième carte est aussi plus performant que celui de la carte DE1-SoC. Le tableau 6.1 montre quelques caractéristiques des deux FPGA utilisés. Quelques aspects essentiels pour la capacité liée à la programmation OpenCL en ressortent :

1. le nombre d'éléments logiques donne la capacité d'insérer des circuits hardware logiques et de les connecter selon le flux de données explicité dans le code ; ceci s'exprime aussi en nombre de registres, de modules flexibles ALM (*Adaptive Logic Module*) et de bascules FF
2. en plus des éléments logiques, des structures plus spécialisées déjà insérées dans la structure du FPGA, comme les DSP ou les multiplicateurs, peuvent être utilisés dans la synthèse du noyau OpenCL en image hardware
3. le grade de rapidité est en relation avec la vitesse de commutation des circuits de type bascule FF, c'est-à-dire avec la vitesse de fonctionnement du FPGA en général : plus bas l'indice, plus rapide le FPGA
4. le FPGA dispose d'une mémoire interne, organisé en petits blocs de mémoire répartis sur la surface du FPGA pour optimiser les accès ; le standard OpenCL manipule les données en trois niveaux hiérarchiques de mémoire : globale, locale et privée, en fonction du temps d'accès, le partage et le contrôle des opérations d'accès par des barrières

La carte Cyclone V GT est montée sur la carte mère du PC hôte dans une prise PCIe libre (elle peut être plus large que $\times 4$) et ensuite un câble USB doit relier l'entrée de programmation JTAG de la carte (visible en extérieur à l'arrière du PC) avec une prise USB du même PC hôte. Ce câble servira au chargement de l'image binaire FPGA obtenue après la compilation, qui se fera au moment de l'exécution du code hôte par l'appel d'une commande

Caractéristique	Carte DE1-SoC	Carte Cyclone V GT
Code FPGA	5CSEMA5F31C6N	5CGTFD9E5F35C7N
System on chip (SoC)	oui	non
Processeur hard	ARM Cortex A9	-
IP hard pour contrôle de mémoire	1	2
IP hard pour connexion PCIe	0	2 (PCIe x4)
Nombre d'éléments logiques (LE)	85k	301k
Nombre de connecteurs	896	1152
Grade de rapidité	6	7
Mémoire interne (M10K)	3970	12200
DSP de précision variable	87	342
Multiplicateurs 18x18	174	684
Prix [Euro]	250	1100

TABLE 6.1 – Un extrait des caractéristiques des deux FPGA en test : DE1-SoC et Cyclone V GT [62]

Quartus. Avec la BSP disponible (voir plus haut) il n'est pas possible de charger cette image par la passerelle PCIe, aspect qui diminue la vitesse de chargement, mais, vu que pour une seule tâche de calcul avec un seul noyau compilé l'image est transféré une seule fois au début, la performance de calcul n'est pas altérée.

6.2.2 La programmation

Un bon nombre d'exemples du livre [59] ont été adaptés (si nécessaire), compilés et exécutés sur la carte DE1-SoC mais surtout sur la Cyclone V GT, toujours avec la réserve que certaines fonctionnalités OpenCL n'étaient pas disponibles pour les FPGA à l'étude, au moins avec les BSP utilisés. L'exemple le plus complexe qui a été le mieux analysé est la Transformée Fourier Discrète (TFD), dans sa version rapide (TFDR, à ne pas confondre avec la transformée rapide FFT, *Fast Fourier Transformation*).

La Transformée de Fourier Discrète d'un signal échantillonné est donné par la formule :

$$X_k = \sum_{n=0}^{N-1} x_n e^{-i2\pi kn/N} \quad , \quad k = 0 \dots (N-1) \tag{6.1}$$

où on peut considérer pour la simplicité que x_n sont des nombres réels. La transformée est exprimée par les nombres complexes X_k, avec une partie réelle et une partie imaginaire. Les valeurs en module $|X_k|$ donnent les composantes de la puissance spectrale. Les valeurs x_n seront générées de façon aléatoire dans un intervalle fixe. La longueur N est variée, mais

gardée toujours à une puissance entière de 2 et pour chaque longueur un grand nombre d'échantillons est généré, pour avoir une statistique sur le temps d'exécution.

La solution directe et sans aucune optimisation est celle-ci :

```
for (k = 0 ; k < N ; ++k) {
 Xre[k] = 0;
 Xim[k] = 0;
 for (n = 0 ; n < N ; ++n) {
  Xre[k] += x[n] * cos(n * k * PI2 / N);
  Xim[k] -= x[n] * sin(n * k * PI2 / N);
 }
}
```

Dans ce code on effectue N^2 calculs de composante. La version rapide de la transformée est basée sur le fait qu'on peut exprimer la transformée comme une somme de deux termes :

$$X_k = E_k + e^{-\frac{2\pi i}{N}k} O_k \quad , \quad X_{k+\frac{N}{2}} = E_k - e^{-\frac{2\pi i}{N}k} O_k \qquad , \quad k = 0 \ldots (\frac{N}{2} - 1)$$

avec E_k et O_k les transformées de la partie du signal x_n formée par les index pairs et les index impairs, respectivement. Le procédé de partition paire - impaire continue jusqu'à la longueur triviale du vecteur de valeurs. Le code pour la variante CPU qui accompagne le livre [59] se trouve ici [63].

Les résultats présentés dans la figure 6.4 ont été obtenus avec la dimension N variée de 128 (2^7) à 8192 (2^{13}). Pour chaque dimension N, le temps d'exécution a été moyenné sur 100 itérations, avec l'image FPGA binaire chargée une seule fois au début. Plus le temps est long, moins le calcul est efficace en rapidité d'exécution. La courbe bleue, linéaire en échelle double logarithmique, représente la dépendance N^2 de l'algorithme direct de calcul de la TFD. Sur le même graphique on peut voir les résultats obtenus avec la variante TFDR parallélisé avec OpenCL sur trois périphériques de calcul : CPU (Intel Xeon E5-1607, 3GHz), GPU (NVidia Quadro 2000M, 1.1GHz) et FPGA (Cyclone V GT). Grâce à l'astuce de calcul rapide de la TFD, le nombre de items qui s'exécutent en parallèle est la moitié de la longueur N du vecteur à transformer. Les résultats avec le GPU s'arrêtent à $N = 2048$, limités par la capacité de cette carte graphique (*maximum number of threads per block* = 1024).

On peut voir dans le graphique que dans cet exemple relativement simple (sans branchement dans l'exécution des instructions à l'intérieur du noyau, juste des boucles *for*) le FPGA peut être parfois un peu plus efficace que le CPU avec OpenCL et qu'il peut concurrencer un vieux type de GPU. Son avantage par rapport au GPU consiste dans le fait qu'il peut passer à l'échelle le nombre d'items d'exécution pour des valeurs plus grandes que N.

Outre l'observation du temps total d'exécution, une étude plus détaillée a été menée pour séparer, par exemple, le temps de transfert des données à transformer vers la mémoire interne du FPGA. Ceci peut évidemment pénaliser la performance globale et il dépend essentiellement de la gamme du matériel utilisé, par exemple la vitesse de la connexion PCIe.

OpenCL utilise aussi un autre type d'accélération, basé sur la vectorisation des opérandes et des opérations élémentaires appliquées dessus. Les types de vecteurs sont dérivés des types

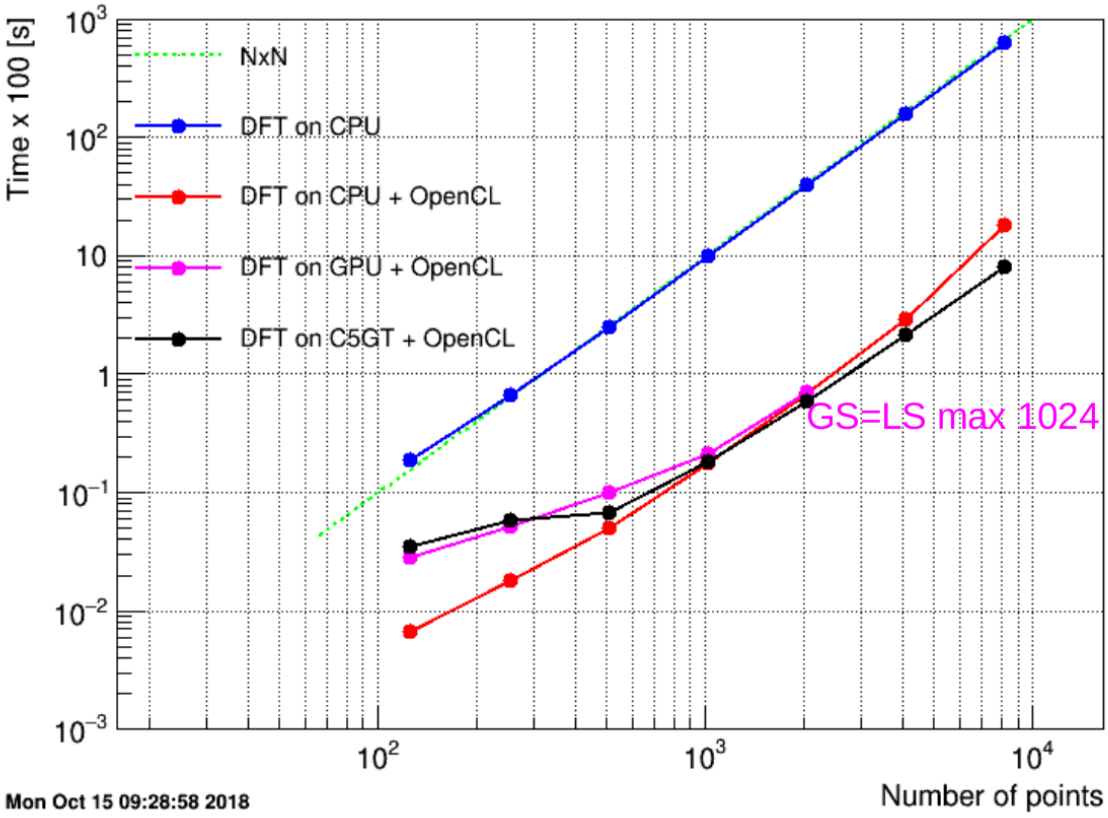

FIGURE 6.4 – La Transformée Fourier Discrète Rapide parallélisée avec OpenCL sur CPU, GPU et FPGA

de scalaires (pas toutes) et donnent : char$n$, uchar$n$, ..., float$n$, avec valeurs possibles $n = 2, 3, 4, 8$ et 16 (la largeur de vecteur préférée est indiquée par l'implémentation particulière dans le *device* ciblé). Le cas de la double précision dépend essentiellement du *device* utilisé, pour les FPGA en étude il n'est pas possible de faire des calculs en double précision.

Une opération avec vecteurs s'écrit en OpenCL :

```
float4 data_vec_1 = (float4) (1.4, 2.3, 3.2, 4.1);
float4 data_vec_2 = (float4) (0.1, 0.2, 0.3, 0.4);
float4 data_vec_3 = data_vec_1 + data_vec_2;
```

et les composantes du vecteur résultant en $float$ scalaire seront accessibles par :

```
data_vec_3.s0, data_vec_3.s1, data_vec_3.s2, data_vec_3.s3
```

FIGURE 6.5 – Carte d'accélération Intel avec FPGA Aria 10 GX

6.3 Évolution des accélérateurs avec FPGA Altera/Intel

Depuis l'acquisition du fabricant de FPGA Altera, Intel a développé une politique autour du FPGA comme accélérateur de calcul par son groupe de solution de programmation PSG. Il semble que les activités se concentrent dans deux directions :

1. le choix d'une carte équipé d'un FPGA de référence, avec tout le support nécessaire mis à jour pour les dernières versions de logiciel
2. l'intégration d'un FPGA sur un processeur Intel et l'extension du compilateur propriétaire *i++* avec des fonctionnalités OpenCL

En ce qui concerne le premier point, Intel s'est arrêté sur le FPGA Arria 10 GX [64] (figure 6.5). Selon la fiche technique, les applications ciblées sont l'analyse des grandes données, l'intelligence artificielle, le calcul de haute performance, la sécurité informatique, *etc.* Quelques caractéristiques de cette carte PAC (*Programmable Acceleration Card*) sont présentées dans le tableau 6.2, la carte est validée par Intel pour les serveurs Dell R740 et R640 avec le système d'exploitation CentOS 7.4. En octobre 2018, une carte PAC avec un FPGA d'une autre famille de produits, Stratix 10 SX, a été annoncée par Intel [65].

Intel a aussi crée une suite de logiciels pour l'extension de calcul d'un processeur Xeon avec une carte d'accélération FPGA [66], basée sur des SDK et RTE OpenCL (en version 1.2 du décembre 2018 pour la carte Aria 10 GX). Dans les versions Quartus supérieures à 17.0 (Intel), un compilateur *i++* est disponible pour la compilation des codes en mode HLS (*High Level Synthesis*), utilisé à la fois pour la compilation du code hôte et pour celle du code

Caractéristique	Intel PAC Aria 10 GX	D5005
Code FPGA	10AX115N2F40E2LG	1SX280HN2F43E2VG
IP hard pour contrôle de mémoire	16	?
IP hard pour connexion PCIe	4 (PCIe x8, x16)	4 (PCIe x16)
Nombre d'éléments logiques (LE)	1150k	2800k
Nombre de connecteurs	1517	1760
Grade de rapidité	2	2
Mémoire interne (M20K)	54260	11721
DSP de précision variable	1518	?
Multiplicateurs 18x19	3036	11520
Prix [Euro]	6100	?

TABLE 6.2 – Intel PAC Aria 10 GX et Stratix SX (D5005)

noyau à exécuter sur l'accélérateur FPGA, en indiquant selon le cas le type d'architecture à utiliser par une simple option de compilation.

En mai 2018, Intel fait sortir son multi-processeur *scalable* avec un FPGA Arria 10 GX intégré [67,68], avec un prix recommandé autour de $5000, mais peu de retour est disponible pour le moment par rapport aux autres solutions d'accélération. Cette configuration rappelle celle de *system on chip* de la carte DE1-SoC avec un processeur ARM.

6.3.1 Solutions Intel avec OpenVINO pour l'apprentissage automatique

Dans le domaine de l'apprentissage automatique avec des images (vidéo surveillance, *etc.*), Intel a développé une API commune pour le CPU, GPU, VPU (*Video Processing Unit*[1]) mais aussi pour les FPGA. L'accélération est implémentée dans sa phase d'inférence, sur un modèle d'apprentissage déjà entraîné, avec une large compatibilité envers les modèles existants sur le marché (TensorFlow, Caffe, MXNet, *etc.*). Le fabricant Terasic a une offre autour de $520 pour une carte de développement équipée d'un FPGA Cyclone V GX (similaire à la version GT), avec un BSP fourni pour la version 17.1 de Quartus et la version OpenVINO 2019 R1.

Bien sûr le *toolkit* OpenVINO n'est pas limité aux images. Par exemple, si le modèle d'apprentissage est basé sur un réseau de neurones, une fois le modèle optimisé sur les données d'apprentissage et vérifié sur celles de test, sa topologie est *exporté* dans un fichier XML. Les outils de OpenVINO, à partir de ce fichier, génèrent une image binaire avec la description topologique du réseau, qui sera ensuite *exécutée* sur le CPU, GPU, VPU ou FPGA avec les

1. Le VPU développé par Movidius est basé sur une micro-architecture multi-parallélisée qui fonctionne un peu comme un CPU ou un GPU, avec un set d'instructions fortement orienté vers le parallélisme des données. Le processeur Intel Movidius Myriad 2 est largement utilisé dans l'industrie dans le domaine de la vision autonome.

nouvelles données inconnues.

6.4 Accélération avec FPGA Xilinx sur la plate-forme ACP du Laboratoire Leprince-Ringuet

Les résultats des études effectuées au Laboratoire de Physique de Clermont ont été présentés dans les réunions du projet Reprises [69], avec aussi une démo dans une réunion au Centre de Calcul de l'IN2P3. Ces études se sont déroulées pendant la période de l'acquisition par Intel du fabricant de FPGA Altera, quand Intel a commencé à définir ses orientations en matière d'accélérateur avec FPGA.

En 2019 la décision a été prise d'équiper un nœud de calcul avec un accélérateur FPGA, sur la plate-forme ACP (*Accelerated Computing in Physics*) au LLR. La solution retenue a été celle d'un serveur Dell PowerEdge R7245 avec deux processeurs AMD EPYC et une carte d'accélération Alveo 280P (P = modèle en pré-production) équipée avec un FPGA Xilinx ([70], figure 6.6). Les ressources en logiciels pour l'exploitation de cette carte (SDAccel [71]) contiennent les compilateurs pour l'hôte et pour le FPGA, sur la base de l'API OpenCL et le Xilinx *run time* XRT spécifique pour cette carte. Ces ressources sont compatibles avec les systèmes d'exploitation CentOS 7.4, 7.5, 7.6 et Ubuntu 16.04 LTS, 18.04 LTS. L'environnement SDAccel est aussi présent sur des instances AWS F1 d'Amazon (*Amazon Web Services*, un service de *cloud*) et sur des instances pour le calcul élastique EC2 [72].

FIGURE 6.6 – Carte accélérateur pour centre de données, Alveo U280. La version 280P de la plate-forme ACP sera un modèle en pré-production

L'équivalent de Quartus pour les FPGA Xilinx est la suite de logiciels Vivado.

Les FPGA Xilinx ont des architectures différentes par rapport à leurs concurrents Altera/Intel. Dans la carte Alveo U280P, le FPGA utilisé est optimisé pour fonctionner exclusivement avec cette configuration. Dans les solutions Intel, le même type de FPGA peut fonctionner sur une carte de développement (utilisée, par exemple, par les développeurs en micro-électronique pour les détecteurs de particules) et aussi sur un accélérateur spécialisé.

L'essentiel du chapitre 6 –

Les changements de paradigme dans le domaine de la programmation, avec les prospectives de l'utilisation des architectures de programmation qui s'éloignent de l'architecture classique de von Neumann, doivent tenir compte du fait que les chercheurs physiciens ont depuis toujours été les auteurs de la plupart de leurs codes de simulation et analyse, en s'appuyant sur des langages de programmation de haut niveau relativement facile à apprendre et à utiliser pour des performances plus ou moins satisfaisantes dans la plupart des situations.
Les optimisations de code deviennent de plus en plus critiques avec le pronostic de la loi de Moore et l'inclusion d'autres architectures, comme les GPU, VPU, TPU (*Tensor Processing Unit*) et FPGA semble de plus en plus naturelle, vu les stratégies abordées par les fabricants de hardware. Un facteur non négligeable serait aussi la consommation des FPGA, qui fonctionnent aux fréquences moins élevées que celles des CPU et GPU et qui pourraient diminuer la consommation des centres de données d'un facteur important.
Il reste, pourtant, le travail à faire pour voir où intervenir dans les applications avec des parties écrites pour les accélérateurs. L'existence d'une API comme celle offerte par OpenCL est très importante quand on veut programmer des dispositifs atypiques comme les FPGA, sans avoir besoin de connaissances en micro-électronique. Les efforts d'Intel d'intégrer plus de programmation FPGA dans leur compilateur propriétaire *i++* reste l'autre option qui pourrait se concrétiser à l'avenir.
Les prochains essais avec le nœud FPGA sur la plate-forme ACP donneront plus d'information sur les possibilités concrètes de transfert d'une partie de calcul sur un accélérateur de ce type. Il sera aussi ouvert à une communauté plus large, avec des conséquences possibles de réplication dans les autres centres.

Chapitre 7

Les GPUs... pour quoi faire ?

Sujet du chapitre 7 –
Les processeurs graphiques (GPU) sont utilisés comme *co-processeur* de calcul pour les processeurs de la machine hôte (CPU). Les GPUs utilisent un modèle de calcul basé sur SIMD (*Single Instruction Multiple Data*). Les capacités des GPUs sont capables d'améliorer dramatiquement les performances des applications de calcul scientifique.

Sommaire

7.1 Introduction 98
7.1.1 Structure d'un GPU 100
7.1.2 Fonctionnement d'un GPU 100
7.2 Omniprésence des GPUs 101
7.3 Utilisation des GPUs 102
7.4 Programmation 103
7.4.1 Langages de programmation bas niveau 103
7.4.2 Langages de programmation haut niveau 103
7.5 Retour sur expériences 103
7.5.1 Projet Sympatick_G 103
7.5.2 Projet CMS-MEM 105
7.6 Recommandations 107

	K80	P100	V100
Fréquence de base	560 MHz	1480 MHz	1530 MHz
Mémoire RAM	24 GB	16 GB	16 GB
Nombre de cœurs	4992	3584	5120
Perf double précision	2.91 TFlops	5.3 TFlops	7.8 TFlops
Perf simple précision	8.74 TFlops	10.6 TFlops	15.7 TFlops
Perf demi précision	-	18.7 TFlops	-
Vitesse mémoire	480 Gbits/s	732 Gbits/s	pic 900 GB/s
Consommation maximale	300 W	300 W	300 W

TABLE 7.1 – Différentes caractéristiques des GPU Tesla.

7.1 Introduction

Les *Graphical Processing Units* (GPU) sont des composants électroniques qui ont été développés à l'origine pour effectuer des rendus graphiques. En effet, cette tâche nécessite une importante puissance de calcul puisque le signal reçu dans chaque pixel doit être calculé le plus rapidement possible afin que l'utilisateur puisse avoir une expérience fluide des animations affichées.

Grâce au GPUs, les CPUs ont été délestés de l'affichage et ont pu être améliorés pour effectuer des tâches plus complexes.

De nos jours les GPUs ont gagné en puissance de calcul et en efficacité et permettent les rendus de scènes en trois dimensions par l'utilisation de matrices 4×4.

Cette puissance de calcul n'est cependant pas toujours nécessaire, mais il est dommage de ne pas l'utiliser. Par exemple des applications de simulations ou d'analyses de données n'ont pas vocation à afficher immédiatement des résultats. Mais il est possible d'utiliser les GPUs pour effectuer des calculs sans faire de rendu sur un écran, c'est le *General-Purpose computing on Graphics Processing Units* ou GPGPU.

La figure 7.1 montre des GPUs NVidia utilisés pour le calcul qui sont présents dans les centres de calculs de l'IN2P3. Le tableau 7.1 montre les différentes caractéristiques de quelques-unes de ces cartes.

FIGURE 7.1 – GPUs utilisés pour le calcul. **En haut à gauche** : Quadro. **En haut à droite** : Tesla Kepler 80. **En bas à gauche** : Tesla Pascal 100. **En bas à droite** : Volta 100.

7.1.1 Structure d'un GPU

Contrairement aux CPUs qui possèdent quelques dizaines de cœurs au maximum, les GPUs en comptent de plusieurs dizaines à plusieurs milliers. Cependant, cette disposition pose un problème de surchauffe qui est résolu en réduisant la fréquence d'horloge des cœurs des GPUs.

Les cœurs des GPUs s'apparentent aux unités arithmétiques et logiques (ALU, *Arithmetic and Logical Units*) des CPUs. En effets, ces cœurs ne font que du calcul et ne peuvent pas faire de la prédiction de branchement comme peut le faire un CPU (voir section 2.2.3.6).

Les composants qui vont chercher et chargent les instructions sont communs à plusieurs cœurs. Une fois assemblé, un groupe de cœurs de calculs affublé d'un chargeur d'instruction (ou plusieurs suivant les architectures) devient un *Streaming Multiprocessor* ou SM, voir figure 7.2.

Chaque multiprocesseur dispose d'une mémoire propre, très rapide qui permet d'accélérer les échanges de données entre des cœurs voisins. Cela permet d'optimiser des calculs où les valeurs des données voisines sont nécessaires, comme les convolutions, l'application de filtre ou les produits de matrices qui sont à la base du *machine learning*.

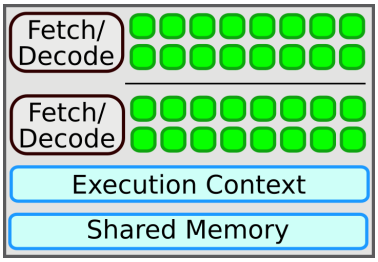

FIGURE 7.2 – Les unités élémentaires de calculs dans un GPU (*Streaming Multiprocessor* SM, en vert) utilisent une mémoire partagée et les threads exécutés sont ordonnés par un même composant.

Ces multiprocesseurs sont à leur tour assemblés pour former la partie calcul du GPU à laquelle on ajoute une mémoire RAM, voir figure 7.3.

7.1.2 Fonctionnement d'un GPU

Dans un GPU les données peuvent être stockées dans des tenseurs à une, deux ou trois dimensions et peuvent être mises en correspondance avec des *threads* qui effectueront des calculs sur ces données. Ces *threads* sont définis dans des blocs de une, deux ou trois dimensions et sont eux-mêmes inclus dans une grille à une deux ou trois dimensions.

Cette configuration modulaire permet de simplifier l'écriture des fonctions de calculs, appelées *kernels*. Si deux *kernels* dans deux *threads* contiguës sont exécutés sur deux données également contiguës, les échanges de données seront plus efficaces.

Une fois que les différents *threads* sont définis, ils sont exécutés par le GPU. Les *threads* sont alors organisés en *warps* qui contiennent un nombre fixe de threads (par exemple 32 pour les K20 ou les K80).

Ensuite, le GPU exécute les deux premières instructions des *threads* de la première *warp*

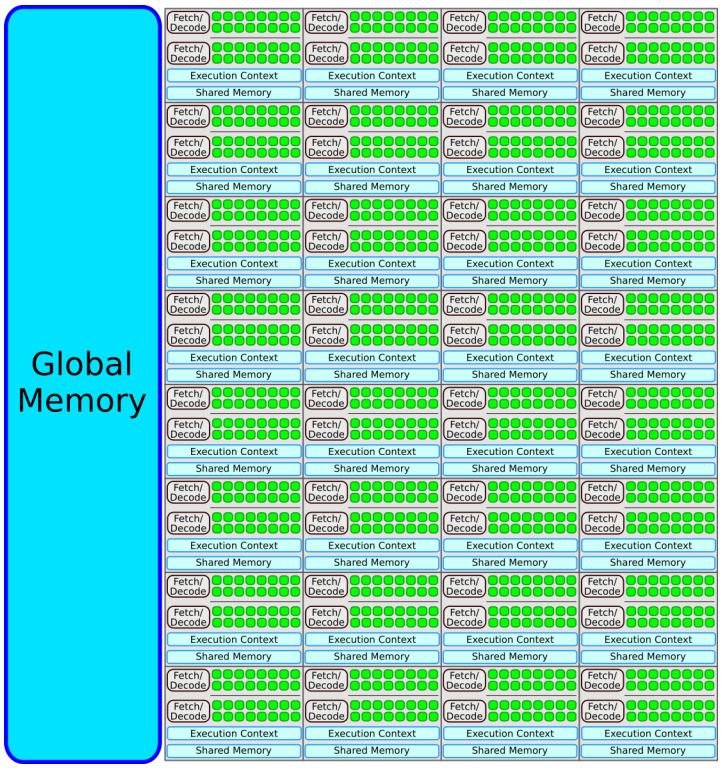

FIGURE 7.3 – L'architecture d'un GPU est composée des unités décrites dans la figure 7.2.

à la dernière, puis les deux instructions suivantes et ainsi de suite.

De ce point de vue, les *threads* d'un GPU sont davantage synchronisés que ceux d'un CPU. Cette synchronisation doit être prise en compte par les développeurs bas niveau afin de ne pas dégrader les performances de calcul.

En effet, si un *kernel* contient une condition, seuls les *threads* ayant une condition positive vont être exécutés immédiatement. Ensuite, les *threads* ayant une condition négative seront exécutés. Cela implique qu'une condition coûtera le temps total d'exécution des résultats positif et négatif. On parle de divergence des branches de calcul dans le GPU.

Ainsi, le GPU sera fortement sous-employé à l'exécution de cette condition, ce qu'il est préférable d'éviter.

7.2 Omniprésence des GPUs

Les GPUs sont de plus en plus présents dans les centres de calculs à travers le monde, car ils offrent une puissance de calcul importante au prix d'une consommation électrique réduite en comparaison de celle des CPUs.

Les superordinateurs les plus puissants dans le *top 500* [73] [1] sont aujourd'hui composés de GPUs. La machine la plus puissante, *Summit* [74], et la seconde plus puissante, *Sierra* [75]

1. Classement des superordinateurs les plus puissants au monde.

sont composées toutes les deux de NVIDIA Volta GV100. La troisième, *Sunway TaihuLight* [76], utilise des processeurs développés spécialement pour l'occasion.

La classification *Green 500* [77] liste les superordinateurs les plus puissants en prenant en compte leur consommation électrique. Les dix meilleures machines de ce classement utilisent des GPUs :

1. *DGX SaturnV Volta* [78]
2. *Summit* [74]
3. *AI Bridging Cloud Infrastructure (ABCI)* [79]
4. *MareNostrum P9 CTE* [80]
5. *TSUBAME3.0* [81]
6. *PANGEA III* [82]
7. *Sierra* [75]
8. *Advanced Computing System (PreE)* [83]
9. *Taiwania 2* [84]
10. *Huawei G5500* [85]

Les GPUs utilisés sont principalement des cartes Tesla Volta (V100) ou Pascal (P100).
On notera également que *Summit* [74] est première au *top 500* et seconde au *Green 500*.

Note 7 –
Même si les machines évoquées sont principalement composées de GPUs pour effectuer les calculs, l'ordonnancement des calculs au niveau de la machine de fait par le biais de CPUs classiques, car les GPUs ne sont pas conçus pour cela (voir section suivante).

7.3 Utilisation des GPUs

Les GPUs sont très performants pour effectuer des calculs simples à grande échelle, car ils sont massivement parallèles. Cependant ces calculs doivent être suffisamment importants pour que le temps de transfert des données sur le GPU soit amorti, sinon l'accélération du programme sera faible voire inexistante.

Les GPUs sont particulièrement efficaces pour traiter des objets identiques en interaction les uns avec les autres. Le problème à N corps en interaction gravitationnelle qui a théoriquement une complexité en N^2 voit sa complexité diminuer en N s'il y a autant de cœurs que de particules sur le GPU.

Il en va de même pour les multiplications de matrices qui ont une complexité en N^3 réduite à N^2 dans les mêmes conditions d'utilisation que précédemment.

Le domaine du *Deep Learning* qui est en effervescence actuellement a connu une révolution technique et conceptuelle lorsque les temps d'entraînement de plusieurs centaines de jours se sont réduits à seulement quelques jours avec l'utilisation de GPUs toujours plus performants. Dès lors, l'exploration de nouvelles architectures fut possible à échelle humaine.

De manière plus générale, le traitement d'images, de vidéos et la modélisation ont été lourdement affectés par les GPUs sans qui les temps de calculs seraient beaucoup trop long.

Les GPUs sont mêmes utilisés dans certains hôpitaux pour ajuster des doses de radiations pour des patients en temps réel ou encore filtrer le signal en sortie d'IRM. La petite taille des GPUs, lorsqu'ils sont bien utilisés, peut remplacer des dizaines d'ordinateurs.

7.4 Programmation

7.4.1 Langages de programmation bas niveau

Il existe plusieurs Langages de programmation bas niveau pour GPU. Le langage CUDA, développé par NVidia, permet de programmer simplement des *kernels* sur tous les GPUs NVidia. Le langage OpenCL est un langage dérivé du C qui permet une portabilité sur des cartes autres que NVidia. Le langage GLSL est un langage qui a été initialement développé pour créer des *shaders* et fut détourné pour être le premier langage GPGPU avant que CUDA et OpenCL ne soient développés.

Une fois que les *kernels* ont été développés, ils doivent être interfacés avec un autre langage, comme C++ par exemple. Certaines bibliothèques comme le ***CUDA-toolkit*** fournissent des *kernels* optimisé pour l'algèbre linéaire qui fonctionnent comme leurs équivalents sur CPU (voir section 3.4.3).

7.4.2 Langages de programmation haut niveau

Depuis quelques années, certains programmes comme ***Tensorflow*** [31], ***Pytorch*** ou ***Chainer*** permettent d'utiliser efficacement les GPUs en développant des programmes en *Python*. Cela permet aux développeurs de se concentrer sur leurs calculs et de moins se soucier de l'architecture cible.

Certains générateurs de code, comme ***Loopy*** [32], permettent aussi de générer du code CUDA.

7.5 Retour sur expériences

7.5.1 Projet Sympatick_G

La Tomographie d'Émission MonoPhotonique (TEMP) est une modalité d'imagerie médicale basée sur l'administration à un patient de molécules radio-labellisées. Le radio-isotope

utilisé en TEMP est un émetteur de photons détectés par une gamma-caméra. En mode tomographique, la gamma-caméra tourne autour du patient et acquiert un ensemble de projections. À partir des projections acquises, on estime la distribution 3D du radio-isotope injecté. Les effets physiques comme l'effet Photoélectrique et la diffusion Compton sont responsables de la dégradation des images reconstruites, car ils sont à l'origine d'une fausse estimation de la concentration du produit radioactif et/ou de sa localisation. Si ces effets physiques ne sont pas corrigés, une dégradation importante du rapport signal-sur-bruit, de la restauration de contraste et de la résolution spatiale est observée dans les images reconstruites ce qui pourrait fausser le diagnostic établi par le médecin. Une méthode pour corriger ces effets physiques consiste à les modéliser et à intégrer le modèle dans un algorithme de reconstruction itérative. Plusieurs méthodes analytiques ont été utilisées pour modéliser ces effets (Photoélectrique, Compton). Bien que les méthodes analytiques soient rapides, leur mise en œuvre peut être complexe quand il s'agit de les implémenter pour des milieux hétérogènes (composition du corps humain) et les géométries originales de détecteurs. Aussi l'utilisation des simulations Monte-Carlo permet la modélisation des effets physiques subies par les photons durant un examen TEMP et donc la correction implicite lors de la reconstruction. Cependant, ces simulations sont très coûteuses en temps de calcul et leur utilisation est à ce jour inappropriée pour des applications cliniques où souvent le taux d'examens d'imagerie journalier est assez élevé surtout si des reconstructions d'images personnalisées sont demandées. Durant ce projet, nous proposons une méthode d'accélération sur carte graphique (GPU) de la modélisation des effets physiques subis par des photons à l'intérieur d'un patient et ce dans un délai acceptable en routine clinique.

Les simulations Monte-Carlo sont considérées comme un outil de référence pour la modélisation du parcours des particules dans la matière. Dans les applications médicales comme la médecine nucléaire et la radiothérapie, une modélisation rigoureuse des processus physiques ayant lieu permet de calculer avec précision le dépôt de dose dans le patient ainsi qu'une reconstruction d'images optimisée notamment dans le cas d'imagerie TEMP (Tomographie par Émission MonoPhotonique), TEP (Tomographie d'Émission de Positons) et TDM (Tomodensitométrie). Cependant, les simulations Monte-Carlo sont assez consommatrices en temps de calcul ce qui les rend incompatibles pour un usage en routine clinique.

Lors de ce projet, nous avons développé un code de simulation hybride (modèle Monte-Carlo analytique) qui s'exécute sur les architectures de cartes graphiques (multi GPU) permettant d'accélérer les calculs Monte-Carlo. Le code développé utilise des tableaux de sections efficaces issus de Geant4 et de Gate. La dose déposée est calculée à partir d'un volume voxélisé et d'une description de sources mono-énergétiques rayonnant de façon isotrope. Nous utilisons des fichiers de directions pré-calculées et réparties uniformément sur la sphère. La méthode de dépôt de dose que nous utilisons est basée sur la méthode d'estimation des longueurs des traces (TLE).

L'optimisation du code a nécessité une réécriture quasi complète de l'algorithme de simulation pour prendre en compte les contraintes d'exécution sur un GPU. En effet, d'une part les branchements sont à éviter parce qu'ils ralentissent le flot d'exécution en introduisant une divergence des threads du GPU. D'autre part, pour exploiter le modèle SIMD (Single Instruction Multiple Data), nous avons regroupé les appels du *kernel* pour plusieurs gerbes de particules similaires. Finalement nous avons mis en œuvre des regroupements de code pour tirer parti du cache mémoire.

Le parallélisme inhérent aux GPUs nous a permis d'accélérer les temps de calcul. Le facteur d'accélération entre un code CPU de référence et notre code GPU est de l'ordre de 200.

7.5.2 Projet CMS-MEM

7.5.2.1 Une méthode d'analyse pour CMS

L'application CMS-MEM est un outil d'analyse basé sur la Méthode des Éléments de Matrice (MEM) [86]. Elle permet de combiner de manière optimale l'information théorique décrivant des processus physiques avec l'information expérimentale décrivant la résolution d'un détecteur. Elle a été développée dans sa version « Htautau » au LLR, et elle est utilisée dans l'expérience CMS pour caractériser les propriétés du boson de Higgs, produit en association avec des quarks top, et se désintégrant en paires de leptons tau.

À l'opposé des méthodes supervisées (réseau de neurones, arbre de décision, machines à vecteur de support), la MEM permet de définir des discriminants pour la classification d'événements, sans apprentissage, en s'appuyant sur la physique pour calculer les probabilités d'observer les états finaux attendus. Cette nouvelle méthode plus précise est, néanmoins, extrêmement consommatrice de temps CPU.

Dans ce contexte, l'équipe CMS du LLR, par le biais du travail de recherche de T. Strebler sur le canal *ttH*, a relevé le défi de développer une application dite « HPC ». Dans une première étape, elle a été exploitée sur des plates-formes multi-nœuds / multi-cœurs, puis dans un second temps sur des architectures multi-nœuds / multi-GPUs. L'implémentation multi-nœuds / multi-cœurs a été utilisée avec succès dans les comptes-rendus publics de la collaboration CMS [87].

7.5.2.2 Contraintes et défis technologiques

Nous avons souhaité dès le début des développements, initialement réalisés sur la plate-forme GridCL[2], relever un certain nombre de défis techniques. Pour traiter de l'analyse du canal *ttH* par MEM dans des délais raisonnables, nous avons conçu l'application autour des standards portables ***MPI*** et ***OpenCL*** pour agréger, respectivement, la puissance de calcul des nœuds et celle des GPUs ou autres accélérateurs de calcul rattachés à chaque nœud. L'autre choix de conception consiste à déployer tous les calculs sur les cartes accélératrices ou GPUs pour minimiser les communications «*host/devices*». Cette stratégie nous a amené à développer une extension OpenCL/CUDA dans le générateur de code *MadGraph* [88], qui engendre l'élément de matrice au cœur de la méthode sous la forme d'un code source. Ceci nous a conduit à manipuler des «*kernels*»[3] allant de 10^4 à $2 \cdot 10^4$ lignes de code, caractéristique peu courante pour ces technologies, permettant ainsi d'éprouver la robustesse des compilateurs pour accélérateurs de calcul.

Plus récemment encore, notre participation à la Cellule de Veille Technologique (CVT)

2. Plate-forme de développement, financée principalement par le LabEx P2IO
3. Terme utilisé pour désigner les codes téléchargés sur l'accélérateur de calcul, une fois compilés

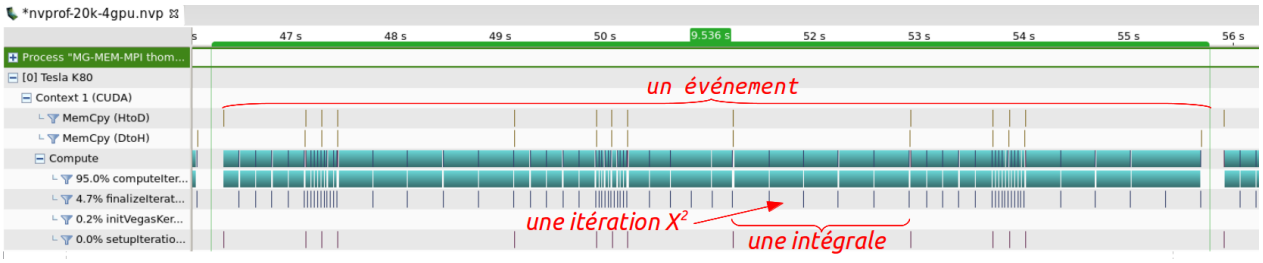

FIGURE 7.4 – Détails au niveau de l'événement des échanges entre l'hôte et un GPU. Pour chaque événement, plusieurs scenarii sous forme d'intégrales sont évalués. Chaque intégrale est calculée plusieurs fois (5 fois) pour effectuer un test de khi-deux.

GENCI [4], nous a mis face à une difficulté supplémentaire. Leur plate-forme pré-exascale [5] combine des processeurs IBM OpenPOWER à instructions non x86, et des cartes de génération Pascal, pour lesquelles NVIDIA ne fournit pas encore d'implémentation d'OpenCL. Soucieux de profiter des évolutions de CUDA et des outils NVIDIA (non accessibles depuis OpenCL), nous avons construit une passerelle OpenCL 1.1/CUDA, pérennisant ainsi nos développements OpenCL face aux aléas. Cette passerelle a été largement validée et exploitée sur GridCL, sur les GPUs Pascal P100 de l'IDRIS et sur la plate-forme GPU du CC-IN2P3.

7.5.2.3 Production sur la plate-forme GPUs du CC-IN2P3

Parmi les possibilités de déploiement de CMS-MEM en mode production, l'acquisition d'une plate-forme GPUs par le CC-IN2P3 a été une aubaine pour effectuer les analyses requises. Cette machine dispose d'une puissance de calcul importante, répartie sur 10 nœuds équipés chacun de 4 accélérateurs NVIDIA K80. Le portage de CMS-MEM sur cette plate-forme a été effectué sans difficulté et le mode de soumission a été adapté pour répondre aux contraintes du système de queues.

Comme illustré dans la figure 7.4, le taux d'occupation des GPUs est excellent : les kernels calculent la majorité du temps d'exécution et les communications host/devices sont négligeables. chronologie des communications (traits noirs) et kernels actifs (en bleu) sur un nœud (4 GPUs)

Pour évaluer le gain global en performance de l'application, nous avons pris comme référence l'analyse de 3000 événements. Le temps CPU pour traiter de cette configuration, s'évalue approximativement à 15 heures sur une plate-forme type MPI du CC-IN2P3 comportant 100 cœurs (100 processus MPI). Après quelques difficultés de mise au point des classes (le point délicat étant de coupler des classes « parallèles » avec la réservation des ressources GPUs), nous avons pu lancer nos premières exécutions de production sur 2, 4, puis 6 nœuds. Pour cette dernière configuration, 6 nœuds totalisant 24 GPUs, l'application a requis 1767 secondes. Le facteur d'accélération substantiel obtenu, approximativement 30,

4. Cellule de Veille Technologique (CVT) GENCI est en charge de préparer les communautés scientifiques françaises à l'émergence des prochaines générations de supercalculateurs

5. Les architectures dites exascale désignent les futurs supercalculateurs qui disposeront d'une puissance de calcul supérieure à 10^{18} opérations en arithmétique flottante par seconde (ExaFlops)

n'est pas seulement imputable aux GPUs. La réécriture du calcul des éléments de matrice C++, en kernels C/OpenCL entre dans une part importante de ce gain. Ainsi la puissance de calcul de la configuration avec 6 nœuds est équivalente à une plate-forme MPI comportant 3000 cœurs pour notre application CMS-MEM. Outre le gain en puissance de calcul, l'exploitation des accélérateurs de calcul ou GPU permet de réduire significativement la facture de la consommation électrique des centres de calcul, dans notre cas, cette consommation est réduite d'un facteur 5 [6].

Cette version, première du genre, est en cours d'exploitation sur les nœuds GPUs du CC (prochainement 8 nœuds), et alimente les résultats d'analyse du canal ttH pour la thèse de T.Strebler, libérant ainsi la grille d'une charge significative de travail.

Dans les années à venir, son évolution profitera des axes de réflexion du groupe de travail RI3 « CodeursIntensifs » [89] et des projets IN2P3 en cours de constitution autour des thèmes des conteneurs, de la génération de code, de la précision et de la reproductibilité numérique dans un contexte de calcul parallèle.

7.5.2.4 Remerciements

Nous adressons nos sincères remerciements au LabEx P2IO pour avoir financé (avec une contribution de l'équipe CMS et du LLR) la plate-forme GridCL, brique indispensable aux développements de CMS-MEM. Nous remercions le Centre de Calcul de l'IN2P3 pour l'élaboration du contexte de production de CMS-MEM, ainsi que la CVT GENCI pour les échanges lors des groupes de travail autour d'experts HPC.

7.6 Recommandations

Les GPUs offrent une puissance de calcul brute à moindre coût et moindre consommation face aux CPU, tout en occupant moins d'espace.

La large utilisation des GPUs dans différents domaines (*Deep Learning*, Animation, Calcul HPC, *etc.*) garantit leur pérennité et l'existence d'une communauté active.

Les structures de données doivent être adaptées pour être efficaces sur GPU. Ces adaptations ont aussi des effets collatéraux bénéfiques pour la partie CPU parce qu'elles améliorent la localité des données.

L'effort nécessaire à l'adaptation des codes existants ou l'écriture de nouveaux codes requiert des formations aux architecture et à la programmation des processeurs graphiques.

Tout ceci demande des compétences spécifiques d'Ingénieur en Calcul Scientifique, obtenues soit par le recrutement de nouveaux personnels soit par l'évolution et la formation de personnels existants.

6. Comparaison effectuée en prenant uniquement en compte la consommation électrique des processeurs (E5-2698 v3 16 cœurs) de 135 W et la consommation par GPU de 150 W

L'essentiel du chapitre 7 –

Les GPUs étaient à l'origine conçus pour effectuer des calculs de rendus graphiques et peuvent maintenant être utilisés pour des calculs massivement parallèles à un coût réduit en comparaison des CPUs classiques.
C'est pour cette raison que les superordinateurs utilisent aujourd'hui massivement des GPUs afin de combiner puissance de calcul et consommation électrique.
Le domaine du *Deep Learning* a notamment connu un essor important grâce à la démocratisation de cette technologie et du HPC, sans qui, les temps d'entraînement des réseaux de neurones n'auraient pas été possibles à échelle humaine.
Malgré cela, tous les calculs ne se prêtent pas à l'utilisation de GPUs même si certaines astuces permettent de traiter des problèmes considérés comme scalaires.
Les GPUs seront de plus en plus présents dans le domaine du calcul intensif et devront être utilisés le plus efficacement possible.

Chapitre 8

La problématique de la précision

Sujet du chapitre 8 –

La précision des calculs est un sujet délicat qui est trop souvent simplifié par le rejet des types flottants simple précision et l'utilisation abusive de la double précision. Malgré certaines idées reçues, il peut être pertinent d'utiliser la précision simple dans certains cas afin d'accélérer les calculs car tous les calculs n'ont pas besoin du même niveau de précision. À l'inverse, l'utilisation de la double précision, ne résout pas tous les problèmes, et la pertinence des résultats doit systématiquement être vérifiée.

Sommaire

8.1 Introduction 110
8.2 Erreur de calcul 110
8.3 Sécurité numérique et évaluation de la précision d'un calcul . . . 111
8.4 Reproductibilité des résultats 112

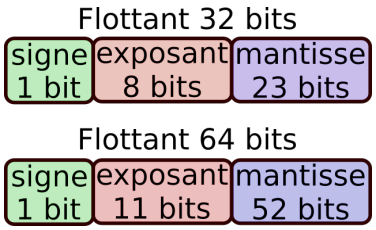

FIGURE 8.1 – Norme IEEE 754 qui décrit les flottant 32 et 64 bits.

8.1 Introduction

La précision, et sa banalisation (*« commoditization »*) notamment en raison de l'effondrement de son coût, et particulièrement de sa reproductibilité, est à l'origine de nos sociétés industrielles depuis le milieu du XVIII^{e} siècle. L'évolution de la précision numérique est allée encore plus vite et encore plus loin que la précision mécanique.

Après un développement assez anarchique des formats et des algorithmes de calcul en virgule flottante à l'origine du secteur informatique, une norme a émergé en 1985 qui a standardisé les formats : IEEE 754 [90–92] (voir figure 8.1) et ses évolutions [93, 94].

Outre un standard de format, cette norme fournit des garanties de précision pour les diverses opérations et fonctions, ainsi qu'une interface avec la FPU (*Floating Point Unit*) sous forme de contrôle du mode d'arrondi, de flag de statut, de représentation des nombres exceptionnels ou dénormaux et d'exceptions masquables en cas de débordement, de division par zéro, d'indétermination voire de calcul inexact. Les puristes reprochent l'impact de cette approche sur les performances (exception et dénormaux) et sur l'efficacité de la représentation.

Ses défauts ne doivent pas nous faire oublier ses nombreuses avancées et la situation antérieure. Le parti pris de cette norme a été de donner une forme de contrôle à l'utilisateur sur cette boite noire qu'est le calcul numérique, et des moyens pour surveiller la précision tout en gardant la performance. Avant même cette standardisation, il faut porter au crédit du langage `C` d'avoir permis une amélioration de la garantie des résultats numériques en procédant à tous les calculs intermédiaires en double précision.

8.2 Erreur de calcul

L'erreur peut se manifester durant deux grandes étapes théoriques de chaque calcul, qui sont pratiquement souvent fusionnées : d'une part l'évaluation en précision illimitée du calcul à partir de données essentiellement arrondies, d'autre part l'arrondi de ce résultat. Aussi dans le cas idéal, en partant de données tombant juste dans la représentation choisie, l'arrondi du résultat n'impliquera qu'une incertitude d'une demie `ulp`[1] ce qui est l'« arrondi exact ». La multiplication, la division, la soustraction de nombres qui diffèrent de moins d'un facteur deux, la racine carrée sont ainsi garantis par la norme. Par contre, les fonctions transcendantes, même les plus élémentaires de la trigonométrie, ou l'exponentielle et le logarithme,

1. *« Unit in the Last Place »*, soit la valeur du ϵ machine

sont affectées par une incertitude plus grande : la question qui se pose pour l'arrondi en base deux est de savoir où s'arrête de longue séquence de chiffres 1 : c'est le dilemme du fabricant de tables (de logarithmes, de sinus...). Le contrôle de l'arrondi est donc un sujet en soi.

Les apports de cette norme sont toutefois restés cantonnés à une petite communauté : bien que la large implantation d'Intel, promoteur de cette norme, ait amené ces fonctionnalités sur tous les bureaux depuis le coprocesseur `8087`, rares ont été les systèmes et les langages à proposer la double précision étendue ou les flottants en base dix (pourtant cruciaux pour les calculs comptables), sans parler de propager l'interface du FPU dans les bibliothèques de ces langages...

8.3 Sécurité numérique et évaluation de la précision d'un calcul

Force est de reconnaître qu'au sein de la population des développeurs, même dans les domaines scientifiques et techniques, la fraction consciente des problèmes de calcul numérique (une des branches des mathématiques), et en particulier des problèmes de mise en œuvre informatique du calcul numérique est notoirement peu étoffée.

Conscient de ces deux aspects, les experts du calcul numérique préconisent une attitude d'autodéfense au béotien [95]. Le cas échéant, en agitant (certes avec pédagogie) le spectre de suppressions catastrophiques [96]. Cette approche est d'ailleurs commune à tout le génie logiciel : un ensemble de bonnes pratiques est là pour garantir la protection contre les échecs retentissants, même avec une compréhension partielle de leur justification ultime.

En pratique, la base de la sécurité numérique est de pouvoir faire le même calcul en simple précision et en double précision et d'obtenir des résultats compatibles (ou alors en double précision et en double précision étendue). Le cas de la quadruple précision est à part, car il n'en existe pas d'implémentation matérielle, ce qui réduit dramatiquement ses performances (au moins d'un ordre de grandeur) et restreint par suite son usage.

Une première recommandation est donc de produire du code numérique aussi générique que possible, en notant bien que cette généricité ne saurait se résoudre à celle des types des variables mises en jeu dans le calcul.

Un premier obstacle vient généralement des constantes littérales : les constantes double précision étendue du `C` (`long double`) doivent être suffixée, de même qu'en `Fortran` chaque précision est spécifiée par un suffixe : `D0`, `Q0` ou `_8`, `_10`, `_16` en `Fortran` moderne. Sans quoi le compilateur procédera à des promotions de précision selon des règles obscures. Ce dernier point parle en faveur d'un recours aux constantes nommées : les `#define` du `C/C++` n'échappent pas à cette obligation de suffixation, mais les `const` du `C++` ou les `parameter` du `Fortran` permettent d'associer clairement un nom à une valeur et à sa précision.

Bien que la majorité des fonctions intrinsèques existent sous forme générique dans les bibliothèques associées au compilateur, la difficulté apparaît dès qu'on doit produire ses propres fonctions : au-delà de la taille des variables et de la précision des coefficients employés, c'est le nombre de ces coefficients qui doit changer (notamment s'ils correspondent à un développement). Si ce développement n'est pas en série entière, mais un approximant de Padé,

les valeurs changeront du tout au tout d'une précision à l'autre. Enfin le nombre d'itération requises pour atteindre une précision donnée est également sujet à fluctuation.

Pour aller au-delà de ces approches d'autodéfense et de validation par stabilité, on peut recourir à des environnements plus récents d'évaluation de la précision numérique. En l'occurrence, *CADNA*[2], *Verificarlo*[3] et *verrou*[4] qui peut être utilisé avec *valgrind* [3]

Toutes les approches jusqu'à présent s'appliquent déjà à des programmes séquentiels. La validation de la précision de calculs avec des programmes parallélisés passe au début par une démarche analogue : la comparaison du résultat en séquentiel et en parallèle peut déjà être surprenante. En effet, la première différence entre la version séquentielle et la version parallèle d'un calcul vient de l'ordre des séquences d'opérations : par exemple, les réductions d'un Monte Carlo vont passer d'une longue somme des résultats de chaque itération dans la version séquentielle à une somme des sommes partielles issues de la sous-séquence de chaque thread. Mathématiquement, l'associativité et la commutativité de l'addition des nombres réels nous garantissent l'identité des résultats. Mais les nombres à virgule flottante ne vérifient pas ces deux propriétés élémentaires avec la même systématique. Dans ce cas, on peut améliorer cette approche en utilisant des accumulateurs de plus haute précision, ou l'algorithme de sommation de W. Kahan, mais cela complique l'expression du code et devient difficilement applicable à tous les calculs parallélisés.

8.4 Reproductibilité des résultats

Bien que cet aspect ne soit que très peu présent en utilisant des types entiers, garantir la reproductibilité des résultats est une tâche ardue en arithmétique flottante. En effet, chaque calcul numérique a, par définition, une précision limitée par le nombre de chiffres significatifs que les unités de calcul de l'unité arithmétique et logique peuvent fournir dans le CPU. Il y a donc une limite physique à la précision d'un calcul et celle-ci est respectée au mieux par les développeurs qui fournissent les fonctions mathématiques standards (`sin`, `cos`, `tan`, `exp`, `log`, `sqrt`, *etc.*).

À cela s'ajoute le fait que l'ordre dans lequel les calculs sont effectués peut influer sur le résultat final. D'ordinaire un calcul vectorisé produira des résultats plus précis que sa version séquentielle, alors qu'il est courant de demander que la version vectorisée produise des résultats identiques à la version séquentielle qui est généralement la référence, puisque c'est la première à avoir donnée un résultat.

Comme nous l'avons vu dans le chapitre 4, les calculs sont de plus en plus parallélisés ce qui ajoute une part non négligeable d'imprévus dans le déroulement des calculs effectués. Même si chaque calcul effectué sur un *thread* donne un résultat avec une certaine précision, la méthode d'agrégation de ceux-ci peut porter préjudice au résultat final et plus particulièrement l'ordre dans lequel il seront traités lorsqu'il est nécessaire de les réduire avec une addition par exemple.

L'utilisation de générateurs de nombres pseudo-aléatoire pose aussi problème, car il doit être possible, sur demande, de rejouer une séquence de nombres afin de tester un compor-

2. http://cadna.lip6.fr/
3. https://github.com/verificarlo/verificarlo
4. http://edf-hpc.github.io/verrou/vr-manual.html

tement imprévu ou confirmé un résultat. Lorsque ce problème se pose sur un programme parallèle, les *threads* doivent tous avoir une séquence de nombres pseudo-aléatoire propre, mais il est également nécessaire de garantir que toutes ces séquences seront effectivement différentes, sinon le résultat global ne sera pas le fruit d'un comportement réellement pseudo-aléatoire. Voir, dans certains cas, aucune amélioration du temps d'échéance ne sera obtenue ainsi qu'aucune amélioration du résultat final (si chaque *threads* doit explorer une partie d'un espace relativement grand pour trouver un minimum, ou calculer une intégrale avec une méthode stochastique).

L'essentiel du chapitre 8 –

La préoccupation de la précision des calculs est un sujet généralement délaissé par les développeurs d'analyses ou de simulations sous prétexte de l'utilisation de la double précision qui vaut gage de résultats cohérents. Ce qui n'est pas le cas.
Malgré cette réputation, le domaine de la précision permet d'accélérer considérablement certains calculs lorsqu'ils n'ont pas besoin d'une précision importante et à l'inverse d'utiliser des types adéquats lorsque la qualité de la précision est cruciale.
C'est un très bon candidat qui permet de compléter de manière pertinente les approches de l'optimisation de code ou de la parallélisation des calculs en apportant un éclairage différent sur les problématiques abordées.

Chapitre 9

Recommandation de production de logiciels open-source

Sujet du chapitre 9 –

Le développement de logiciels (simulation, analyse, modèle) est une tâche difficile mais qui peut néanmoins être grandement facilitée et simplifiée par l'utilisation de règles simples. Cela permet un développement plus efficace, et également, de partager efficacement les programmes ou bibliothèques ainsi développés à des collègues ou toute personne qui pourrait en avoir besoin.

Sommaire

9.1 Introduction . . . 116
9.2 Modularité . . . 116
9.3 Réutilisabilité . . . 116
9.4 Développement . . . 117
9.4.1 Convention de nommage . . . 117
9.4.2 Écrire des fonctions courtes . . . 117
9.4.3 Outils de développement . . . 117
9.4.4 Cycles courts . . . 118
9.4.5 Tests unitaires . . . 118
9.4.6 Tests de qualité d'exécution/fuites mémoire . . . 118
9.5 Chaîne de compilation . . . 119
9.5.1 Choix du compilateur . . . 119
9.6 Documentation . . . 120
9.7 Gestion des versions et sauvegarde des projets . . . 121
9.7.1 Utilisation de ces outils . . . 122

9.1 Introduction

Les programmes scientifiques d'analyse ou de simulations devront répondre à des défis toujours plus ambitieux. L'augmentation de la complexité des défis à relever ne doit cependant pas contribuer à complexifier ces logiciels.

En effet, lorsque la complexité d'un programme augmente, sans que sa conception ne l'ait pris en compte, elle devient rapidement hors de contrôle ce qui implique que toutes modifications ou améliorations dudit programme nécessiteront un temps de développement toujours croissant.

Le développement de fonctionnalités d'un tel programme atteindra une asymptote qui sera indépendante du nombre de contributeurs et contraindra au développement d'une nouvelle version de ce programme qui devra, lui, prendre en compte l'ajout de fonctionnalités de la manière la plus simple possible.

Lorsque ce type de limitation arrive au niveau d'un important programme dont dépendent de nombreux scientifiques, le problème devient on ne peut plus préoccupant.

9.2 Modularité

Il est important de concevoir un programme complexe comme un ensemble de fonctionnalités cohérentes qui répondent chacune à une problématique.

De cette manière, les modifications et les améliorations de ces fonctionnalités sont grandement facilités. Le développement du programme sera donc plus souple et efficace qu'un programme monolithique.

D'autre part, la compréhension et l'utilisation de ces parties simplifiées seront également améliorées et la communauté rassemblée autour de l'utilisation et du développement de ce programme grandira plus rapidement.

9.3 Réutilisabilité

Tous les programmes et les bibliothèques développés doivent l'être dans l'optique d'être utilisées par des personnes tierces, d'autant que le développeur d'un programme deviendra complètement étranger à celui-ci au bout que quelques jours voir quelques semaines puisqu'il aura lui-même acquis de l'expérience.

Dans l'ensemble, les programmes et les bibliothèques développés comme étant des tests sont tous devenus utiles pour une communauté ou un groupe de chercheurs après plusieurs mois. Cela est parfaitement normal, puisque l'on améliorera ces tests jusqu'à ce qu'ils donnent des résultats satisfaisants.

De ce fait, il est préférable de traiter un « petit projet de tests » comme un projet à part entière en le concevant de la meilleure manière possible. Cela est facilité par le fait que ces projets tests commencent généralement par une taille relativement réduite.

Comme ceci, le développement de petits projets devient un entraînement afin d'en développer de plus importants, et les réflexes de conception efficace seront acquis de sorte que la conception des projets suivants sera bien plus rapide pour un résultat plus efficace.

9.4 Développement

9.4.1 Convention de nommage

Avoir une convention de nommage permet, d'un simple coup d'œil, d'identifier la nature d'un mot clé utilisé. Par exemple, si les variables commencent par une minuscule, les classes par une majuscule et que les macros sont complètement écrites en majuscules il sera très facile de les distinguer. Ce qui réduit les ambiguïtés, donc les risques de bogues et le temps de développement.

Nommer les variables, fonctions, classes, modules, *etc.*, d'une manière adéquate et standard permet également de faciliter une future utilisation par des collègues qui ne seront pas *a priori* familiers avec.

9.4.2 Écrire des fonctions courtes

Le développement de fonctions courtes (plus courtes que la hauteur de l'écran) offre, malgré sa simplicité apparente, de nombreux avantages.

Tout d'abord, l'entièreté de la fonction peut être vu en une fois. Cela implique qu'il sera beaucoup plus simple de la relire, et donc de trouver des bogues potentiels.

Si cette fonction effectue des calculs, elle sera mieux optimisée par le compilateur, car sa complexité sera réduite.

Si un programme lent utilise des fonctions courtes, le profilage des performances de son exécution sera d'autant plus précis et la cause des points chauds sera d'autant plus facile à déterminer. Ces fonctions courtes seront d'autant plus simples à remplacer par des fonctions optimisées (ou de bibliothèques tierces) afin d'obtenir de meilleures performances.

La documentation globale générée (voir section 9.6) associée à un programme sera plus conséquente et précise. À l'inverse, la documentation relative à chaque fonction sera bien plus simple à écrire.

9.4.3 Outils de développement

Il existe de nombreux outils pour facilité le développement de logiciels.

Cette section se concentrera sur l'écriture des programmes (la sauvegarde et le versionnement de ces derniers sera traitée dans la section 9.7).

Bien qu'il suffise d'un simple éditeur de texte pour écrire un programme, il est préférable d'utiliser des environnements de développement (IDE, *Integrated Development Environment*) [1].

1. Les IDEs doivent tout de même être évités par les développeurs débutants afin qu'ils acquièrent les

Ils fournissent des fonctionnalités comme la génération de code, la navigation dans un code existant (cela est particulièrement utile pour comprendre la structure d'un code que l'on ne connaît pas), et ils peuvent compléter automatiquement et dynamiquement le code écrit par un développeur en temps réel, ce qui constitue un gain de temps incomparable lors d'une phase de développement.

De nombreux environnements de développement sont disponibles comme *KDevelop* [97], ou certains éditeurs de texte avancés comme *Charm* [98] ou *SublimeText* [99].

9.4.4 Cycles courts

Privilégier les cycles de développement de quelques jours ou quelques semaines aux cycles plus longs augmente la flexibilité du développement et permet de gérer plus efficacement les imprévus.

Les imprévus surviennent toujours lors d'un développement (fonctionnalités manquantes, paramètres superflus, *etc.*). Dès lors, il est plus efficace de les prendre en compte dans la conception et le développement plutôt que de considérer qu'il n'y aura pas d'imprévu dans un cycle de développement.

9.4.5 Tests unitaires

Les tests unitaires sont des programmes qui vérifient qu'une fonction se comporte bien comme prévu avec des jeux de paramètres définis. Ainsi, une nouvelle fonctionnalité est ajoutée dans un programme, accompagnée de son test unitaire qui garantira son bon fonctionnement tout au long de la vie du programme. Si une mise à jour casse le bon fonctionnement de cette fonctionnalité, son test associé renverra une erreur ce qui permettra de repérer immédiatement le problème. Lors d'un cycle de développement standard, les tests unitaires sont lancés au moins une fois pas jour pour vérifier que le programme en question reste cohérent.

Il est aussi possible de tester automatiquement la qualité de la précision d'un programme (voir chapitre 8).

9.4.6 Tests de qualité d'exécution/fuites mémoire

Un programme peut se comporter correctement au premier abord mais contenir des erreurs qui ne seront pas remontées par le système d'exploitation.

Généralement, un dépassement mémoire [2] provoque une erreur de type *segmentation fault*. Ceci est causé par le fait que le programme qui a produit ce dépassement a tenté une lecture ou une écriture dans un endroit de la mémoire auquel il n'avait pas les droits d'accès. D'où le célèbre *segmentation fault* de la part du système d'exploitation.

---

automatismes nécessaires à un développement efficace. Mais une fois que ces réflexes sont acquis, les IDEs gagnent énormément de temps.

2. Lecture au-delà de la mémoire allouée. Lire $N+1$ éléments dans un tableau de N éléments, par exemple.

Cependant, si le dépassement mémoire touche une partie de la mémoire réservée pour le même programme (une autre variable ou un autre tableau) le système d'exploitation ne relèvera pas d'erreur puisqu'un programme peut faire ce qu'il veut de sa mémoire.

Dans l'absolu, ce comportement n'est tout de même pas souhaité par le développeur, mais comme aucune erreur n'est soulevée par le système d'exploitation, le développeur peut considérer que son programme est correct.

Un autre problème fréquent, est l'omission de la désallocation de la mémoire allouée dynamiquement (avec *malloc*, *new* ou *mmap*). Ce cas peut être décelé si la consommation mémoire est bien trop importante en comparaison de la taille de la mémoire RAM, mais sera extrêmement difficile à mettre en évidence si la consommation est faible.

Le programme *valgrind* [3] permet d'analyser le comportement d'un programme pendant son exécution et ainsi déceler les fuites mémoires. Il peut même déterminer à quelle ligne de code le tableau qui n'est pas désalloué est effectivement alloué, (voir cours [4]). Il détecte également les écrasements involontaires de variables suite à des dépassements mémoire et informe quelle ligne de code produit cet écrasement illégal (voir cours [4]).

Il est très efficace de compléter les tests unitaires en les lançant également avec *valgrind* afin de garantir que l'exécution ne contient pas d'erreurs. Il faut néanmoins prendre en compte le fait que l'exécution avec *valgrind* sera légèrement plus lente.

Le programme *valgrind* peut également vérifier des programmes parallèles sur une même machine.

9.5 Chaîne de compilation

Le développement de programmes complexes implique un découpage des fonctionnalités en fichiers, bibliothèques, sous programmes, *etc.*, pour améliorer la lisibilité de celui-ci.

Le programme *CMake* [100] permet une compilation automatique de fichiers de programmes écrit principalement en C/C++.

Il teste la présence des dépendances sur le système avant de commencer la compilation (ce qui gagne énormément de temps).

Certains programmes installent directement leurs dépendances via *CMake* (comme PLIBS_9 [28]).

Il peut également compiler et installer des *wrappers Python* et des fonctionnalités peuvent être ajouté dans un langage qui lui est propre.

Il simplifie également l'utilisation des tests unitaires et facilite ainsi leur insertion dans le processus de développement.

9.5.1 Choix du compilateur

Il existe de nombreux compilateurs comme *GCC*/*G++* [23] ou *CLang*/*CLang++* [101] qui ont tous de nombreuses versions.

Classiquement, une version est choisie et fixée jusqu'à la fin du projet. Cela permet théoriquement de garantir que les résultats seront cohérents ou du moins compatibles entre eux.

Une méthode diamétralement opposée permet d'obtenir une bien meilleure confiance dans la qualité des résultats et du programme développé.

Elle consiste à utiliser plusieurs versions de plusieurs compilateurs différents et à vérifier que les résultats sont compatibles entre eux. Par exemple, *GCC* 7.2, *GCC* 8.3, *GCC* 9.3 et *CLang* 9.0, *CLang* 10.0.

Utiliser différentes versions d'un même compilateur permet de s'assurer que les différentes passes d'optimisations donnent les mêmes résultats (si ce n'est pas le cas, il y a très probablement un problème dans le programme lui-même et non dans les compilateurs qui sont des programmes extrêmement suivis et surveillés par la communauté des développeurs).

Utiliser plusieurs compilateurs différents permet de s'assurer qu'aucune maladresse n'a échappée au premier compilateur (comme l'omission d'une paire d'accolades ou l'utilisation du mauvais indice pour accéder aux valeurs d'un tableau).

Quel que soit le compilateur utilisé, il est indispensable d'activer tous les avertissements (*warnings*) avec l'option -Wall car ils portent un regard critique sur le travail du développeur, et de transformer les avertissements en erreurs avec l'option -Werror afin d'éviter qu'un flot d'avertissements masque la lecture d'un avertissement critique pour la pertinence du programme.

9.6 Documentation

La documentation est une partie critique dans de trop nombreux projets, alors qu'une documentation efficace est la seule garantie pour que les utilisateurs apprécient et recommandent l'utilisation de tel ou tel programme.

Dans l'idéal, une documentation est en fait constituée d'au moins deux documentations :

— une documentation pour les utilisateurs
— une documentation pour les développeurs

La documentation des utilisateurs est la plus complexe à produire, car elle requiert une vision d'assez haut niveau des fonctionnalités fournies, et est bien plus efficace et appréciée si elle est composée de cours et d'exemples (qui peuvent être inspirés de certains tests unitaires). Il est tout de même préférable d'écrire ces cours ou ces exemples une fois que l'on est sûr que les fonctionnalités présentées ne seront pas supprimées.

La documentation des développeurs est la plus simple à réaliser et peut être utilisée comme un outil de développement à part entière plutôt que le dernier livrable d'un produit fini.

En effet, l'écriture continue de cette documentation à l'ajout de chaque fonction permet d'une part d'expliquer clairement l'objectif de cette fonction (si l'objectif n'est pas clair, il

faut revoir la conception ou découper la fonction en plusieurs fonctions aux objectifs simples et clairs, cela permet d'éprouver la conception du programme avant même de commencer à écrire du code, ce qui est un gain de temps en soi).

Dans le cas où l'on a pas le temps d'écrire la fonction, avoir sa documentation permet de se rappeler plus facilement de ces objectifs et cela évite de changer d'avis en plein développement (ce qui provoque des bogues qui peuvent être extrêmement difficiles à trouver et à corrigés).

Enfin cette documentation peut être écrite en quelques mots avec la description des paramètres ainsi que de la valeur retournée. Et bien que cela ne prenne que quelques secondes pendant son développement (car les idées sont fraîches) cela prendra de plusieurs minutes à plusieurs jours si cette documentation est écrite des mois ou des années plus tard).

Il est indispensable d'écrire cette documentation pendant la phase de développement et non après, car cela va gagner un temps conséquent en fin de projet.

La documentation des développeurs peut être mise en forme par des programmes comme *Doxygen* [102], *Sphynx* [103] grâce à l'utilisation de commentaires spéciaux. Cela simplifie d'autant son écriture.

9.7 Gestion des versions et sauvegarde des projets

Le développement d'un projet nécessite une sauvegarde de celui-ci dans un état antérieur mais le plus à jour possible afin d'éviter la perte définitive du travail déjà effectué.

Dans ce domaine, des programmes de gestions de versions ont été mis au point. Le plus puissant d'en eux est le programme *Git* [104]. Il permet de sauvegarder un projet complet sur autant de supports désirés (disques durs, clés USB, serveurs, *etc.*), de sauvegarder l'historique de développement et également de naviguer dans cet historique afin de retrouver une version antérieure si la version la plus récente ne convient pas.

La notion de *branches* permet de manipuler autant d'historiques différents que l'on souhaite, chaque historique et chaque changement peuvent être fusionnés automatiquement et si ce n'est pas le cas, une aide à la fusion permet de montrer les différentes modification entre deux versions conflictuelles.

La sauvegarde de projets de l'IN2P3 est disponible au CCALI [105] *via* la plate-forme *gitlab*, et la sauvegarde de projets en général est disponible gratuitement sur la plate-forme *Github* [106].

Ces outils permettent également de dupliquer un projet existant et de stocker sa copie sur l'espace de l'utilisateur qui l'a demandé. Cela permet d'avoir une sauvegarde complète d'un projet mais en effectuant des changements comme sur un projet personnel (les restrictions de modifications sont plus strictes lorsque l'utilisateur n'est pas propriétaire du projet qu'il développe).

Une fois que les modifications ont été effectuées, l'utilisateur peut demander une fusion (*pull request*) afin que ces modifications soient appliquées dans la version principale. Il est aussi possible de le faire au niveau des branches, on parle alors de (*merge request*).

Ces outils sont indispensables pour développer efficacement.

9.7.1 Utilisation de ces outils

Ces outils offrent de nombreuses fonctionnalités qui doivent cependant être utilisée judicieusement.

Une utilisation efficace consiste à définir quelques personnes comme ayant le droit de modifier la branche principale (*master*) de l'historique des modifications sur *gitlab* [105] ou *Github* [106]. Ces personnes ont donc la responsabilité de maintenir la branche *master* à jour et fonctionnelle en effectuant les fusions d'autres branches développées par eux-mêmes et les autres développeurs du projet.

Lorsqu'une branche est fusionnée sur *master* (ou une autre branche) ses modifications doivent être fusionnées (*squash commits*) pour que la modification finale soit la plus claire possible, elle doit aussi être supprimée afin d'éviter une nouvelle fusion alors que cette branche ne sera certainement plus à jour, et permet aussi de simplifier la recherche de bogues via la fonction *git bisec*.

Des tests unitaires (voir section 9.4.5) doivent être ajoutés à chaque nouvelle fonctionnalité pour vérifier leur fonctionnement au cours du développement. Il est préférable de préparer des tests qui vérifient si le programme *valgrind* ne détecte aucune manipulation illégale dans l'exécution des tests unitaires.

L'ensemble des tests doit être lancé idéalement à chaque nouvelle modification (*commit*) et au moins une fois par jour.

Il est préférable de générer la documentation (voir section 9.6) après avoir lancé les tests unitaires. Cette documentation peut être mise en ligne automatiquement avec le système de *pages* de *gitlab* [105] et *Github* [106]. De cette façon, une documentation à jour est toujours disponible et cohérente avec la version principale.

Pour des raisons de clarté, une branche qui résout un problème (ou *issue*) doit porter le même nom que ce dernier.

L'essentiel du chapitre 9 –

Les techniques modernes de développement de logiciel permettent, à l'aide d'outils adéquats, d'améliorer la qualité du développement logiciel, la visibilité les programmes, des bibliothèques, ainsi que leurs développeurs, favorisent les échanges et permettent de gagner énormément de temps.
Ces pratiques, pas ou peu connues des physiciens, doivent être partagées afin que leurs développements deviennent des atouts et non des poids.

Chapitre 10

Conclusion

Sommaire

10.1 Enseignements à propos de la quête de performance 124
10.1.1 Optimisation 124
10.1.2 Vectorisation 124
10.1.3 Parallélisation 125
10.1.4 Langages de programmation 125
10.1.5 Calcul sur FPGA 126
10.1.6 Calcul sur GPU 127
10.1.7 Problématique de la précision 127
10.1.8 Recommandations de production de logiciel open source 128
10.2 Thèmes techniques prioritaires 128
10.2.1 Profilage du calcul et des entrées/sorties en contexte parallèle . . . 128
10.2.2 Structures de données performantes 128
10.2.3 Modes de programmation PPP 128
10.2.4 Calcul sur FPGA 129
10.2.5 Calcul multi-précision 129
10.2.6 Reproductibilité numérique en contexte parallèle et hétérogène . . 129
10.2.7 Apprentissage automatique 129
10.3 Recommandations stratégiques 129
10.3.1 1. Recruter des ingénieurs pour le calcul 129
10.3.2 2. Former massivement 129
10.3.3 3. Avoir le matériel approprié pour une veille active 130
10.3.4 4. Développer une recherche appliquée interne 130
10.3.5 5. Constituer une force de frappe trans-laboratoires « IN2P3 Optimisation » 130

Au terme de cette réflexion sur l'optimisation et la parallélisation des codes de physique, nous avons voulu lister quelques enseignements à l'usage des physiciens, identifier des thèmes prioritaires pour les années à venir, et terminer par quelques recommandations à destination des décideurs de l'institut, dans le contexte de l'exercice de prospective en cours.

10.1 Enseignements à propos de la quête de performance

10.1.1 Optimisation

L'optimisation est une tâche ardue qui ne peut être effectuée dans de bonnes conditions et en obtenant une bonne accélération que si le programme en question a été un minimum conçu pour cela. Dans le cas contraire, l'optimisation consistera à réécrire le programme de départ, mais en prenant en compte ce nouvel aspect, ce qui, en fin de compte coûtera un temps de développement bien supérieur à celui de la version non-optimisée.

Pour éviter ce problème, certaines parties du programme doivent être optimisées avant la phase de développement. Ainsi, le choix du format de données est crucial de ce point de vue, car tous les calculs seront basés sur celui-ci. Un mauvais choix de format de données impliquera une réécriture quasi-complète du programme en question et le développement de convertisseurs afin de lire les données du format précédent.

Cependant, quelques règles permettent d'éviter au mieux ces problèmes. Un format de données parallélisable qui permet la vectorisation sera composé préférentiellement de structures de tableaux plutôt que de tableaux de structure. Cela permettra de fournir des interfaces exposant des opérations collectives.

Le format de données lui-même ne peut être défini si aucun calcul ne l'utilise. En effet, c'est le type de calcul qui définit le stockage optimal des données, donc c'est la conception des calculs qu'effectue le programme qui détermine son format de données.

D'une manière générale la conception d'un programme ou d'une bibliothèque est toujours la partie clé, et ce, bien avant le développement proprement dit.

10.1.2 Vectorisation

Il existe différents types de vectorisation.

L'auto-vectorisation est effectuée par le compilateur, mais nécessite un apport d'indices de la part du développeur, car le compilateur ne peut pas deviner *a priori* l'utilisation désirée. Cette méthode permet une bonne portabilité des applications.

La programmation par fonctions intrinsèques produit du code bien plus optimisé que celui produit par un compilateur au prix d'une portabilité diminuée.

La génération de code (***Tensorflow***, ***Loopy***) permet d'effectuer des optimisations que les compilateurs ne peuvent pas faire, comme adapter le code au nombre de cœurs/GPU statiquement. Cette dernière technique préserve la portabilité, mais complexifie la construction de l'exécutable puisqu'un compilateur récent doit être disponible sur la machine cible.

10.1.3 Parallélisation

À la manière de la vectorisation, différentes méthodes de parallélisation sont possibles. L'approche standard en calcul intensif consiste à utiliser les standards ***MPI*** pour la communication entre les nœuds de calcul et ***OpenMP*** pour le calcul *multi-thread* sur chaque nœud. Elle est relativement simple à mettre en œuvre dans des programmes peu abstraits (boucles sur des tableaux), mais peut devenir très complexe dès que la complexité du code et des données augmente.

Dans les langages qui permettent le développement d'abstractions sophistiquées, par exemple C++, Rust ou Scala, il est souvent plus pertinent de faire appel à des approches plus haut niveau : bibliothèques d'algorithmes parallèles prêts à l'emploi, calcul distribué ayant l'apparence de la mémoire partagée, approches fonctionnelles et par flux de données, ou encore machine learning...

À ce niveau, le C++ rattrape petit à petit son retard sur les autres langages de programmation en intégrant à sa bibliothèque standard le support natif des threads depuis C++11 et un ensemble d'algorithmes parallèles standardisés depuis C++17. Il est prévu d'introduire une manière ergonomique d'utiliser ces algorithmes en C++20, et la bibliothèque ***HPX*** préfigure peut-être l'avenir du calcul distribué dans ce langage.

Une des difficultés majeure de la programmation parallèle est le débogage des applications. En effet, la gestion d'un nombre important de processus et de *threads* entraîne un très grand nombre de fluctuations, qui, lorsqu'elles ne sont pas prévues ou connues du développeur, peuvent se révéler extrêmement problématiques pour reproduire un bogue et comprendre la cause de celui-ci. Des logiciels de débogage permettent de gérer plusieurs *threads* sur une seule machine (comme *gdb*), mais au niveau d'un cluster de calcul, de nombreux développement sont encore en cours et doivent être suivis.

Une manière de simplifier la programmation parallèle est de ne pas la pratiquer dans toute sa généralité (manipulation arbitraire de mémoire partagée et échanges arbitraires de messages), mais de se cantonner à la place à un certain paradigme de programmation parallèle bien défini, comme celui de la programmation fonctionnelle. Il devient alors plus simple de raisonner sur le fonctionnement du programme. Mais le choix du paradigme doit être effectué avec soin, car tous les paradigmes ne s'appliquent pas à toutes les applications, et il est très difficile de vérifier le bon respect d'un paradigme sans utiliser un langage de programmation qui l'intègre nativement.

10.1.4 Langages de programmation

Il n'existe pas, et ne peut pas exister, de langage de programmation universel et idéal pour le calcul scientifique, en raison d'objectifs incompatibles. Le choix du ou des langages dépendra des contraintes du projet :

— Langages complexes mais hautement expressifs (ex : C++, Rust, Scala) pour l'exploration de nouvelles techniques d'optimisation/vectorisation/parallélisation.
— Langages privilégiant la facilité de prise en main (ex : Julia, Python, Go) pour les personnels donc la programmation n'est pas le cœur de métier.
— Langages permettant l'apprentissage graduel (ex : Julia) ou combinaison de langages

(ex : Python + C++) quand on doit allier fortes performances et interface simple et souple.

Par ailleurs, il existe une tension forte entre adoption de la meilleure approche théorique et compatibilité optimale avec l'écosystème de calcul établi.

10.1.5 Calcul sur FPGA

L'installation et la mise en exploitation d'un nœud de calcul avec accélérateur FPGA nécessite un certain accompagnement technique, en même temps est assez bien documentée sur les sites du fabricant par de nombreuse fiches techniques et guides d'utilisation. La programmation OpenCL a l'avantage que le noyau peut être exécuté et vérifié d'abord sur le CPU de la machine hôte. La compilation pour le FPGA dispose aussi du mode émulation, qui offre un autre niveau de diagnose avant la compilation finale, qui est assez longue et très gourmande en ressource de mémoire et CPU (elle est par défaut multi-thread).

Le choix du FPGA détermine la liste des options de programmation OpenCL, ce qui nécessite un temps d'exploration des différentes options du standard OpenCL pour chaque situation concrète. Cela fait que l'évolution du matériel utilisé ne profite pas d'une relative standardisation comme pour les CPU ou même les GPU, mais cela pourrait changer à l'avenir. L'implication forte du personnel technique est aujourd'hui évidente comme, par exemple, dans les développements sur le superordinateur MareNostrum du Centre National de Supercalcul de Barcelone, qui a récemment lancé un appel d'embauche sur un poste d'ingénieur spécialiste avec des compétences fortes dans le domaine des FPGA.

L'utilisation des accélérateurs de calcul basé sur les FPGA s'inscrit dans le même paradigme de l'optimisation du code là où elle est absolument nécessaire, sans perdre la lisibilité du code ni affecter le cycle d'évolution du code sur la durée de vie du projet (de recherche, en général) auquel il est attaché. Les parties du code susceptibles d'être transférées sur un accélérateur doivent être bien identifiées par rapport au reste du code, qui reste à exécuter sur la machine hôte, et c'est au nouveau programmeur de s'approprier ces nouvelles méthodes.

Le prix actuel des accélérateurs avec FPGA (Intel et Xilinx) peut paraître élevé aujourd'hui, par rapport à un CPU ou même un GPU. Il est probable que cette différence reflète l'ampleur de leur utilisation dans le domaine du calcul. Les applications de calcul avec FPGA sont moins visibles, car elles sont très spécialisées et ont parfois une composante R&D assez importante. Des mesures de la significative réduction de la consommation électrique par rapport aux CPU et GPU ont été plusieurs fois présentées dans des publications de spécialité, ce qui ouvre la possibilité de construire des centres de calcul plus éco-responsables. Avant de passer à l'échelle nécessaire pour profiter de ces avantages, il faudra faire le travail d'insertion des FPGA dans les schémas de calcul traditionnel et les utiliser dans des applications concrètes qui existent déjà dans les laboratoires de l'IN2P3, afin de rassembler le plus d'expériences utilisateur.

Enfin, le choix entre les deux acteurs majeurs sur le marché des FPGA pour le calcul, Intel et Xilinx, représente un autre volet de la stratégie à suivre. Pendant que Xilinx semble offrir des FPGA plus performants (et onéreux) qui reflètent l'ancienneté et la primauté de l'entreprise dans leur fabrication, Intel (qui a acquis l'autre fabricant de FPGA, Altera) marche dans une direction plus orientée vers les solutions de calcul intégré CPU+FPGA dans un nouveau standard de processeur performant. Après une certaine expérience accumulée

avec un FPGA Intel au LPC, la mise en opération du nœud FPGA Xilinx sur la plateforme ACP au LLR permettra de compléter l'étude des FPGA par l'exploration d'un nouvel environnement.

10.1.6 Calcul sur GPU

Les GPUs sont des accélérateurs de calcul particulièrement efficaces pour traiter des calculs d'algèbre linéaire et plus généralement des cas où de nombreuses valeurs doivent être mise à jour, comme la simulation du problème à N corps en interaction gravitationnelle.

Leur consommation électrique est moindre en comparaison d'un calcul sur CPU, car leur architecture est axée sur les calculs mathématiques simples et massivement parallèles.

Néanmoins, leur utilisation efficace ne peut se faire qu'avec un format de données adéquat (vecteurs, matrices, tenseurs).

10.1.7 Problématique de la précision

Calculer vite ne suffit pas, il faut aussi produire un résultat juste. Hélas, l'approche utilisée pour répondre à ce besoin a souvent été d'avoir recours systématiquement au calcul en double précision, et de ne réfléchir aux questions de stabilité numérique que lorsqu'un résultat éminemment faux était observé durant la phase de validation. Un résultat passant cette validation est ensuite considéré comme exacte, et toute modification du code résultant en une modification de celui-ci, fût-ce d'un millionième de pourcent, est donc supposée introduire une erreur de calcul.

Cette stratégie de développement est problématique pour plusieurs raisons :

— Si la double précision est la forme de calcul flottant la plus précise dont on peut espérer une bonne portabilité entre matériels, elle est aussi la plus lente, et la plus gourmande en ressources mémoires.

— Certains matériels récents, comme les FPGA, ne possèdent pas de support natif de la double précision, et ne peuvent émuler celle-ci qu'à un coût prohibitif. D'autres, comme les GPUs, la pratiquent nativement mais avec un surcoût très important.

— À l'inverse, le matériel support de précisions réduites se développe, poussé par l'expansion rapide du *Deep Learning*. L'applicabilité de ces précisions aux calculs de physique est une question ouverte, que l'absence de cahier des charges clair sur la précision désirée en sortie d'un calcul ne permet pas d'explorer.

— La recherche d'une reproductibilité exacte des résultats de calcul parallèles par rapport aux résultats séquentiels passe souvent par des coûts de développements élevés, une consommation de ressources plus importante, et un passage à l'échelle moins bon, alors que ces calculs produisent en réalité souvent des résultats plus précis que leurs homologues séquentiels.

— La qualité de la physique produite repose intégralement sur celle de procédures de validation basées sur le test, alors que le test peut uniquement prouver la présence d'erreur, et pas leur absence.

Pour toutes ces raisons, une réflexion sur la précision numérique, ainsi que sur les problématiques liées à la reproductibilité numérique comme la génération de nombres aléatoires, doit être engagée à l'IN2P3 en collaboration avec les personnels chercheurs.

10.1.8 Recommandations de production de logiciel open source

Sous-section HORS-SUJET pour Reprises ?

— La phase de conception est cruciale pour tout ce qui suit (développement/optimisation/parallélisation/GPU) et peut même empêcher les phases d'optimisation/parallélisation/GPU
— Utilisation du logiciel de gestion de versions https ://git-scm.com/
— Utilisation de la plate-forme *gitlab* de l'IN2P3 pour sauvegarder les projets https ://-gitlab.in2p3.fr/
— Bonnes pratiques d'utilisation de ces outils : branches, *merge request*/*pull request*
— Test unitaire, lancer automatiquement sur https ://gitlab.in2p3.fr/ ou https ://github.com/
— Documentation à écrire tout de suite, car elle peut aider au développement et automatiquement mise à jour et en ligne avec https ://gitlab.in2p3.fr/ ou https ://github.com/

10.2 Thèmes techniques prioritaires

10.2.1 Profilage du calcul et des entrées/sorties en contexte parallèle

Nous avons exposé l'importance du profilage des codes à l'exécution, que ce soit sur les temps de calcul ou les échanges de données, afin d'avoir des actions ciblées pour augmenter les performances. Le problème est plus complexe dans un contexte parallèle, et le personnel de l'institut possède un savoir-faire insuffisant en la matière. On peut aussi intégrer à cette thématique le « profilage numérique », c'est-à-dire l'utilisation d'outils d'arithmétique stochastique pour identifier les instabilités du calcul flottant.

10.2.2 Structures de données performantes

Nous avons exposé l'importance du choix en amont des structures de données, afin d'avoir la possibilité ensuite d'optimiser/vectoriser/paralléliser le calcul. Il est essentiel d'étudier les structures les plus appropriées dans la majorité des cas, et de déboucher sur des recommandations pour les projets qui démarrent.

10.2.3 Modes de programmation PPP

Directives ou bibliothèques, auto-vectorisation ou instructions intrinsèques, langages C++20 / Fortran2018 / Go / Rust / Julia, OpenCL ou SyCL, programmation fonctionnelle et parallélisme de données, génération de code : les façons de programmer se multiplient. Il faut les étudier toutes, pour déterminer celles qui offrent le meilleur équilibre entre Performance, Portabilité et Praticité.

10.2.4 Calcul sur FPGA

Le FPGA voudrait détrôner le GPU, et évolue beaucoup. Mais ses caractéristiques et sa programmation sont très différents, et nous amènent aux frontières de l'électronique. La question de la précision numérique dans le calcul FPGA doit aussi faire l'objet d'une étude poussée, les études préliminaires montrant qu'il est difficile (et peut-être déraisonnable) de tirer exactement les mêmes résultats numériques d'un FPGA que d'un CPU ou d'un GPU.

10.2.5 Calcul multi-précision

Faire varier la précision des calculs est une piste sérieuse pour gagner en performance, en particulier au vu des nouveaux matériels, qui peuvent être beaucoup plus efficaces pour des précisions réduites. Il y a plusieurs façons de le faire, en ciblant une sous-partie du code, ou même en changeant dynamiquement de précision pendant un calcul itératif. Dans notre communauté encore souvent attachée au dogme du double, tout reste à faire sur ce thème.

10.2.6 Reproductibilité numérique en contexte parallèle et hétérogène

Matériel hétérogène, conteneurs, exécution concurrente, nombres pseudo-aléatoires... : la reproductibilité « bit à bit » des résultats est mise à mal. Comment y remédier ?

10.2.7 Apprentissage automatique

C'est évidemment un thème du présent, et qui concerne également les problématiques du groupe Reprises. Nous nous interrogeons sur la pertinence des bibliothèques « tout-en-un » les plus populaires, ou l'utilisation des GPUs est grandement facilitée... mais est-elle aussi efficace qu'elle pourrait l'être ?

10.3 Recommandations stratégiques

10.3.1 1. Recruter des ingénieurs pour le calcul

Après quelques décennies plutôt axées sur le génie logiciel, on peut estimer que l'institut est à niveau, et connaît (s'il ne pratique pas toujours) les outils et les méthodologies de développement qui lui sont utiles. Par contre, face à un matériel de calcul qui se diversifie et se complexifie, il devient vital de se doter de spécialistes de calcul scientifique, tant pour le développement des logiciels (DEV « calcul ») que l'administration des machines (ASR « calcul »).

10.3.2 2. Former massivement

Le travail des physiciens ne peut plus être fait dans l'ignorance des contraintes informatique. Il faut urgemment former le plus de chercheurs possibles, notamment par le recours aux tutoriels en ligne. Il ne s'agit pas d'en faire des experts, mais de leur donner les rudiments de

théorie et de jargon pour qu'ils évitent les erreurs basiques et sachent comment commencer et où trouver l'information. L'institut doit trouver des moyens d'encourager les formateurs internes et de faire reconnaître l'importance de leurs contributions.

10.3.3 3. Avoir le matériel approprié pour une veille active

Au foisonnement du matériel répond un foisonnement des offres logicielles. Il faut encourager les projets aidant à évaluer toutes ces offres, et à aider les chercheurs à s'orienter dans cette jungle. Cela suppose de disposer d'un « techlab IN2P3 » intégrant du matériel récent (GPU, FPGA), à l'image de la plate-forme amorcée par P2IO et administrée au LLR, ou de plate-formes de pré-production comme les clusters HPC et GPU du CC.

10.3.4 4. Développer une recherche appliquée interne

Les mutations informatiques en cours justifient une recherche appliquée interne, qui ne sera « excellente » que si elle se fait en association avec des laboratoires d'informatique. Il faut encourager cette collaboration en proposant des co-financements de thèses co-encadrées par l'IN2P3 et l'INS2I. Pour soutenir cette recherche, il est pertinent d'accueillir quelques chercheurs transdisciplinaires, et il faut encourager les ingénieurs informatique disposant d'une thèse à préparer une HDR.

10.3.5 5. Constituer une force de frappe trans-laboratoires « IN2P3 Optimisation »

Malgré tout l'intérêt qu'il y a déjà à constituer des projets R&D et des réseaux métiers où les participants échangent leurs expertises mutuelles, il faut reconnaître qu'entre deux réunions chacun retourne travailler isolément sur les collaborations de son laboratoire. À un moment où l'optimisation des codes devient une affaire de spécialistes, il serait légitime de constituer une forme d'équipe dédiée, distribuée sur plusieurs laboratoires, une sorte de « *task force* » qui pourrait intervenir globalement avec un fort impact sur les projets de physique.

Appendices

Annexe A

Optimisation et prédicteur de branchements

Comme nous l'avons vu dans la section 2.2.3.6 les branchements perturbent l'efficacité du préchargement des instructions par le CPU. Cet exemple montre concrètement les différences de vitesses d'exécution typiques que l'on peut attendre en fonction de la présence d'un branchement ou non.

Exemple tiré de [22] :

Soit, une fonction qui copie les valeurs d'un tableau **x** ou **y** dans un tableau **z** en fonction de la valeur d'une variable aléatoire :

$$z_i = \begin{cases} x_i & \alpha_i < p \\ y_i & \alpha_i \geq p \end{cases}, \quad x_i, y_i \in \mathbb{R}, \quad p \in [0,1], \quad \alpha_i \in U(0,1), \quad i \in 1, N$$

La fonction C++ associée à cette copie est :

```
1  void dummyCopy(float* tabResult,
2     const float * tabX, const float* tabY, const float * tabProba,
3     long unsigned int nbElement, float proba)
4    {
5    for(long unsigned int i(0lu); i < nbElement; ++i){
6      if(tabProba[i] < proba){
7        tabResult[i] = tabX[i];
8      }else{
9        tabResult[i] = tabY[i];
10     }
11   }
12 }
```

Puisque aucun calcul n'est fait dans cette fonction, les variations de la performance ne seront dues qu'à la condition ligne 6.

Il existe aussi une manière d'écrire cette fonction sans **if** :

```
void dummyCopy(float* tabResult,
    const float * tabX, const float* tabY, const float * tabProba,
    long unsigned int nbElement, float proba)
  {
  for(long unsigned int i(0lu); i < nbElement; ++i){
    float cond(tabProba[i] < proba);
    tabResult[i] = tabX[i]*cond + (1.0f - cond)*tabY[i];
  }
}
```

Bien que ces deux fonctions produisent les mêmes résultats, leurs performances sont très différentes (voir figure A.1).

Tout d'abord, on constate que les performances de la fonction avec le **if** dépendent de la valeur de la variable *proba* passé en paramètre, en dépit du fait que la quantité d'instructions exécutées n'en dépend pas.

La performance est la plus dégradée lorsque $proba = 0.5$.

On peut le comprendre en prenant en compte l'existence du prédicteur de branchement. Comme le branchement dépend d'une variable aléatoire uniforme, cela implique dans ce cas que le prédicteur de branchement a toujours une chance sur deux de se tromper.

Cela implique de vider le pipeline, opération qui coûte 300 cycles. Bien que la version sans branchements (sans **if**) charge plus de données en entrée et effectue plus de calculs par itération de boucle, ces opérations sont loin d'avoir un tel coût. C'est la raison pour laquelle elle est 4 à 10 fois plus rapide.

De plus, l'absence de branchement facilite également la vectorisation (voir chapitre 3), ce qui augmente encore les performances (voir figure A.2).

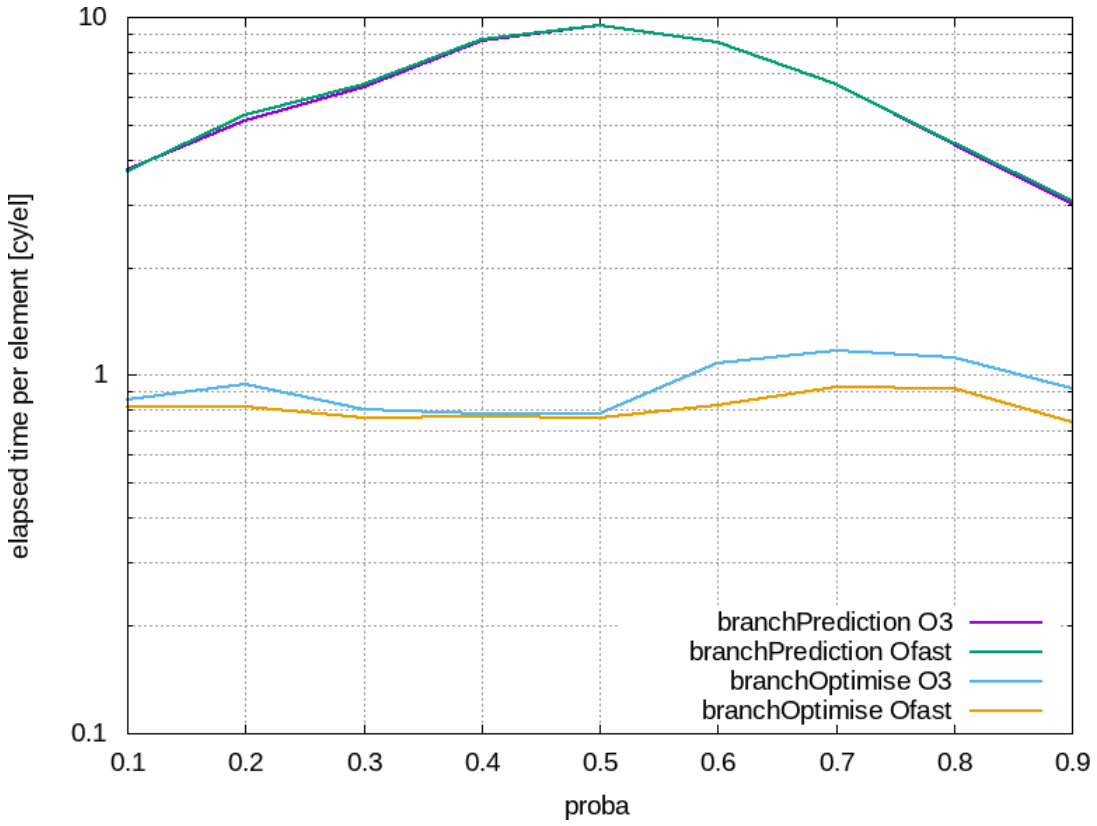

FIGURE A.1 – Performances de la fonction avec le **if** en violet et vert, et sans le **if** en bleu et orange, en fonction du seuil *proba* passé en paramètre. Aucune fonction n'est vectorisée.

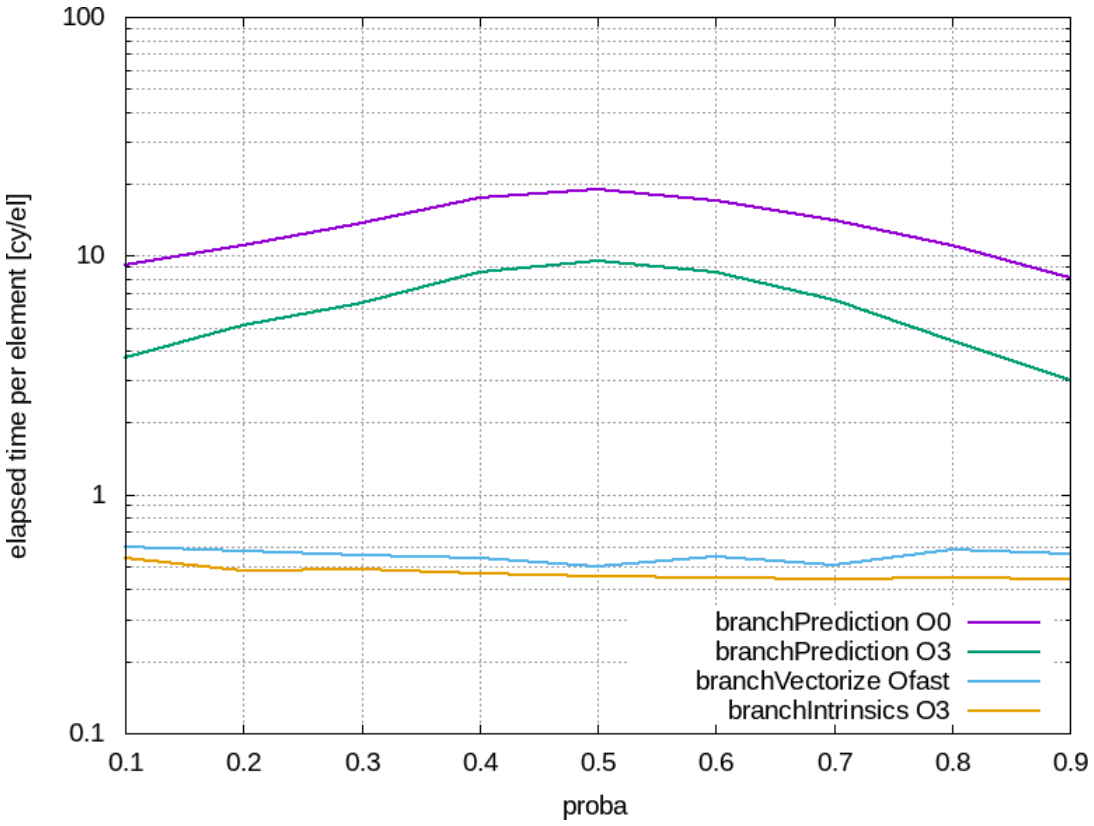

FIGURE A.2 – Performances de la fonction avec le **if** en violet et vert, et vectorisée en bleu et orange, en fonction du seuil *proba* passé en paramètre.

Bibliographie

[1] CTA Consortium. Introducing the CTA concept. *Astroparticle Physics*, 43 :3 – 18, 2013.

[2] P. J. Hall (ed). An ska engineering overview. *SKA Memorandum 91*, 2007.

[3] Valgrind. http://www.valgrind.org/.

[4] Introduction to the valgrind debugger/profiler. http://lappweb.in2p3.fr/~paubert/INTRODUCTION_VALGRIND/index.html.

[5] Maqao. http://www.maqao.org/.

[6] Gprof. https://ftp.gnu.org/old-gnu/Manuals/gprof-2.9.1/html_mono/gprof.html.

[7] Perf. https://perf.wiki.kernel.org/index.php/Tutorial.

[8] Sébastien Valat, Marc Pérache, and William Jalby. Introducing kernel-level page reuse for high performance computing. In *Proceedings of the ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, MSPC '13, pages 3 :1–3 :9, New York, NY, USA, 2013. ACM.

[9] Intel. Math kernel library. https://software.intel.com/en-us/mkl.

[10] GNU. Basic linear algebra subprograms. https://www.gnu.org/software/gsl/doc/html/blas.html?highlight=blas.

[11] Automatically tuned linear algebra software (atlas). http://math-atlas.sourceforge.net/.

[12] Lapack - linear algebra package. http://www.netlib.org/lapack/.

[13] Lapack++ - linear algebra package in c++. https://math.nist.gov/lapack++/.

[14] Eigen. https://eigen.tuxfamily.org/dox/.

[15] Armadillo, c++ library for linear algebra & scientific computing. http://arma.sourceforge.net/.

[16] Stellar Group. High performance parallex - hpx. http://stellar-group.org/libraries/hpx/.

[17] Qualité : faire simple et utile. https://qualsimp.sciencesconf.org/.

[18] *Architecture software developer's manual volume 1*. 2016.

[19] Intel. Automatically replacing malloc and other c/c++ functions for dynamic memory allocation. https://software.intel.com/en-us/node/506096.

[20] Pierre Aubert. Data format generator. https://gitlab.in2p3.fr/CTA-LAPP/PLIBS_9_DATA_GENERATOR.

[21] Pierre Aubert. *Informatique Hautes Performances pour la détection de rayons gammas*. 2018.

[22] How to optimize computation with hpc. http://lappweb.in2p3.fr/~paubert/ASTERICS_HPC/index.html.

[23] GNU. Gnu c compiler and gnu c++ compiler - gcc/g++. http://www.gcc.org/.

[24] C++ language reference. https://en.cppreference.com/w/cpp/language.

[25] CERN. Wlcg : World-wide lhc computing grid. http://cern.ch/.

[26] Fortran 09. https://www.fortran90.org/.

[27] Xsimd : C++ wrappers for simd intrinsics. https://xsimd.readthedocs.io/en/latest/.

[28] Pierre Aubert. *PLIBS 9*. https://gitlab.in2p3.fr/CTA-LAPP/PLIBS_9.git.

[29] Xtensor : Multi-dimensional arrays with broadcasting and lazy computing. https://xtensor.readthedocs.io/en/latest/.

[30] Jochen Härdtlein, Alexander Linke, and Christoph Pflaum. Fast expression templates. In Vaidy S. Sunderam, Geert Dick van Albada, Peter M. A. Sloot, and Jack J. Dongarra, editors, *Computational Science – ICCS 2005*, pages 1055–1063, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[31] Tensorflow : An end-to-end open source machine learning platform. https://www.tensorflow.org/.

[32] Loopy : A code generator for array-based code on cpus and gpus. https://github.com/inducer/loopy.

[33] Pierre Aubert. *PLIBS 8 : kernel generator*. https://gitlab.lapp.in2p3.fr/CTA-LAPP/PLIBS_8_UTILS/tree/master/PKERNEL_SHADOK.

[34] GNU. Time : Programs timer. https://www.gnu.org/software/time/.

[35] D. Heck, J. Knapp, J. N. Capdevielle, G. Schatz, and T. Thouw. CORSIKA : A Monte Carlo code to simulate extensive air showers. 1998.

[36] Bernlöhr K. Simulation of imaging atmospheric cherenkov telescopes with corsika and sim_telarray. *Astroparticle Physics*, 30 :193–316, 2008.

[37] Actis M. et al. (CTA Consortium). Design concepts for the cherenkov telescope array cta : an advanced facility for ground-based high-energy gamma-ray astronomy. *Experimental Astronomy*, 32(3) :193–316, Dec 2011.

[38] Hassan T. et al. Monte carlo performance studies for the site selection the cherenkov telescope array. *Astroparticle Physics*, 93 :76–85, 2017.

[39] Acharyya A. et al. Monte carlo studies for the optimisa- tion of the cherenkov telescope array layout. *Astroparticle Physics*, 111 :35–53, 2019.

[40] Arrabito L. et al. Performance optimization of the air shower simulation program for the cherenkov telescope array. 2019 (Accepted).

[41] Lauter C. A new open-source simd vector libm fully implemented with high-level scalar c. In *Proceedings of the 50th Asilomar Conference on Signals, Systems and Computers, Pacific Grove, United States*, 2016.

[42] Piparo D. et al. Speeding up hep experiment software with a library of fast and auto-vectorisable mathematical functions. *Journal of Physics : Conference Series*, 513(5) :052027, 2014.

[43] CTA Consortium. CTA data management technical design report version 2.0. 2016.

[44] A. Beck et. al. Adaptive SIMD optimizations in particle-in-cell codes with fine-grain particle sorting. *Computer Physics Communications*, 2019.

[45] Hierarchical data format, hdf5. https://support.hdfgroup.org/HDF5/.

[46] *Computationnal centre of IN2P3*. https://cc.in2p3.fr/.

[47] Open Group. Threading building blocks. https://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread.h.html.

[48] The OpenMP Team. The openmp api specification for parallel programming. https://www.openmp.org/.

[49] Intel. Threading building blocks. https://github.com/intel/tbb.

[50] OFI Working Group. Libfabrix : Open fabrics interfaces (ofi). https://ofiwg.github.io/libfabric/.

[51] MPI Group. Open mpi : A high performance message passing library. https://www.open-mpi.org/doc/.

[52] The ZeroMQ authors. Zeromq - an open-source universal messaging library. https://zeromq.org/.

[53] Apache Software Foundation. Hadoop. https://hadoop.apache.org.

[54] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark : Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.

[55] Map Reduce for C. Google. https://github.com/google/mr4c.

[56] Acceleware, WP-2014060-1.0. OpenCL on FPGAs for GPU programmers. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/wp/wp-201406-acceleware-opencl-on-fpgas-for-gpu-programmers.pdf.

[57] The Khronos Group : connecting software to silicon. https://www.khronos.org/about/.

[58] OpenCL 1.2 : reference pages. https://www.khronos.org/registry/OpenCL/sdk/1.2/docs/man/xhtml/.

[59] Matthew Scarpino. *OpenCL in Action*. Manning, 2012.

[60] Terasic : DE1-SoC board. https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=205&No=836&PartNo=4.

[61] Cyclone V GT : board support package for Intel FPGA SDK OpenCL. http://www.alterawiki.com/wiki/The_board_support_package_of_Cyclone_V_GT_Development_Kit_for_Intel_FPGA_SDK_OpenCL.

[62] Intel. Cyclone V Device Overview. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/cyclone-v/cv_51001.pdf.

[63] Code pour la Transformée Fourier Discrète. https://gitlab.in2p3.fr/bogdan-vulpescu/Reprises/tree/fpga/OpenCL/OpenCLBook/Ch14/rdft.

[64] Intel. Programmable Acceleration Card with Intel Arria 10 GX FPGA. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ds/ds-pac-a10.pdf.

[65] Intel. FPGA Programmable Acceleration Card D5005. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ds/ds-pac-d5005.pdf.

[66] FPGA Acceleration Stack. https://www.intel.com/content/www/us/en/programmable/solutions/acceleration-hub/acceleration-stack.html.

[67] https://ark.intel.com/content/www/us/en/ark/products/139940/intel-xeon-gold-6138p-processor-27-5m-cache-2-00-ghz.html.

[68] Semiconductor $ Computer Engineering Wiki. https://en.wikichip.org/wiki/intel/xeon_gold/6138p.

[69] Bogdan Vulpescu. https://gitlab.in2p3.fr/CodeursIntensifs/Reprises/wikis/lpc.

[70] Xilinx Alveo 280 Data Center Accelerator Card. https://www.xilinx.com/products/boards-and-kits/alveo/u280.html.

[71] SDAccel : Enabling Hardware-Accelerated Software. https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html.

[72] Official repository of the AWS EC2 FPGA Hardware and Software Development Kit. https://github.com/aws/aws-fpga.

[73] Top 500. https://www.top500.org/.

[74] Summit - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband. https://www.top500.org/system/179397.

[75] Sierra - IBM Power System S922LC, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband. https://www.top500.org/system/179398.

[76] Sunway TaihuLight - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway. https://www.top500.org/system/178764.

[77] Green 500. https://www.top500.org/green500/.

[78] DGX SaturnV Volta - NVIDIA DGX-1 Volta36, Xeon E5-2698v4 20C 2.2GHz, Infiniband EDR, NVIDIA Tesla V100. https://www.top500.org/system/179166.

[79] AI Bridging Cloud Infrastructure (ABCI) - PRIMERGY CX2570 M4, Xeon Gold 6148 20C 2.4GHz, NVIDIA Tesla V100 SXM2, Infiniband EDR. https://www.top500.org/system/179393.

[80] MareNostrum P9 CTE - IBM Power System AC922, IBM POWER9 22C 3.1GHz, Dual-rail Mellanox EDR Infiniband, NVIDIA Tesla V100. https://www.top500.org/system/179442.

[81] TSUBAME3.0 - SGI ICE XA, IP139-SXM2, Xeon E5-2680v4 14C 2.4GHz, Intel Omni-Path, NVIDIA Tesla P100 SXM2. https://www.top500.org/system/179093.

[82] PANGEA III - IBM Power System AC922, IBM POWER9 18C 3.45GHz, Dual-rail Mellanox EDR Infiniband, NVIDIA Volta GV100 . https://www.top500.org/system/179689.

[83] Advanced Computing System(PreE) - Sugon TC8600, Hygon Dhyana 32C 2GHz, Deep Computing Processor, 200Gb 6D-Torus. https://www.top500.org/system/179593.

[84] Taiwania 2 - QCT QuantaGrid D52G-4U/LC, Xeon Gold 6154 18C 3GHz, Mellanox InfiniBand EDR, NVIDIA Tesla V100 SXM2. https://www.top500.org/system/179590.

[85] Huawei G5500, Xeon E5-2650v4 12C 2.2GHz, NVIDIA Tesla V100, 100G Ethernet. https://www.top500.org/system/179597.

[86] A titre d'exemple. https://arxiv.org/abs/hep-ex/0406031.

[87] CMS-PAS-HIG-17-003. https://cds.cern.ch/record/2257067.

[88] MadGraph. https://launchpad.net/mg5amcnlo.

[89] David Chamond et al. Projet codeur intensif. https://gitlab.in2p3.fr/CodeursIntensifs/CodeursIntensifs/wikis/home.

[90] W. Kahan and J. Palmer. On a Proposed Floating-point Standard. *SIGNUM Newsl.*, 14(si-2) :13–21, October 1979.

[91] IEEE Standard for Binary Floating-Point Arithmetic. *ANSI/IEEE Std 754-1985*, pages 1–20, Oct 1985.

[92] IEEE Standard for Radix-Independent Floating-Point Arithmetic. *ANSI/IEEE Std 854-1987*, pages 1–19, Oct 1987.

[93] IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*, pages 1–70, Aug 2008.

[94] IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, July 2019.

[95] David Goldberg. What Every Computer Scientist Should Know About Floating-point Arithmetic. *ACM Comput. Surv.*, 23(1) :5–48, March 1991.

[96] W. Kahan. Prof. W. Kahan's web pages. https://people.eecs.berkeley.edu/~wkahan/.

[97] Kdevelop - a cross-platform ide for c, c++, python, qml/javascript and php. https://www.kdevelop.org/.

[98] Pycharm - the python ide for professional developers. https://www.jetbrains.com/pycharm/.

[99] Sublimetext - a sophisticated text editor for code, markup and prose. https://www.sublimetext.com/.

[100] CMake group. Cmake is an open-source, cross-platform family of tools designed to build, test and package software. https://cmake.org/.

[101] LLVM. Clang : a c language family frontend for llvm. https://clang.llvm.org/.

[102] Dimitri van Heesch. Doxygen - generate documentation from source code. http://www.doxygen.nl/.

[103] Georg Brandl. Sphynx - python documentation generator. http://www.sphinx-doc.org/en/master/.

[104] Linus Torvalds. Git - fast version control. https://git-scm.com/.

[105] Gitlab : plateforme de développement privée. https://gitlab.in2p3.fr/.

[106] Github : plateforme de développement gratuite. https://github.com/.