UNIVERSITÉ DE REIMS CHAMPAGNE-ARDENNE

École Doctorale SNI, Sciences de l'Ingénieur et du Numérique

THÈSE

Pour obtenir le grade de :

Docteur de l'Université de Reims Champagne-Ardenne

Discipline : **Informatique**

Spécialité : **Calcul Haute Performance**

Présentée et soutenue publiquement par

Guillaume Colin de Verdière

le 16 octobre 2019

A la recherche de la haute performance pour les codes de calcul et la visualisation scientifique

Searching for the highest performance for simulation codes and scientific visualization

---

Thèse dirigée par **Michaël Krajecki, Professeur des Universités**

JURY

Michaël Krajecki	Professeur des universités, Université de Reims Champagne-Ardenne	Directeur
François Bodin	Professeur des universités, Université de Rennes 1	Rapporteur
William Jalby	Professeur des universités, Université de Versailles-St Quentin	Rapporteur
Olivier Flauzac	Professeur des universités, Université de Reims Champagne-Ardenne	Examinateur
Guillaume Gellé	Professeur des universités, Université de Reims Champagne-Ardenne	Examinateur
Charles Hansen	Professeur des universités, Université de l'Utah, USA	Examinateur
Céline Hudelot	Professeur des universités, Ecole CentraleSupelec	Examinateur
Jean-Marc Denis	Distinguished Expert, Atos	Invité

CReSTIC - Centre de Recherche en Sciences et Technologies de l'Information et de la Communication de l'URCA EA3804 Équipe Calcul et Autonomie dans les Systèmes Hétérogènes

Titre français : **A la recherche de la haute performance pour les codes de calcul et la visualisation scientifique**

Cette thèse vise à démontrer que l'algorithmique et la programmation, dans un contexte de calcul haute performance (HPC), ne peuvent être envisagées sans tenir compte de l'architecture matérielle des supercalculateurs car cette dernière est régulièrement remise en cause. Après avoir rappelé quelques définitions relatives aux codes et au parallélisme, nous montrons que l'analyse des différentes générations de supercalculateurs, présents au CEA lors de ces 30 dernières années, permet de dégager des points de vigilances et des recommandations de bonnes pratiques en direction des développeurs de code. En se reposant sur plusieurs expériences, nous montrons comment viser une performance adaptée aux supercalculateurs et comment essayer d'atteindre la performance portable voire la performance extrême dans le monde du massivement parallèle, incluant ou non l'usage de GPU. Nous expliquons que les logiciels et matériels dédiés au dépouillement graphique des résultats de calcul suivent les mêmes principes de parallélisme que pour les grands codes scientifiques, impliquant de devoir maîtriser une vue globale de la chaîne de simulation. Enfin, nous montrons quelles sont les tendances et contraintes qui vont s'imposer à la conception des futurs supercalculateurs de classe exaflopique, impactant de fait le développement des prochaines générations de codes de calcul.

Mots clés : HPC, Simulation numérique, codes de calcul, visualisation scientifique, GPU, parallélisme

English title : **Searching for the highest performance for simulation codes and scientific visualization**

This thesis aims to demonstrate that algorithms and coding, in a high performance computing (HPC) context, can not be envisioned without taking into account the hardware at the core of supercomputers since those machines evolve dramatically over time. After setting a few definitions relating to scientific codes and parallelism, we show that the analysis of the different generations of supercomputer used at CEA over the past 30 years allows to exhibit a number of attention points and best practices toward code developers. Based on some experiments, we show how to aim at code performance suited to the usage of supercomputers, how to try to get portable performance and possibly extreme performance in the world of massive parallelism, potentially using GPUs. We explain that graphical post-processing software and hardware follow the same parallelism principles as large scientific codes, requiring to master a global view of the simulation chain. Last, we describe tendencies and constraints that will be forced on the new generations of exaflopic class supercomputers. These evolutions will, yet again, impact the development of the next generations of scientific codes.

Key works : HPC, numerical simulation, scientific code, scientific visualization, GPU, parallelism

Discipline : Informatique
Spécialité : Calcul Haute Performance

Université de Reims Champagne-Ardenne
CReSTIC – EA3804
UFR Sciences Exactes et Naturelles,
Moulin de la Housse, 51867 Reims

Remerciements

Je tiens tout d'abord à remercier le CEA/DAM qui m'a permis de traiter de sujets passionnants tout au long de ma carrière et d'avoir accès à des matériels hors norme. C'est grâce à cette noble institution, aux richesses bien souvent méconnues, que j'ai pu acquérir ce savoir informatique présenté dans ce manuscrit.

Je remercie le professeur Michaël Krajecki qui m'a fait l'amitié de m'accompagner durant cette thèse. J'associe à ces remerciements Arnaud Renard, à qui je dois ce grand plongeon dans une thèse en VAE. Je suis aussi sensible à l'honneur que m'a fait le président Gellé d'être examinateur et président de mon jury.

Un grand remerciement va au professeur Charles Hansen pour m'avoir accueilli dans son équipe lors de mon séjour au Los Alamos National Laboratory de 1994 à 1995 et m'avoir fait découvrir le parallélisme en graphique, au travers du rendu volumique. Son amitié, indéfectible depuis lors, est précieuse pour moi. Je suis particulièrement touché de sa présence dans mon jury, lui qui vient de si loin spécialement pour cette occasion. Il sait que notre maison lui sera toujours ouverte.

Depuis plus de 10 ans, dans de nombreuses occasions, j'ai eu l'occasion d'avoir des débats passionnés avec les professeurs Bodin et Jalby. Ne pas les avoir comme rapporteurs de cette thèse aurait été impensable. Qu'ils en soient remerciés.

Je veux remercier Mme Céline Hudelot, de m'avoir fait l'amitié d'être dans mon jury en souvenir des thésards que nous avons suivis dans le passé. C'était à mon tour d'être le jeune doctorant.

Merci aussi à Jean-Marc Denis d'avoir accepté d'être membre industriel invité. Il est le témoin, au travers des projets auxquels nous collaborons, des évolutions annoncées dans la dernière partie de ce manuscrit.

Je tiens enfin et surtout à remercier Sophie Colin de Verdière, mon épouse, qui m'a soutenu tout au long de ce projet de thèse. Je lui dédis tout particulièrement ce travail.

– ♦ ∞ ♦ ∞ ♦ ∞ ♦ ∞ ♦ ∞ ♦ ∞ ♦ –

Table des matières

Table des figures

Liste des tableaux

1 Introduction

1.1 Contexte

Mon activité au sein du CEA/DAM [1] a toujours été centrée sur la Simulation Numérique et le calcul haute performance (ou HPC pour **H**igh **P**erformance **C**omputing). La Simulation Numérique est l'un des trois piliers du Programme Simulation conduit par le CEA/DAM pour assurer sa mission liée à l'effort de dissuasion de la France. Les deux autres piliers (modèles physiques et expérimentation) se nourrissent aussi des capacités de traitement apportées par la simulation numérique. En tant qu'acteur de la Simulation Numérique, je vois donc cette dernière comme au centre d'un édifice indispensable à la conduite des programmes du CEA/DAM.

La simulation numérique repose sur une composante logicielle qui elle-même s'appuie sur une composante matérielle (Figure 1.1). Pour le matériel, on trouve les divers types de calculateurs, les réseaux, les moyens de stockage de l'information, les moyens de visualisation des résultats (écrans, mur d'images, imprimantes diverses). Pour le logiciel, ce seront les divers codes de calcul [2], les gestionnaires de bases de données (constantes physiques, résultats de calculs), les outils de maillage et les outils de dépouillement [3] des résultats (le plus souvent de visualisation).

Ces deux composantes sont intimement liées. Les progrès du matériel influent sur les logiciels (nouveaux paradigmes de programmation possibles aux changements de calculateurs par exemple). Les progrès des logiciels de simulation (modèles physiques plus complexes et constantes physiques plus précises) et des besoins des utilisateurs croissants (compréhension de détails toujours plus fins) nécessitent des moyens informatiques toujours plus performants.

Les constantes de temps de ces deux composantes sont très différentes : si les ordinateurs de production sont changés tous les 5 à 7 ans, les grands codes de calcul ont une durée de vie de plusieurs décennies. Au CEA/DAM, nous comptons environ 10 ans pour le développement d'un nouveau code multi-physique, 5 ans pour sa validation et de 15 à 20 ans de vie en production tout en subissant une évolution continue. Un grand code verra passer au moins 6 générations de calculateurs. La probabilité que l'architecture du

1. CEA/DAM : **C**ommissariat à l'**E**nergie **A**tomique, **D**irection des **A**pplications **M**ilitaires. Les armes nucléaires, les réacteurs nucléaires pour la propulsion navale et les matières nucléaires stratégiques sont au cœur de sa mission pour la dissuasion nucléaire française. Forte de son expertise, la DAM soutient les Autorités nationales dans la lutte contre la prolifération nucléaire, la lutte contre le terrorisme, et certains domaines de l'armement conventionnel. Pour réussir sa mission dans le respect des coûts et des délais impartis par l'État, la DAM relève sans cesse des défis scientifiques et technologiques de premier plan.

2. Un code de calcul est un logiciel informatique complexe permettant de simuler un ou plusieurs phénomènes physiques. Un programme prend le statut de code dès qu'il traite de plusieurs équations (plus ou moins couplées), qu'il atteint un nombre de lignes de source important et qu'il a vocation à devenir un outil de production, au sens d'avoir une communauté d'utilisateurs l'utilisant régulièrement.

3. Par outil de dépouillement, il faut entendre un logiciel qui utilisera les résultats bruts produits par les codes de calcul pour en déduire des données secondaires aisées à calculer [et donc inutile à stocker dans la base produite par le code]. Ces nouvelles données pourront alimenter de nouvelles bases de données ou être affichées sous forme de courbes ou d'images. Historiquement ces outils sont interactifs mais pas nécessairement, surtout avec la montée en puissance des méthodes de dépouillement dites *in situ*.

premier soit identique à celle du sixième est nulle comme cela sera démontré plus loin dans la partie historique des ordinateurs du CEA/DAM.

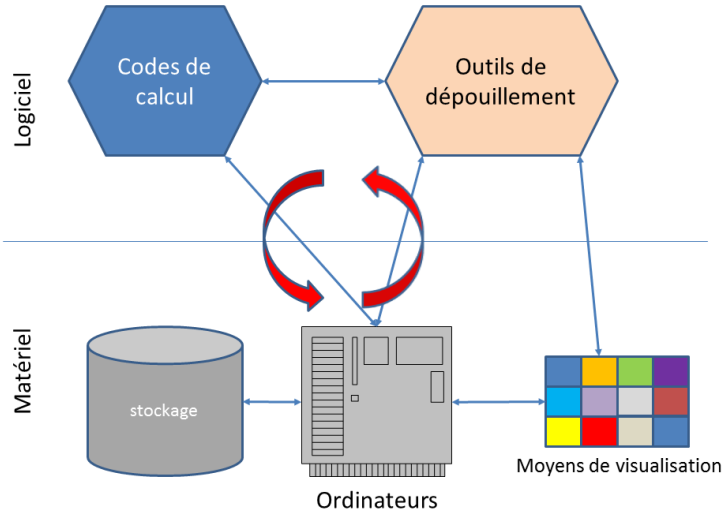


FIGURE 1.1 – Les composants de la Simulation Numérique sont à la fois matériels et logiciels. Ils influent l'un sur l'autre.

1.2 Enjeux et plan

L'enjeu principal est donc de s'assurer en permanence de la bonne adéquation des moyens logiciels et matériels aux justes besoins de la mission. Cela implique donc une recherche permanente des meilleures solutions dans tous les domaines. Pour la partie codes de calcul cela couvre la modélisation de la physique, les méthodes numériques, l'architecture logicielle et aussi la nécessaire optimisation des algorithmes pour utiliser au mieux les ordinateurs. Pour les outils de dépouillement, il s'agit de proposer les meilleures représentations possibles à l'utilisateur, obtenues dans des temps les plus courts possibles. Pour la partie matérielle, il faut s'assurer que les données produites pourront être stockées (de façon fiable), que les calculateurs sont assez puissants pour réaliser les simulations en un temps raisonnable et que les moyens de visualisation seront adaptés à des simulations de tailles croissantes.

Mon parcours professionnel s'inscrit dans cette démarche de rechercher à tout moment la meilleure solution pour chacun des composants logiciel ou matériel. Comme illustré sur la figure 1.2, tout au long des années, mes activités ont tourné autour des ordinateurs de puissance avec une alternance code – graphique, chaque étape se nourrissant de la précédente.

> **Verrou scientifique :** Ce manuscrit vise à démontrer que l'algorithmique et la programmation dans un contexte HPC ne peuvent être envisagées sans tenir compte de l'architecture matérielle des supercalculateurs car cette dernière est régulièrement remise en cause.
> Dans le même temps, comme il s'agit de maintenir des codes sur des décennies, soit bien plus que la durée de vie d'un calculateur, il est donc indispensable d'adopter des stratégies de développement de haut niveau qui devront pouvoir être mis en œuvre sur différentes générations de supercalculateurs.

Les diverses recherches réalisées au cours de ma carrière, doivent contribuer, *in fine*, à la définition de ce que pourra être une machine de classe exaflopique [4] dont le CEA/DAM cherche à s'équiper à partir de 2023.

4. 1 Exaflop/s = 10^{18} opérations flottantes par seconde.

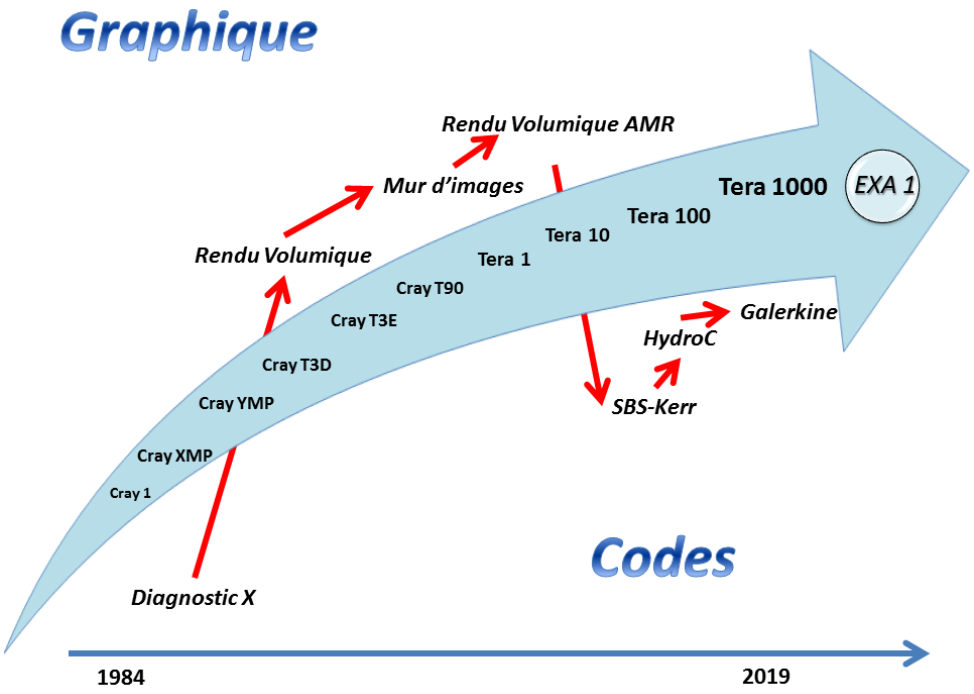


FIGURE 1.2 – Mon parcours professionnel a suivi l'évolution des machines de puissance du CEA/DAM avec une alternance codes et outils de dépouillement. Seules les étapes décrites dans ce manuscrit sont indiquées.

Ce manuscrit se compose de trois parties dans lesquelles, à la fin de chaque chapitre, nous nous attacherons à identifier des points de vigilance et aussi à mettre en avant des bonnes pratiques :

- la première partie est un état de l'art glissant car les technologies informatiques ont beaucoup évolué entre 1984 et 2019. Après un rappel de quelques notions générales relatives aux codes (chapitre 2), introduisant les notions utilisées par la suite, le chapitre 3 décrira les différents ordinateurs sur lesquels j'ai été amené à travailler. Nous détaillerons les points importants de chacune des architectures qui auront un impact sur les codes de calcul ;
- la deuxième partie décrit le corps de mes travaux sur la recherche de la performance tant pour les codes de calcul que pour le graphique matériel et logiciel. Le chapitre 5 donnera une vision orientée codes de calcul, à la recherche de la performance et de la portabilité. Ensuite, au chapitre 6, je montrerai comment l'étude du post-traitement graphique et plus particulièrement du rendu volumique m'a permis de mettre en œuvre le parallélisme tant sur le plan logiciel que matériel, dans une approche finalement commune à celle des codes de calcul ;
- la troisième et dernière partie sera la conclusion de ce manuscrit. Elle montrera qu'effectivement l'algorithmique et la programmation dans un contexte HPC ne peuvent être envisagées sans tenir compte de l'architecture matérielle des supercalculateurs. Cette conclusion (chapitre 8) décrira pourquoi toutes ces recherches sont les fondations des réflexions visant à la réalisation de machines exaflopiques. Les principales conclusions se déclinent sous deux rubriques : des points de vigilance et des bonnes pratiques à conserver en tête en permanence. Les points de vigilance couvrent la conception de la solution logicielle (parallélisme sous toutes ses formes, choix sur la localisation des traitements) mais aussi la programmation qui ne doit pas faire abstraction du matériel utilisé (prise en compte des caches, vectorisation, utilisation d'un langage adapté, recours à des accélérateurs, ...). Les bonnes pratiques en découlent comme penser « parallèle » dès l'origine du projet, concevoir ses structures de données en pesant à la performance ou encore n'utiliser que des constructions standardisées. Enfin, pour clore cette troisième partie, les enjeux de l'exascale seront abordés au chapitre 9.1 et le chapitre 9.2 donnera des pistes d'évolutions du HPC.

Remarque : les activités du CEA/DAM sont de nature sensible et donc ne se prêtent pas forcément à publication. Chaque fois que c'est possible, les publications reposent sur des données neutres ou traitent d'algorithmes généraux. A ce titre, certains détails de mes activités et des mesures de performances associées ne seront donc pas détaillés pour respecter la confidentialité des usages. De plus, étant dans des équipes de production de nombreuses années, mes activités s'y rapportant ne seront pas traitées ici.

– ♦ ∞ ♦ ∞ ♦ ∞ ♦ ∞ ♦ ∞ ♦ ∞ ♦ –

Première partie

Etat de l'art

L'accès aux calculateurs de puissance a toujours été un enjeu majeur pour le CEA/DAM. Éléments contribuant à la crédibilité de la dissuasion, les supercalculateurs sont indispensables pour résoudre les problèmes de physique qui se posent aux ingénieurs du CEA. En effet, les simulations traitent de physiques différentes très couplées, sur des objets complexes dont la discrétisation conduit à des maillages de grandes tailles. En parallèle, plus les moyens expérimentaux progressent, plus les modélisations doivent être précises et détaillées pour restituer les expériences de laboratoire. Les modélisations servent aussi à prévoir des expériences (simulations prédictives). Cette double capacité de prédiction et de restitution est à la base de la garantie des armes nucléaires par la simulation. Seuls des supercalculateurs d'exception permettent de conduire ces simulations en un temps raisonnable (pouvant aller de quelques heures à quelques semaines). Le CEA/DAM s'est donc attaché dès l'origine à s'équiper des meilleurs supercalculateurs du marché voire participer à la conception de solutions innovantes.

Les types de logiciels possibles sont intimement liés aux capacités des diverses générations de supercalculateurs. Les premières simulations étaient mono-dimensionnelles. Puis les tailles mémoires augmentant, elles sont devenues bi-dimensionnelles pour devenir tri-dimensionnelles avec les dernières générations de supercalculateurs. Dans tous les cas, optimiser les codes et les outils de dépouillement associés a été une priorité. Or optimiser un logiciel implique de bien connaître le matériel sous-jacent. Ce sera l'objet de cette première partie.

En préambule (chapitre 2), je rappellerai quelques notions relatives aux codes de calcul, ainsi qu'aux notions informatiques qui les sous-tendent, pour poser le contexte de certaines remarques. Puis je m'intéresserai aux différentes machines qui m'ont permis d'avancer dans mes recherches sur la performance des codes et outils de dépouillement. Bien que le CEA/DAM fût équipé de calculateurs dès 1959 (un BULL Gamma 3 de 2 kFlops), je ne peux décrire que les supercalculateurs qui ont influé sur mes activités (depuis mon arrivée au CEA en 1984). Pour chacune des ces machines, je mettrai en avant les points qui ont impacté les divers logiciels développés au CEA/DAM en couvrant tant les aspects matériels que logiciels.

Les premiers supercalculateurs utilisés rentrent dans la catégorie des machines dites vectorielles. Ces machines exceptionnelles ont influencé la structuration des codes par leurs caractéristiques uniques. Ce sera l'objet de la section 3.1.

Puis viendront les clusters de SMP, décrits en section 3.2, qui remplaceront les machines CRAY. Notons que cela n'implique pas la disparition du vectoriel qui a perduré notamment chez les constructeurs japonais NEC et Fujitsu, pour la plus grande satisfaction de la communauté du climat dont les codes sont restés vectoriels jusqu'à aujourd'hui.

Ces deux classes de machines se sont accompagnées de machines plus prospectives permettant d'anticiper les évolutions des supercalculateurs. Elles sont décrites à la section 3.3.

La section 3.4 fera une classification des trois types de machine précédemment décrites.

Enfin, à la section 3.5, je décrirai comment sont comparés entre eux les supercalculateurs et montrerai la difficulté d'obtenir un classement unique.

2 Quelques définitions concernant les codes

Tout d'abord précisons quelques notions relatives aux codes de calcul et notamment ceux concernant la dynamique des fluides. Le CEA/DAM utilise en effet beaucoup ces codes, dits de CFD [1], dans ses missions. L'idée ici n'est pas de faire un survol des méthodes d'analyse numérique mais de préciser les termes qui apparaissent tout au long de ce manuscrit.

Phillip Colella [Col04] classe les algorithmes rencontrés en 7 classes qu'il appelle les 7 nains du HPC par analogie aux personnages des 7 nains de la mine qui ont leurs caractéristiques (personnalités) propres.

1. Les algorithmes sur grilles structurées
2. Les algorithmes sur grilles non structurées
3. L'algèbre linéaire dense
4. L'algèbre linéaire creuse
5. Les transformées de Fourier rapides
6. Les méthodes particulaires
7. Les méthodes de Monte Carlo

Dans ce manuscrit, je ne m'intéresserai qu'aux classes 1 et 2 d'algorithmes qui feront l'objet des définitions de la section 2.1 puis elles seront utilisées dans la section 5.1 sur ma contribution aux codes de calcul et sur celle (section 6.1) consacrée au rendu volumique.

2.1 Différents types de maillages

Pour préciser un peu plus le contexte, les codes de CFD cherchent à résoudre des équations aux dérivées partielles telles que (2.1), ici en mono dimensionnel (ou 1D). Pour ce faire, l'équation est discrétisée en espace et en temps. La discrétisation en espace se fait usuellement au moyen d'un maillage (nains **1** et **2** de Colella). Les figures 2.1 et 2.2 montrent 3 types de maillages en 2D et 3D. Sur ce maillage plusieurs méthodes de résolution peuvent être utilisées. On distingue classiquement des méthodes explicites et implicites, quoique des variations entre les deux existent.

$$\frac{\partial F}{\partial t} = A\frac{\partial^2 F}{\partial x^2} + B\frac{\partial F}{\partial x} + F \tag{2.1}$$

La méthode explicite calcule $\frac{\partial}{\partial t}F_{(x,t+\frac{\Delta t}{2})}$ en fonction de $F_{(x,t)}$ et aussi des termes $F_{(x-\Delta x,t)}$ et $F_{(x+\Delta x,t)}$. La méthode est rapide mais numériquement instable si le pas en temps n'est pas assez petit (condition

1. CFD : Computational Fluid Dynamic ou code de dynamique des fluides.

dite CFL [2]). La méthode implicite calcule $\frac{\partial}{\partial t} F_{(x,t+\frac{\Delta t}{2})}$ en fonction de $F_{(x,t+\Delta t)}$ mais aussi des termes $F_{(x-\Delta x,t+\Delta t)}$ et $F_{(x+\Delta x,t+\Delta t)}$. La méthode est plus lente car elle repose sur du calcul matriciel mais elle est stable numériquement.

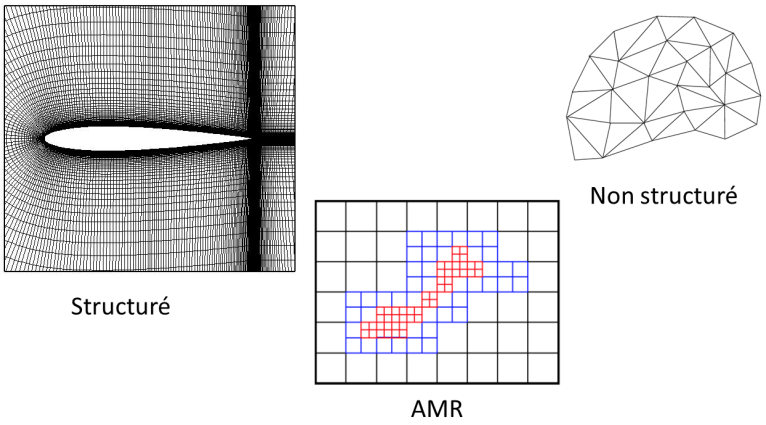


FIGURE 2.1 – Différentes natures de maillages en deux dimensions.

La nature du maillage influe sur la manière de coder l'algorithme. Un maillage structuré étant isomorphe à une matrice, il est aisé de se déplacer d'un élément à l'autre par simple variation des indices. Par exemple, en 2D, $maille(i, j)$ aura pour voisin de droite $maille(i + 1, j)$. Les compilateurs sont depuis longtemps optimisés pour reconnaître de telles structures et donc de générer le meilleur code possible. Un exemple de ce type de maillage sera présenté avec le code laser à la section 5.1.

Un maillage non structuré (comme son nom l'indique) n'a pas de structure régulière associée. Il est donc nécessaire de manipuler des tables d'indirections pour se déplacer d'un élément à l'autre. Sans de grandes précautions, il sera difficile d'assurer la contiguïté des données, condition importante pour la vectorisation du programme qui sera présentée à la section 3.3.3.1.

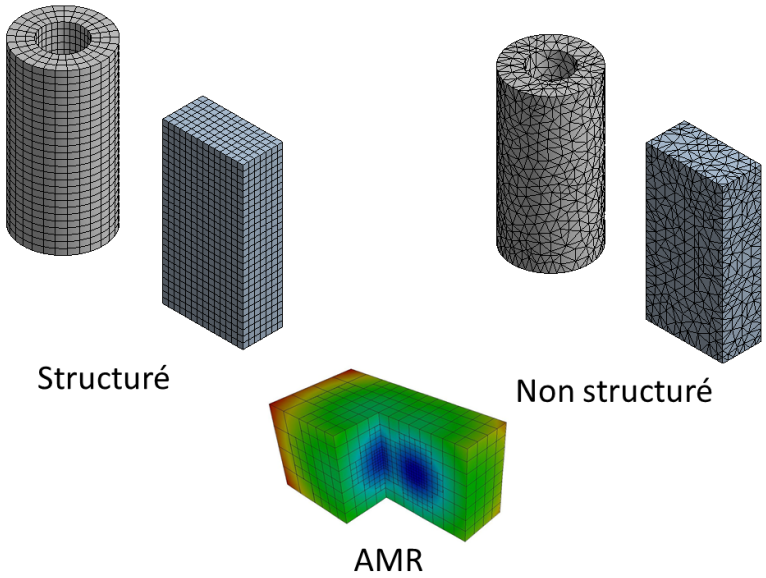


FIGURE 2.2 – Différentes natures de maillages en trois dimensions.

2. condition de Courant Friedrichs Lewy, de la forme $\Delta t \leq C\Delta x^{\alpha}$ avec $\alpha > 1$, qui impose le pas en temps de la simulation en fonction du pas d'espace. Plus le maillage est raffiné, plus le pas en temps est petit donc le nombre d'itérations du calcul (et donc le temps calcul) croît pour arriver à un temps physique donné. Le pas en temps global, déterminé à chaque itération, sera le *minimum* des pas en temps évalués pour chacune des mailles de la géométrie.

Un maillage AMR[3] présente une structuration en arbre (Figure 2.3) qui le place à mi-chemin entre maillage complètement structuré et non structuré. Il y a une certaine régularité des données pour un niveau de profondeur donné mais le passage d'un niveau à l'autre introduit des indirections. On distingue des maillages AMR en arbres purs (≪ tree based ≫) ou par zone (≪ patch based ≫) mais dans ce document nous ne considérerons que la première sorte. Ce type de maillage sera évoqué à la section 6.1 sur le rendu volumique AMR.

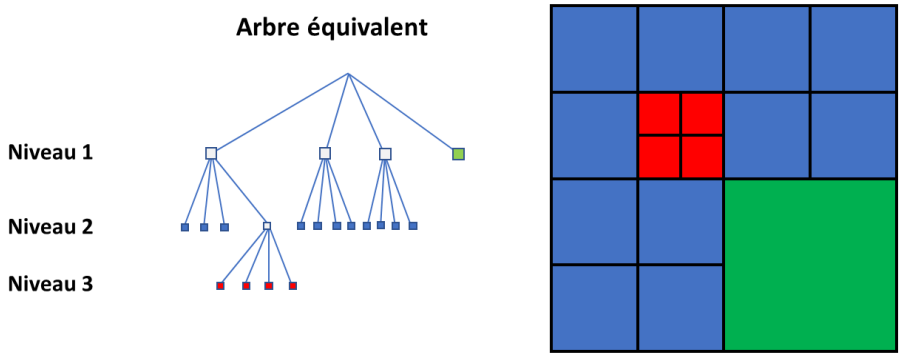


FIGURE 2.3 – Représentation arborescente d'un maillage AMR 2D.

Les simulations réalisées au CEA nécessitent des très grandes tailles de maillage soit parce que la physique l'impose, par exemple la taille d'une maille doit être plus petite que la longueur d'onde étudiée, soit pour comprendre l'influence de détails technologiques ou encore pour simuler le fonctionnement de grands instruments comme le tokamak Iter en y étudiant des phénomènes de petite taille soit plusieurs ordres de grandeur d'échelle. Il est donc nécessaire d'utiliser un supercalculateur pour mener à bien ces simulations, celui-ci apportant de grandes tailles mémoire tout comme la possibilité de réduire significativement les temps de calcul.

Pour implémenter les divers algorithmes d'intérêt sur un supercalculateur, le programmeur va se reposer sur un ou plusieurs modèles de mémoire ainsi qu'un ou plusieurs modèles de parallélisme, selon la performance recherchée et la dimension du problème à traiter. Dans le cas du HPC moderne, les développeurs doivent maîtriser plusieurs modèles de front au sein du code. Les deux sections suivantes détaillent les modèles de parallélisme et de mémoire qui existent. Les sections qui suivront expliqueront l'application de ces modèles aux codes.

2.2 Le parallélisme

Un ordinateur idéal, mais impossible à réaliser, disposerait d'un processeur infiniment rapide et d'une mémoire infinie. L'utilisation de plusieurs ordinateurs et/ou processeurs en parallèle va permettre de démultiplier la puissance de calcul disponible. Avant de décrire quels sont les différents types de parallélisme, il est intéressant de préciser ce que l'on peut attendre du parallélisme en terme performance.

2.2.1 La mesure de la performance

2.2.1.1 Loi d'Amdahl

Gene Amdahl [Amd67] a quantifié, dès 1967, le gain que peut apporter l'utilisation du parallélisme au sein d'un code.

3. AMR pour Adaptative Mesh Refinement = maillage à raffinement adaptatif

Cette loi s'écrit :

$$S_N = \frac{1}{(1-p) + \frac{p}{N}} \tag{2.2}$$

où S_N est le facteur d'accélération, p est la fraction du programme parallélisable, N le nombre de processeurs mis en œuvre pour le parallélisme. Comme :

$$\lim_{N \to \infty} S_N = \frac{1}{1-p} \tag{2.3}$$

on voit que la partie intrinsèquement séquentielle $(1-p)$ fixe la limite supérieure de l'accélération possible d'un code parallélisé sur un nombre infini de processeurs. Pour un code parallélisé à 90%, la limite d'accélération est $\frac{1}{1-0.9} = 10$. Il est donc essentiel de réduire au maximum la partie purement séquentielle en évitant par exemple les barrières (`MPI_Barrier`) entre processus ou travailler sur des algorithmes sans verrous (« lock free ») dans le cas de la parallélisation par thread.

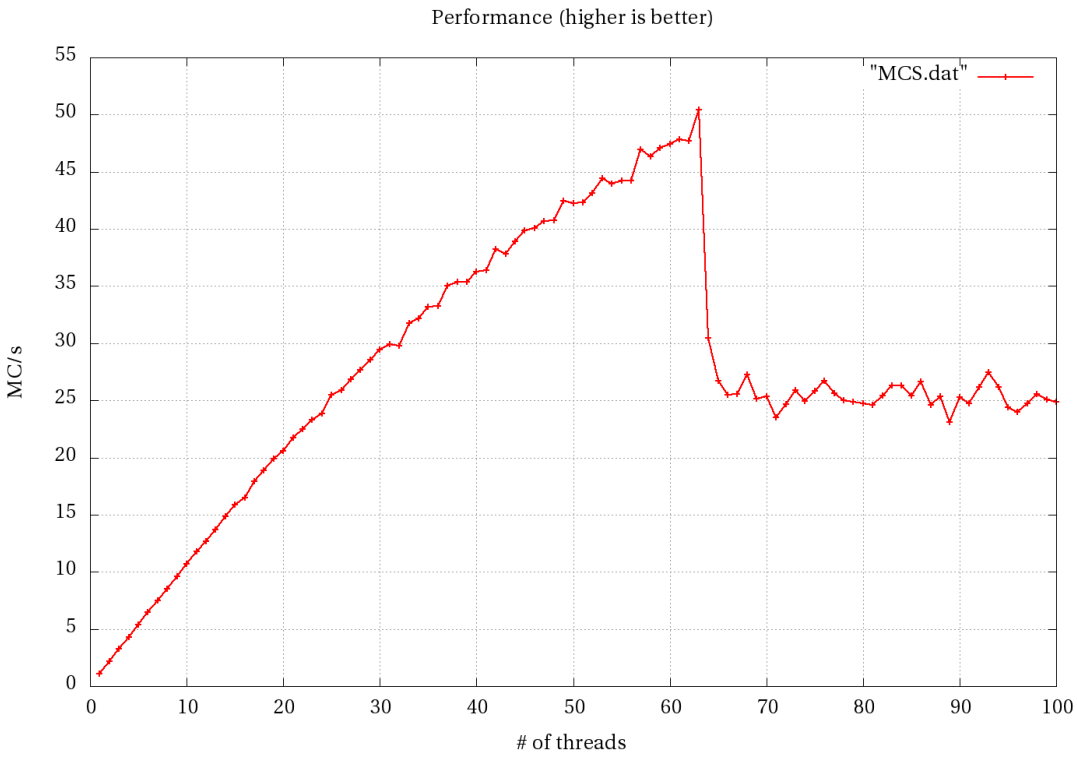


FIGURE 2.4 – Exemple de mesure d'extensibilité forte : la courbe représente la variation de performance (mesurée ici en millions de cellules traitées par seconde [MC/s]) d'une application OpenMP en fonction du nombre de threads utilisés sur une machine à 2x32 cœurs. Le placement des threads est de type compact, c'est à dire que l'on va chercher à affecter les threads d'abord au premier processeur, puis au second quand le premier est plein. Au final, on voit apparaître un effet de saturation quand le premier processeur est pratiquement rempli (décroché à 32 threads). Ensuite, quand on a plus de threads que de cœurs (≥ 64), la gestion des threads devient prédominante et bloque les performances.

2.2.1.2 Extensibilité

La qualité de la parallélisation d'un code se mesure par ses extensibilités fortes et faibles, qui se définissent comme suit :

- extensibilité forte [4] : Pour un problème de taille donnée, le temps de simulation est divisé par le nombre de processeurs mis en jeu pour la parallélisation. Toute partie restée séquentielle produira

4. strong scaling en anglais

un écart à cette loi jusqu'à dominer complètement pour un nombre de processeurs très élevé (conséquence de la loi d'Amdahl). La figure 2.4 illustre cette extensibilité dans un cas réel : on y voit clairement apparaître 3 zones. De 1 à environ 20 threads l'extensibilité est bonne. De 20 à 64, elle se dégrade nettement mais on continue à observer un gain de performances. Au delà de 64 threads, il n'y a plus d'extensibilité. Il est courant de parler d'accélération (« speedup » en anglais) qui est défini par l'équation (2.4) :

$$speedup = S_n = \frac{T_1}{T_n} \tag{2.4}$$

où T_1 est le temps d'exécution de l'application pour un processus et T_n le temps d'exécution sur n processus. Une autre mesure donne une bonne idée du comportement de l'application en parallèle : c'est son efficacité parallèle définie par l'équation (2.5) :

$$E_n = \frac{S_n}{n} = \frac{T_1}{nT_n} \tag{2.5}$$

et illustrée par la figure 2.5. Idéalement la courbe d'efficacité devrait rester à 100 %. En pratique une efficacité de plus de 80 % (à grand nombre de processus parallèles) est considérée comme bonne ;

- extensibilité faible [5] : Pour une quantité de travail donnée par processeur, le temps de simulation reste constant si la taille de la simulation croît en proportion du nombre de processeurs mis en jeu.

En pratique l'extensibilité faible est plus facile à atteindre que la forte car elle est moins sujette à la loi d'Amdahl. Un code idéal se comportera linéairement en extensibilité forte et sera stable en extensibilité faible. L'arrivée des machines pétaflopiques a permis de vérifier les extensibilités de certains codes sur plusieurs dizaines de milliers de cœurs.

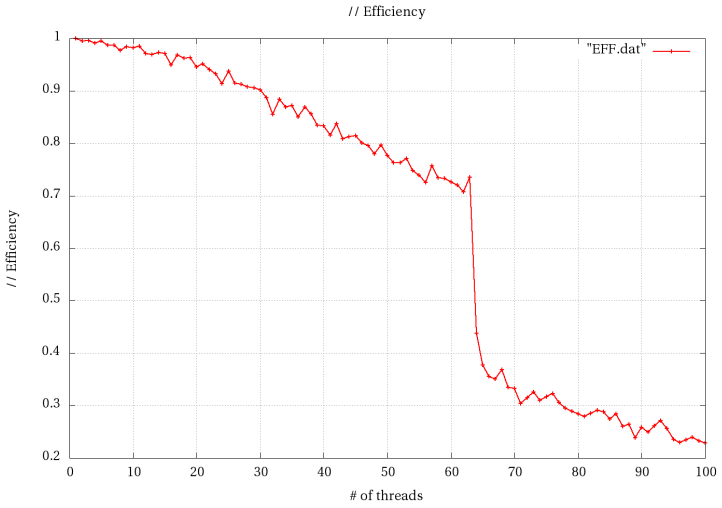


FIGURE 2.5 – Cette courbe représente l'efficacité parallèle de l'application de la figure 2.4. Le changement de comportement à 32 puis 64 threads y est visible.

2.2.2 Contrôle du parallélisme

En reprenant la taxonomie de Flynn [Fly72], nous distinguons quatre grandes classes de parallélisme :

1. SISD : Single Instruction Single Data. C'est le cas le plus simple en l'absence de parallélisme pour un programme séquentiel tournant sur un cœur ;

5. weak scaling

2. SIMD : Single Instruction Multiple Data. Il s'agit ici d'un parallélisme de données où la même instruction est appliquée sur un ensemble de données. Par abus de langage [6], on désigne classiquement l'utilisation d'instructions SIMD sous le vocale de vectorisation où les opérandes de l'instruction sont des vecteurs de taille fixe (2 doubles [7] pour le SSE [8], 8 doubles pour l'AVX512 [9]). La figure 2.6 illustre une opération SIMD ;
3. SIMT : Single Instruction Multiple Threads. NVIDIA a introduit la notion de SIMT pour décrire le comportement de ses GPUs où une instruction est exécutée par un ensemble de threads synchrones ;
4. MISD : Multiple Instruction Single Data. Ce modèle correspond à un ordinateur de type pipeline où différentes opérations sont effectuées en flot sur une donnée. C'est le cas le plus rare ;
5. MIMD : Multiple Instruction Multiple Data. Cet acronyme décrit le fonctionnement d'un processeur multicœur qui exécute plusieurs threads en parallèle. Il est usuel de distinguer deux sous-catégories de ce modèle, très utilisées dans le monde du HPC
   - SPMD : Single Program Multiple Data. Un seul programme est dupliqué et exécuté sur un ensemble de ressources (cœurs et/ou nœuds) et appliqué sur des jeux de données différents. Ce sera le plus souvent un programme reposant sur la bibliothèque d'échanges de messages MPI utilisant tout ou partie d'un cluster ;
   - MPMD : Multiple Program Multiple Data. Plusieurs programmes différents vont coopérer et opérer sur des jeux de données différents. Rentrent dans cette catégorie les programmes de type maître-esclaves, où le programme maître va distribuer le travail aux différents esclaves, ou les co-processus de visualisation dans une des variantes de modèle de dépouillement *in-situ*. Dans ce dernier cas, le programme et son utilitaire de visualisation sont exécutés de manière concurrente et ils s'échangent des informations par différents moyens de communication (fichiers, mémoire partagée, messages MPI, ...).

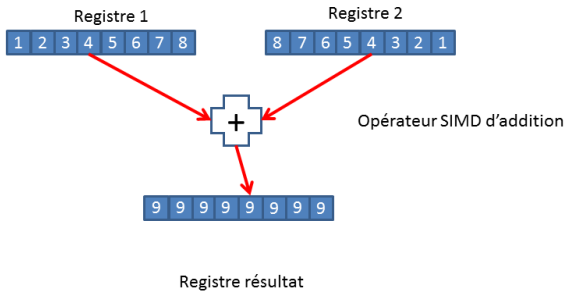


FIGURE 2.6 – Exemple d'addition SIMD de type AVX512 : le vecteur $(1, 2, 3, 4, 5, 6, 7, 8)$ est ajouté au vecteur $(8, 7, 6, 5, 4, 3, 2, 1)$ en une seule instruction, accélérant les calculs d'un facteur 8 potentiellement.

Sur le plan matériel, le parallélisme est mis en œuvre par de multiples options :

- le SIMD décrit plus haut ;
- le SMT (Simultaneous Multi Threading) ou hyperthreading, illustré sur la figure 2.7. Le cœur du processeur est capable de traiter plusieurs threads d'exécution de façon concurrente (en apparence). En pratique l'ordonnanceur (scheduler en anglais) va entrelacer les instructions de chaque thread pour maximiser l'occupation du pipeline d'exécution. Le SMT est surtout utile pour masquer les latences d'accès à la mémoire. Le SMT2 est le plus fréquent et permet un gain de performance d'environ 30 %. Le KNL d'Intel, le ThunderX2 de Marvell implémentent le SMT4. IBM propose les SMT1, SMT2, SMT4, SMT8 sur ses processeurs Power 8 [ibm] ;
- le multicœur tel que représenté sur la figure 2.9 (page 21). Pour les configurations les plus performantes nous pouvons trouver jusqu'à 68 cœurs comme dans le cas du KNL. La grande mul-

6. La CM2 implémentait un SIMD qui ne travaillait pas sur des vecteurs au sens SSE ou AVX.

7. Par abus de langage (dérivé de la syntaxe du langage C) un double est un nombre flottant représenté en double précision, soit sur 64 bits ; un simple est un flottant sur 32 bits. En HPC, la majorité des codes utilisent

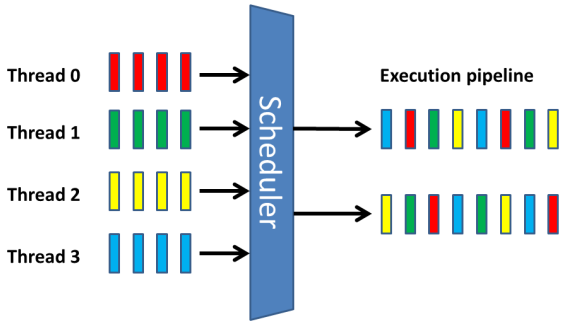


FIGURE 2.7 – Le Simultaneous Multi Threading. Ici est représenté du SMT4 décrivant le fait que le cœur du processeur est capable de traiter 4 threads d'exécution de façon concurrente (en apparence).

tiplicité de cœurs est une tendance de fond pour compenser la stagnation des fréquences utilisées. La figure 2.8 illustre l'évolution du nombre de cœurs par processeurs dans les machines du TOP500. Nous pouvons constater que les monocœurs, qui prédominaient avant les années 2005, ont complètement disparu ;

- la mise en grappe (cluster en anglais) de machines distinctes reliées entre elles par un réseau (rapide dans le cas du HPC ou du HPDA [10]) représenté sur figure 2.12 (page 24).

Un processeur moderne performant offre les trois premiers niveaux de parallélisme. Cela va impacter de façon importante la manière de développer les codes comme développé à la section 2.2.5.

2.2.3 Granularité du parallélisme

On appelle granularité du travail, la quantité de travail réalisée par chaque processeur. Si n_0 est le nombre d'opérations à réaliser alors la granularité est $\frac{n_0}{n_p}$ où n_p est le nombre de processeurs. On peut alors distinguer deux types de parallélisme : un parallélisme à grain fin et un parallélisme à gros grain.

- le parallélisme à grain fin correspond au cas n_p (très) grand. Nous sommes alors très proche du mode SIMD, d'autant plus proche que n_0 correspond à une instruction ;
- *a contrario*, le parallélisme à gros grain correspond à n_p faible vis à vis du nombre n_0 d'instructions et dans ce cas nous sommes dans le cas du modèle MIMD.

Appliqué dans le cas d'OpenMP, le style de programmation par annotation des boucles sera qualifié de programmation à grain fin et l'utilisation des tâches OpenMP sera, au contraire, une programmation à gros grain. Le premier style sera plus facile à mettre en place mais sera plus sensible à la loi d'Amdahl (nombreuses parties séquentielles entre les différentes boucles du programme). Le second sera moins sensible à la loi d'Amdahl mais sera plus difficile à mettre en place. Voir à ce titre la section 5.2.2.

---

la double précision pour la stabilité des calculs itératifs (limitation de la propagation des erreurs d'arrondis).

8. Streaming SIMD Extension
9. Advanced Vector eXtensions SIMD
10. Le HPDA ou High Performance Data Analytics

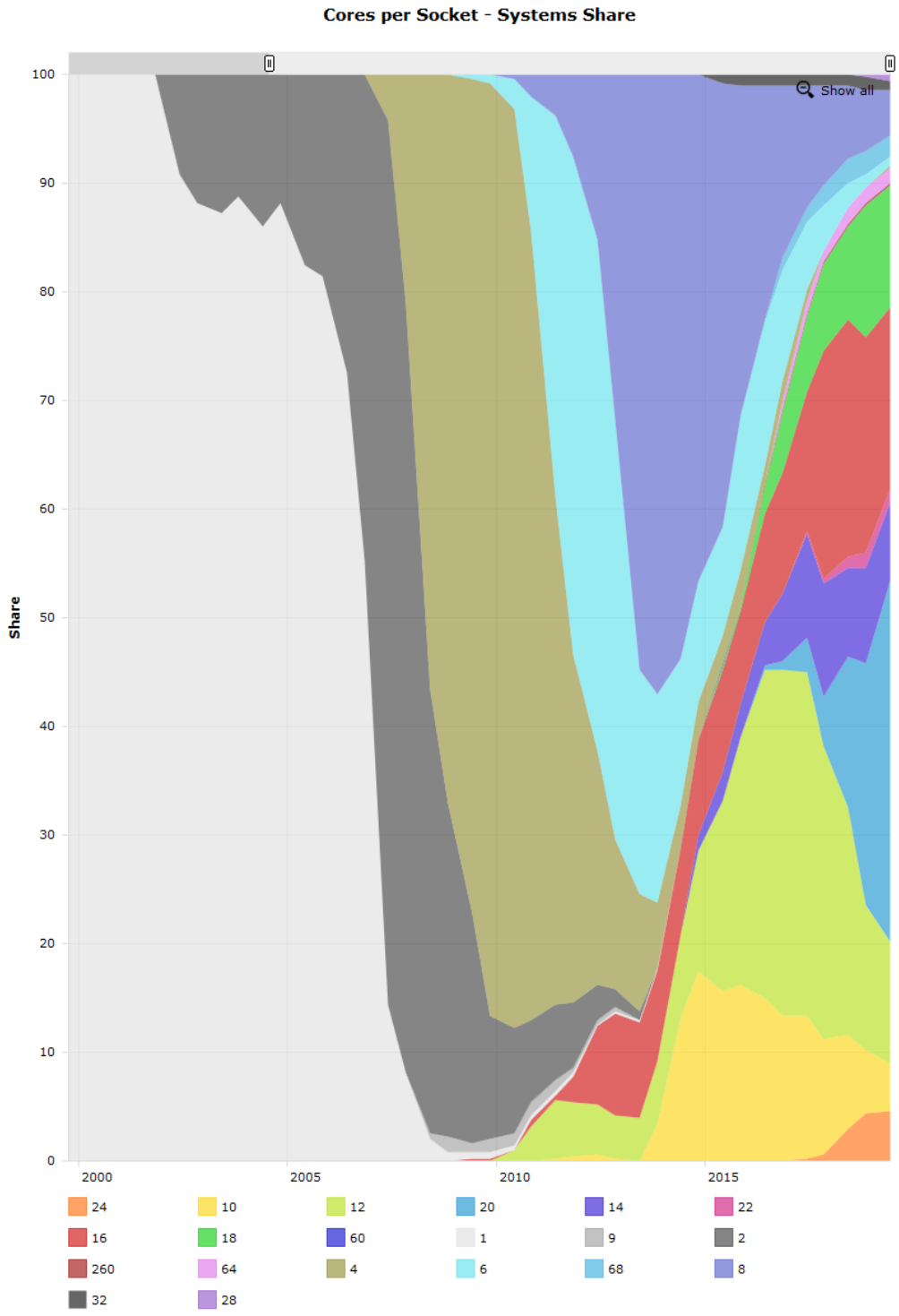


FIGURE 2.8 – Évolution du nombre de cœurs par processeur au cours du temps pour les machines du TOP500. A partir de 2010, il n'y a plus de machines monocœur au TOP500. ©TOP500.org

2.2.4 Modèles mémoire

On distingue deux grandes classes de modèles mémoire utilisées dans le monde du HPC et introduites par le parallélisme : la mémoire partagée et la mémoire distribuée.

2.2.4.1 PRAM

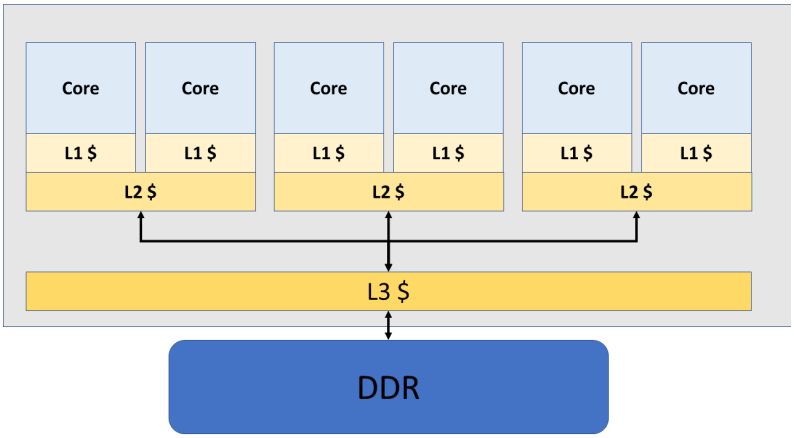


FIGURE 2.9 – Représentation schématique d'un processeur à 6 cœurs. Les différents niveaux de caches (L1$, L2$, L3$) sont gérés par le protocole MOESI. Ici les caches de niveau 1 (L1$) sont privés et les caches de niveau 2 et 3 sont partagés. Cette configuration varie d'une gamme de processeur à l'autre, le cache de niveau 3 (ou LLC pour « Last Level Cache ») pouvant ne pas exister comme dans le cas du KNC d'Intel.

Le modèle PRAM (pour **P**arallel **R**amdom **A**ccess **M**emory) décrit le fonctionnement des ordinateurs à mémoire partagée. C'est le modèle le plus courant car quasi omniprésent (serveurs, ordinateurs portables mais aussi smartphones) dès que la machine possède un ou plusieurs processeurs, à un ou plusieurs cœurs (figure 2.9). La communication entre les processus (ou threads [11]) se fera au travers de cette mémoire partagée. En HPC, ce sera notamment le cas lors de l'utilisation du multithreading offert via OpenMP [oped]. OpenMP est un langage de programmation, compatible C/C++ et FORTRAN, reposant sur des directives, qui permet de mettre en œuvre un parallélisme de thread au sein d'un programme. On distingue principalement trois modes de contrôle des accès mémoires pour les processus (et/ou threads) :

- EREW (Exclusive Read Exclusive Write) : une case mémoire ne peut être lue ou modifiée que par un seul processus à la fois ;
- CREW (Concurrent Read Exclusive Write) : une case mémoire peut être lue par plusieurs processus mais ne peut être écrite que par un seul processus à la fois ;
- CRCW (Concurrent Read Concurrent Write) : de multiple processus peuvent lire et écrire une case mémoire.

Les processeurs multicœurs modernes implémentent le modèle CREW au travers d'un protocole de cohérence de cache de type MESI ou MOESI qui maintient en permanence l'état d'une donnée [12] par des indicateurs qui peuvent être dans l'état (d'où le nom du protocole) :

- **M** : Modified. Cette donnée est la copie valide de la RAM et elle a été modifiée ;

11. Au sens UNIX, un **processus** est l'unité d'exécution manipulée par le système d'exploitation. Il dispose de son propre espace d'adressage et de son contexte d'exécution. Deux processus ne partagent pas leurs mémoires, sauf à utiliser une bibliothèque spécialisée et une zone de mémoire particulière. Un **thread** est un fil d'exécution au sein d'un processus UNIX. Il possède son propre compteur programme (PC) et une pile privée mais il partage l'espace d'adressage du processus avec d'éventuels autres threads du dit processus. Les threads d'un processus peuvent s'exécuter sur différents cœurs du processeur de façon concurrente. Transférer le contrôle de l'exécution d'un thread à l'autre est extrêmement rapide alors que le passage d'un processus à l'autre implique un changement de contexte qui se compte en milliers de cycles.

12. En fait une ligne de cache, voir plus bas

- **O** : Owned. Cette donnée est un des copies présentes dans le système de cache et elle peut être modifiée. La donnée devra être propagée aux autres caches en cas de modification ;
- **E** : Exclusive. Cette donnée est la seule présente dans le système de cache et elle est non modifiée ;
- **S** : Shared. Cette donnée est un des copies présentes dans le système de cache mais elle ne peut pas être modifiée ;
- **I** : Invalid ou inutilisé. Cette donnée doit être mise à jour par lecture de la mémoire principale.

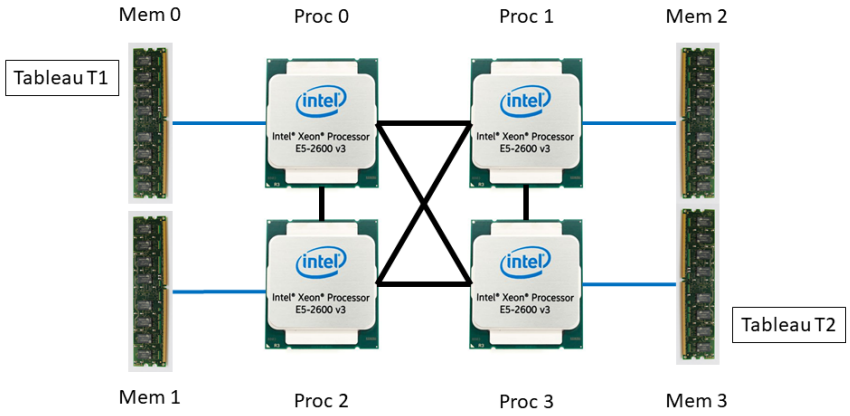


FIGURE 2.10 – Architecture NUMA d'un SMP : Le processeur 3 est à deux liens de distance du tableau T1, ce qui pénalise les performances.

Dans le cas particulier des serveurs possédant plusieurs processeurs, les accès à la mémoire principale peuvent être uniformes (modèle UMA - « Uniform Memory Access ») ou non uniforme (NUMA pour « Non Uniform Memory Access » – voir figure 2.10) avec la variante ccNUMA pour Cache Coherent Non Uniform Memory Access. Les serveurs actuels implémentent, en grande majorité, le modèle ccNUMA qui facilite la programmation des grands codes de calcul.

La notion de cache a une grande importance dans le fonctionnement des processeurs modernes. Comme le montre la figure 2.11, les cœurs de calcul n'ont pas accès directement à la mémoire. Au contraire, ils sont face à une hiérarchie des caches qui permettent de compenser les différences de performances entre vitesse du cœur et débit de la mémoire principale.

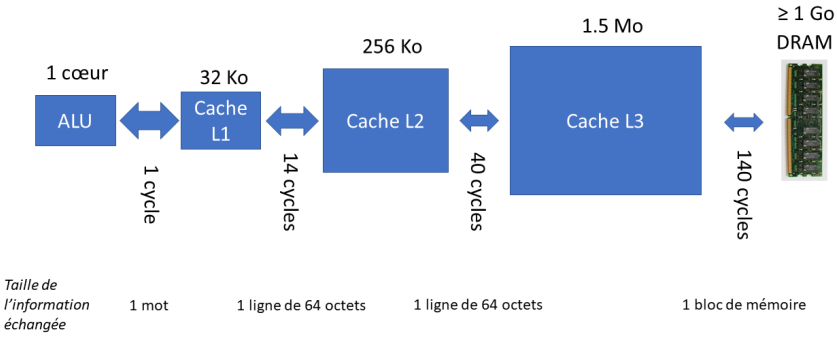


FIGURE 2.11 – Les caches : Un processeur moderne n'accède pas directement à la mémoire DRAM. le cœur va demander au cache de niveau 1 s'il possède l'information, si oui il y accédera en environ 1 cycle d'horloge. Sinon, il faudra faire monter l'information des niveaux inférieurs : c'est un défaut de cache qui prendra plus ou moins de temps à satisfaire selon l'endroit de la hiérarchie où se trouve l'information désirée. Dans cette hiérarchie, la vitesse d'accès est inversement proportionnelle à la capacité de stockage de l'information.

Une excellente description des mécanismes matériels impliqués dans les mécanismes de cache et d'exécution out-of-order est faite dans le livre de Hennessy et Patterson [Dav17]. Nous n'insisterons pas sur les différents mécanismes de caches (« Direct Mapped », « N-way Associative ») mais nous retiendrons le fait qu'un cache fonctionne par lecture ou écriture de blocs indivisibles des données, appelées lignes

de cache [13]. Pour remplir une ligne de cache, il faut donc lire un bloc indivisible de données depuis la mémoire. De même, la modification d'un octet d'une ligne de cache impose de remettre à jour tout le bloc de données correspondant dans la mémoire. Ce mécanisme est très adapté pour le traitement séquentiel de données (par exemple du traitement en « streaming »). Il repose sur l'*a priori* que la donnée suivante à traiter sera celle qui suit la donnée courante. Ce mécanisme est mis en défaut lorsque les programmes ne profitent pas astucieusement de celui-ci. On peut observer :

- du « cache thrashing » [14] où les caches sont constamment remis à jour sans être exploités (pas de localité temporelle des données). Cette situation se produit quand, par exemple, un programme lit de nombreux tableaux différents avant tout traitement ;
- du « false sharing » [15] qui apparaît lorsque deux threads essayent de travailler sur des sous-parties distinctes d'une même ligne de cache. Le mécanisme de cohérence de cache entre alors en jeu et remet à jour constamment les caches depuis la mémoire comme illustré par la table 2.1 ;
- de la perte de performance, si les données ne sont pas structurées de façon optimale. Si par exemple, le code ne manipule que le champ **x** de la structure suivante :

```
1 struct foo { int x; int y; };
```

le champ **y**, stocké immédiatement après **x**, sera obligatoirement remonté de la mémoire pour être mis dans la ligne de cache. On aura alors une mauvaise occupation du cache (une donnée sur deux est inutile) et un perte de bande passante mémoire (la moitié de la bande passante est consacrée à lire/écrire des données inutiles).

```
1  struct foo {
2      int x;
3      int y;
4  };
5  static struct foo f;
6
7  /* The two following functions are running concurrently: */
8
9  int sum_a(void)
10 {
11     int s = 0;
12     for (int i = 0; i < 1000000; ++i)
13         s += f.x;
14     return s;
15 }
16
17 void inc_b(void)
18 {
19     for (int i = 0; i < 1000000; ++i)
20         ++f.y;
21 }
```

TABLE 2.1 – Exemple de code produisant du faux partage de donnée (`https://en.wikipedia.org/wiki/False_sharing`). Si 2 threads exécutent l'un **sum_a** et l'autre **inc_b** en même temps sur des cœurs différents d'un même nœud, le mécanisme de cohérence de cache va chercher à relire constamment **f.x** depuis la mémoire car **f.y** aura été modifié par le deuxième thread. Il s'ensuivra une perte de performance significative du programme. Cette perte de performance est très difficile à diagnostiquer et supprimer.

13. Intel i7-4770 (Haswell), L1 Data cache = 32 KB, 64 octets/ligne, 8-way associative.
14. littéralement emballement
15. ou faux partage de données

2.2.4.2 DRAM

Le modèle DRAM (pour **D**istributed **R**amdom **A**ccess **M**emory) correspond au cas où la mémoire est physiquement distribuée dans des machines différentes (figure 2.12), donc dans des espaces d'adressages physiquement distincts. Les mouvements de données se font par échanges de messages reposant sur un réseau de communication (ou d'interconnexion) au moyen d'une bibliothèque d'échange de messages telle que MPI [mpi].

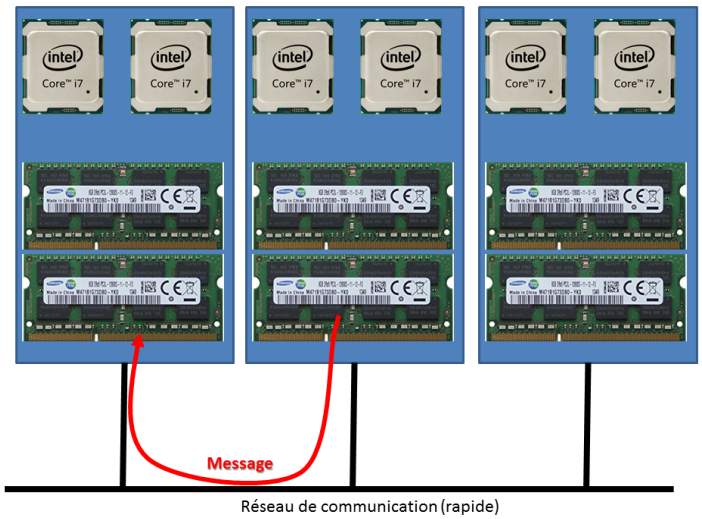


FIGURE 2.12 – Dans le modèle DRAM, la mémoire est physiquement répartie dans des machines distinctes reliées entre elles par un réseau rapide. Le passage d'informations d'une mémoire à l'autre se fait par échange de messages. L'agrégation de plusieurs machines pour n'en faire qu'une seule logique est appelé un cluster (ou grappe en français).

La topologie du réseau (anneau, arbre gras [fat tree], tore, ... – voir figure 2.13) va avoir une influence sur les performances de l'application et sur le choix de l'algorithme à utiliser. Par exemple, si les échanges de données ne se font qu'entre voisins immédiats (précédent - suivant, dans la topologie de l'application), une topologie en anneau est tout à fait adaptée. En revanche , ce même anneau sera très défavorable pour un algorithme où chaque tâche d'un programme communique avec toutes les autres (opération de type all-to-all).

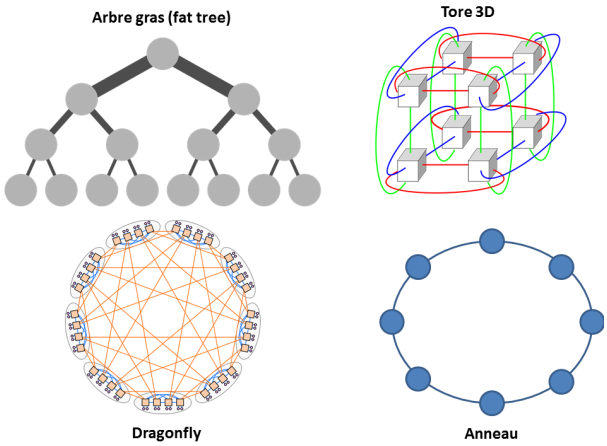


FIGURE 2.13 – Différentes topologies de réseau d'interconnexion d'ordinateurs. Les supercalculateurs du CEA/DAM privilégient la structure en arbre. La structure en anneau n'est pas utilisée pour réaliser des supercalculateurs. (Images Wikipedia).

2.2.5 Le modèle de parallélisme dans les codes

Les possibilités de la technologie actuelle ne permettent pas aux mémoires de gagner en latence même si les débits croissent, notamment avec l'arrivée des mémoires de type HBM. Pour compenser, les concepteurs de processeurs recourent de plus en plus au SMT [16] avec 2 ou 4 threads par cœurs. Cette technologie impose aux codes d'utiliser le multithreading (pthread, TBB [tbb] ou OpenMP [oped]) pour bénéficier d'une amélioration des performances. Par contre, un code MPI pur ne sera pas adapté à ce type d'architecture. La technologie pousse donc à la généralisation d'une programmation hybride de type MPI (en inter-nœud) + OpenMP (en intra-nœud).

Le modèle **SPMD** est le plus souvent utilisé dans les codes pour plusieurs raisons. Tout d'abord cela permet aux développeurs de n'avoir qu'un seul source à maintenir au lieu d'avoir plusieurs programmes indépendants (**MPMD**) qui s'exécutent séparément. De plus, surtout pour la CFD, les codes suivent le modèle BSP [Val90] (**B**ulk **S**ynchronous **P**arallel) dont la progression d'itération en itération est rythmée par le calcul du pas en temps (pour respecter la condition CFL), une opération de réduction sur l'ensemble des mailles de la géométrie calculée. La table 2.2 illustre l'enchaînement des phases d'un code de CFD, faisant ressortir sa conception de type BSP. Le modèle MPMD est à nouveau utilisé pour réaliser de la visualisation *in situ* par couplage faible du code et du logiciel de dépouillement.

Phase	Action	Communications
Initialisation	Lectures des données	Propagation des données aux différents rangs MPI
Début de la boucle de calcul		
	Traitement des bords des sous domaines	Échanges des mailles fantômes en décomposition de domaine
	Calculs de diverses grandeurs	Communications au besoin (par exemple en cas de gradient conjugué)
	Calcul du pas en temps	Réduction sur l'ensemble du maillage pour trouver le pas en temps minimal. Fin de la super étape.
Fin de la boucle de calcul		
Fin du calcul	Sortie des résultats	

TABLE 2.2 – Structure générale d'un code de CFD implémentant le modèle BSP. Le calcul du pas de temps matérialise la fin de la super étape (superstep) définie par le modèle BSP de Valiant.

Pour les simulations qui utilisent un très grand nombre de tâches MPI, la méthode numérique choisie déterminera le coût en communications sur le réseau. Si les méthodes explicites n'imposent que les échanges de mailles fantômes, les méthodes de type implicite reposant sur la résolution de grands systèmes linéaires, souvent par des méthodes itératives comme celle utilisée dans le HPCG (voir la section 3.5.4 page 52), vont être très dépendantes des performances de la bibliothèque d'échange de messages. Le choix de la technologie réseau sera un critère important à prendre en compte dans la conception d'un supercalculateur.

Que ce soit pour la programmation MPI ou OpenMP, la connaissance du placement des cœurs au sein du processeur (sa structure interne) aura une influence sur le placement des tâches MPI et des threads. En général, le système de soumission des exécutables va prendre en charge le placement des tâches MPI et/ou des threads pour un placement optimum. Néanmoins l'utilisateur peut agir pour avoir le placement le plus adapté à son code. Par exemple il peut décider de ne pas voir les threads migrer au gré du bon vouloir du système (`OMP_PROC_BIND=true`) ou être plus précis et vouloir que les threads soient répartis

16. **S**imultaneous **M**ulti**t**hreading qui consiste à augmenter le parallélisme de threads.

sur tous les cœurs (`OMP_PROC_BIND=spread`).

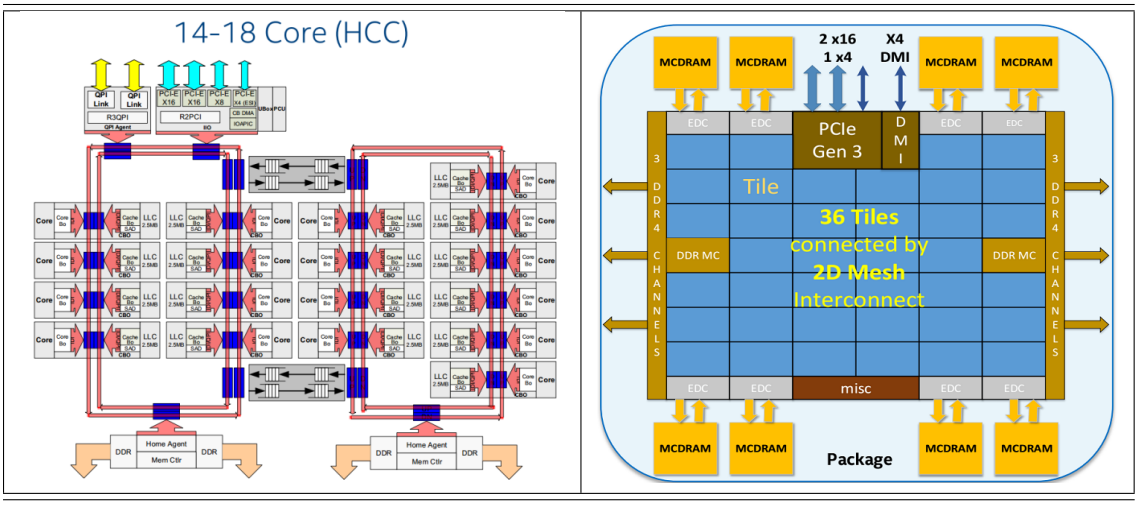


FIGURE 2.14 – Schéma de positionnement des cœurs au sein d'un processeur Haswell d'Intel (à gauche) et d'un KNL (à droite). ©Intel

La figure 2.14 illustre bien le problème. Un Haswell à 18 cœurs, comme illustré ici, va pousser à utiliser 2 tâches MPI par processeur (un par anneau) et utiliser 8 threads par processus MPI, quitte à n'utiliser que 16 cœurs au final sous peine de déséquilibre. On utilisera alors un placement des threads OpenMP du type `OMP_PLACES=cores OMP_PROC_BIND=close`. *A contrario*, un KNL demandera 4 tâches MPI par processeur pour optimiser les accès à la MCDRAM (1 MPI par quadrant) et 9 threads (ou 18 ou 36 car le KNL supporte le SMT4) par tâche MPI. Seule la connaissance intime du matériel permet donc de l'utiliser correctement.

Pour aller vers une optimisation extrêmement poussée, souvent réservée à des spécialistes, il peut être nécessaire de connaître en détail la manière dont fonctionne un cœur de processeur. La figure 2.15 donne une description logique de la façon dont est organisé un cœur de processeur de type Haswell d'Intel. Un outil comme MAQAO [maq] va utiliser ces informations pour émettre des diagnostics sur le comportement d'un code. Néanmoins, pour la majorité des développeurs, il suffira de se reposer sur le compilateur qui, dans la majorité des cas, produira un code de qualité pratiquement égale à ce que l'on pourrait faire manuellement en assembleur.

Enfin, la connaissance des données constructeur, comme la taille des tampons de réorganisation des données (ROB), va permettre de comparer différentes architectures entre elles et d'évaluer les gains potentiels : par exemple un cœur Silvermont, tel qu'utilisé dans le KNL, possède 72 entrées ROB alors que le Haswell en possède 192, dénotant la meilleure capacité du processeur Haswell à traiter les instructions dans le désordre (Out-of-Order).

2.2.6 Le modèle mémoire des codes

Le modèle **PRAM** est celui sur lequel se repose OpenMP. Le programmeur doit avoir conscience de ce modèle, en particulier lors de l'utilisation de variables globales utilisées par plusieurs threads en lecture/écriture, quitte à devoir utiliser la directive `#pragma omp flush (var)` pour forcer la cohérence de la vision d'une donnée par les différents cœurs/processeurs du nœud. Il doit aussi avoir une idée claire du fonctionnement des hiérarchies de cache pour que ses développements n'aillent pas à l'encontre de ces mécanismes.

Le modèle **DRAM** est le modèle de référence pour MPI. Chaque processus MPI possède son espace d'adressage propre sur chacune des machines concernées. Ces processus ne seront pas capables de partager

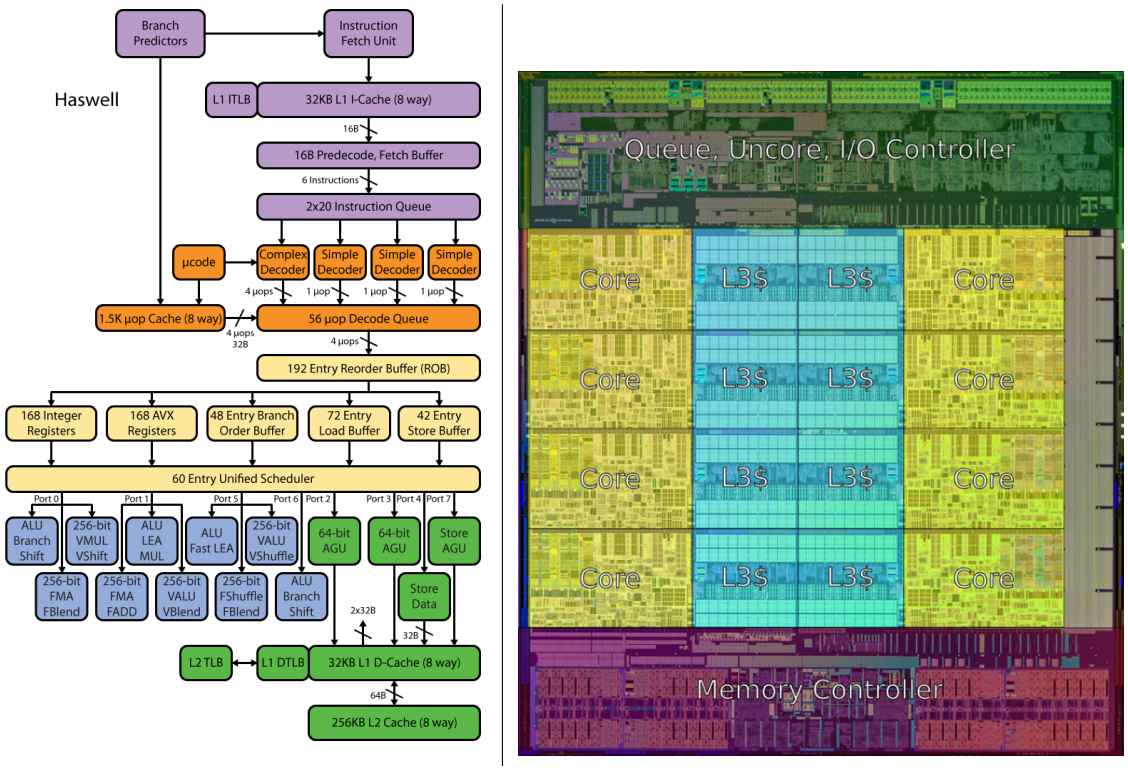


FIGURE 2.15 – A gauche : schéma logique d'un cœur d'un processeur Haswell d'Intel (appelé la microarchitecture du processeur). Ce schéma ne préjuge en aucune façon de la disposition physique des circuits dans le cœur. A droite : photographie annotée d'un processeur Haswell à 8 cœurs. Le schéma de gauche est donc répliqué dans chacune des cases jaunes de droite dénommée « core ». ©Intel.

des informations via la mémoire physique [17]. Or, les simulations de grandes tailles (nombre de mailles élevé) bien souvent ne peuvent rentrer en mémoire d'un seul nœud. Il est donc nécessaire de découper le domaine à traiter et de le répartir sur plusieurs nœuds. On parle alors d'algorithme à décomposition de domaine. Pour assurer la continuité numérique dans ce cas, il est nécessaire d'introduire des zones de mailles tampons (le plus souvent appelées fantômes), d'épaisseur variable selon l'ordre du schéma, comme illustré sur la figure 2.16. Cette zone de recouvrement permet de traiter correctement les mailles du bord des sous-domaines en récupérant les valeurs à jour dans le domaine voisin (les zones vertes du schéma) et en les utilisant localement (zone rouges du schéma). Ce faisant, le découpage en sous-domaine n'impacte pas le résultat de la simulation [18]. Chaque sous-domaine sera pris en charge par un processus MPI.

L'approche MPI n'est pas la seule à reposer sur le modèle DRAM. Les langages de type PGAS (**P**artitioned **G**lobal **A**ddress **S**pace) tels que Chapel, UPC ou X10, prennent en charge la description et les mouvements des données au sein du langage lui-même. Le programmeur n'a plus à faire d'appels explicites à une bibliothèque telle que MPI. L'approche PGAS n'est pas utilisée dans les codes du CEA/DAM.

Pour les portions de code s'exécutant sur un nœud (le plus souvent un rang [19] MPI) qui reposent sur OpenMP (modèle de développement hybride MPI + OpenMP), le modèle **ccNUMA** impose des contraintes particulières. Outre le false sharing, déjà évoqué plus haut, le placement des données va prendre une importance particulière et nécessiter quelques attentions. Comme illustré sur la figure 2.10,

17. Sauf utiliser le mécanisme de fenêtres proposé par MPI-3 mais rarement utilisé en production.
18. Cette affirmation n'est vraie qu'en première approximation. L'ordre des opérations intervient et peut conduire à des variations numériques entre les versions séquentielle et parallèle. Ces variations apparaissent notamment à cause de la bibliothèque d'échange de messages qui ne garantit pas forcément l'ordre des opérations.
19. Les processus MPI sont numérotés de manière unique. Dans la nomenclature MPI, ce numéro est appelé un rang (MPI rank en anglais). Donc, par extension, on désigne un processus de l'ensemble par la dénomination « rang MPI ».

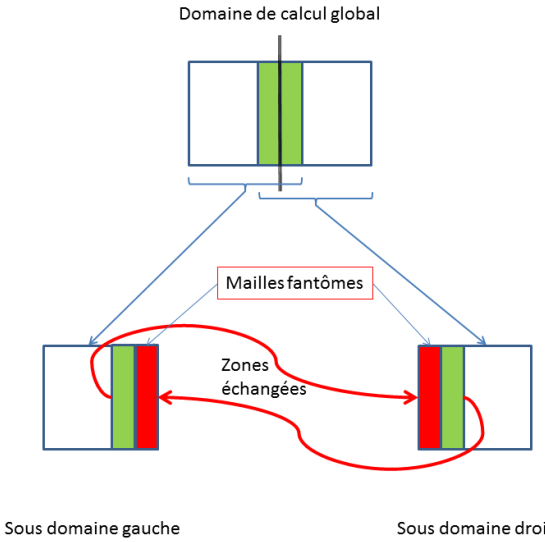


FIGURE 2.16 – Décomposition de domaines et mailles fantômes. Chaque sous-domaine est usuellement affecté à un processus MPI (mais cela peut aussi être un thread).

la localisation d'un tableau aura un impact direct sur les performances d'accès selon le processeur qui aura besoin de la donnée. Le programmeur devra faire attention à ce que le thread qui crée et initialise les tableaux soit celui qui l'utilise ensuite. Sinon, une pénalité d'un facteur 2 est classiquement observée sur les accès aux données. Il devra aussi prendre en compte l'optimisation classique de Linux dite de la « first touch policy » où un tableau créé par `malloc()` n'est en fait effectivement créé en mémoire que si une donnée est écrite dedans. L'écueil classique est qu'un thread 1 alloue un tableau sur un processeur et qu'un autre thread 2, sur un autre processeur, remplisse le tableau le premier. Le tableau sera alors affecté à la mémoire du second processeur, dégradant mécaniquement les performances d'accès observées par le premier processeur (thread 1).

– ♦ ∞ ♦ ∞ ♦ ∞ ♦ ∞ ♦ ∞ ♦ ∞ ♦ –

3 Les supercalculateurs du CEA

3.1 Les machines vectorielles

Les ordinateurs vectoriels créés par Seymour Cray à la fin des années 1970 eurent un impact majeur sur les capacités de calcul des grands centres scientifiques. Ils étaient clairement les machines les plus puissantes de l'époque du fait de caractéristiques uniques. A cette époque, les codes étaient principalement des codes structurés, codés en FORTRAN. Sur les dernières générations, le PASCAL et le C étaient disponibles. La taille mémoire disponible limitait fortement le nombre de mailles des simulations et obligeait à imaginer des algorithmes de pagination applicative (dit « out-of-core ») qui, heureusement, ne sont plus pratiqués aujourd'hui.

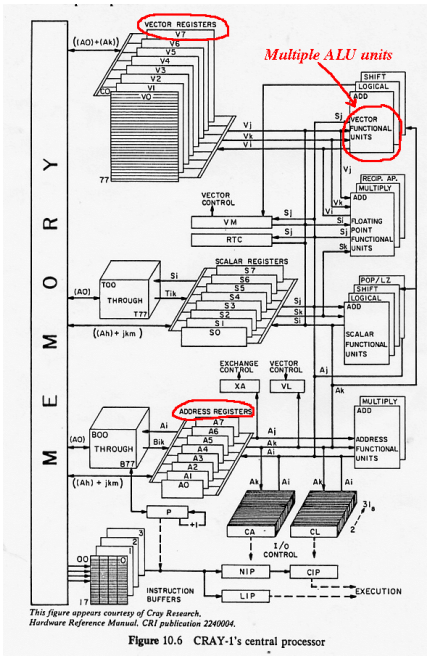


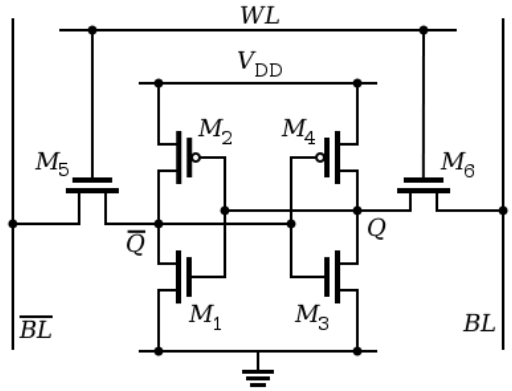
FIGURE 3.1 – Diagramme décrivant la structure d'un processeurs du CRAY 1S. ©Cray

La figure 3.1 donne un aperçu de l'agencement du processeur du CRAY 1S. On y trouve tous les éléments qui feront la force des supercalculateurs vectoriels CRAY :

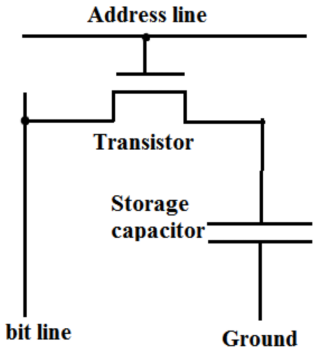
- un système mémoire très performant car utilisant de la mémoire de type SRAM (static random access memory) dont le principe est rappelé dans la figure 3.2. Ce type de mémoire est très cher et a été remplacé par la DRAM (dynamic random access memory) dans les ordinateurs modernes pour le stockage de masse. Par contre à l'intérieur des microprocesseurs, les registres et la mémoire cache de niveau 1 sont toujours réalisés en SRAM ;
- une absence de caches entre la mémoire principale et les unités fonctionnelles du processeur. L'apparition des caches se fera sur les clusters de SMP, abordés à la section 3.2 ;
- des unités vectorielles capables de traiter 64 éléments à la fois, passé une phase d'amorçage du

pipeline ;

- et surtout, un équilibre entre débit mémoire et puissance de traitement qui permettait d'alimenter en continu les unités vectorielles. Cette caractéristique a disparu depuis l'avènement des microprocesseurs et les constructeurs actuels essayent de la retrouver comme ce sera expliqué en fin de manuscrit à la section 9.1.2.



©Wikipedia. Une cellule (1 bit) SRAM est composée de 4 à 6 transistors (ici 6). Les transistors servent à mémoriser un état 1 ou 0 en bascule. Les temps d'accès d'une telle cellule sont très courts. Par contre une cellule SRAM est volumineuse et donc limitera les volumes stockés par unité de surface.



©Wikipedia. Une cellule DRAM est composée d'un transistor et d'une capacité. La capacité sert au stockage de l'information (1 pleine, 0 vide). Une DRAM nécessite d'être rafraîchie régulièrement car la capacité se décharge naturellement au cours du temps. Cette opération impactera les performances (débit et latence) de la mémoire. L'avantage majeure de la DRAM est sa compacité qui autorise à stocker de grands volumes de données par unité de surface.

FIGURE 3.2 – Comparaison de l'implémentation des mémoires SRAM et DRAM.

3.1.1 Caractéristiques

La table 3.1 résumé les caractéristiques des diverses machines Cray de production du CEA/DAM. Il est intéressant de comparer les performances de ces machines à celle d'un smartphone de 2018 (pour un Samsung Galaxy S7 : CPU Exynos 8890 à 8 cœurs, 2.6 GHz, 4 Go de RAM, 1174 MFlops linpack), sans parler des volumes des appareils ni de leurs consommations électriques.

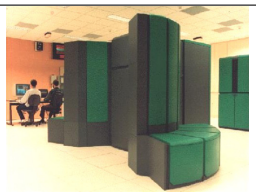
	Cray 1S	Cray XMP	Cray YMP	Cray T90
Mise en service	1982	1987	1990	1996
Nombre de CPU	1	4	8	24
Taille mémoire	16 Mo	64 Mo	1 Go	4 Go
Fréquence	83 MHz	117 MHz	167 MHz	500 MHz
Performance	166 Mflops	0.96 Gflops	2.7 Gflops	43.2 Gflops
				

TABLE 3.1 – Caractéristiques principales des ordinateurs CRAY de production du CEA/DAM.

La relative simplicité de l'architecture de ces machines s'est ressentie sur les codes. Leur structuration était simple car ils avaient une vision plate de la mémoire, au « banking » près. En effet les machines

Cray avaient une particularité que l'on retrouvera dans les GPUs de NVIDIA : la mémoire était découpée en banc permettant de paralléliser les accès mémoires contigus. Dans l'exemple de la figure 3.3 la mémoire est répartie en 4 bancs. La machine est capable de lire en une seule transaction mémoire les 4 premiers éléments du tableau puis les 4 suivants et ainsi de suite. Par contre, si le programme a besoin d'accéder aux éléments 1 et 5 dans notre exemple, il se produira un conflit de banc qui se traduira par une attente : il faut attendre que l'élément 1 soit lu pour que l'on puisse lire l'élément 5. Le contournement classique consistera (dans le cas des tableaux à plusieurs dimensions) d'introduire une colonne de plus, inutile pour l'algorithme, mais qui provoquera un décalage en mémoire faisant sauter le conflit. C'est la technique dite du « padding » ou rembourrage.

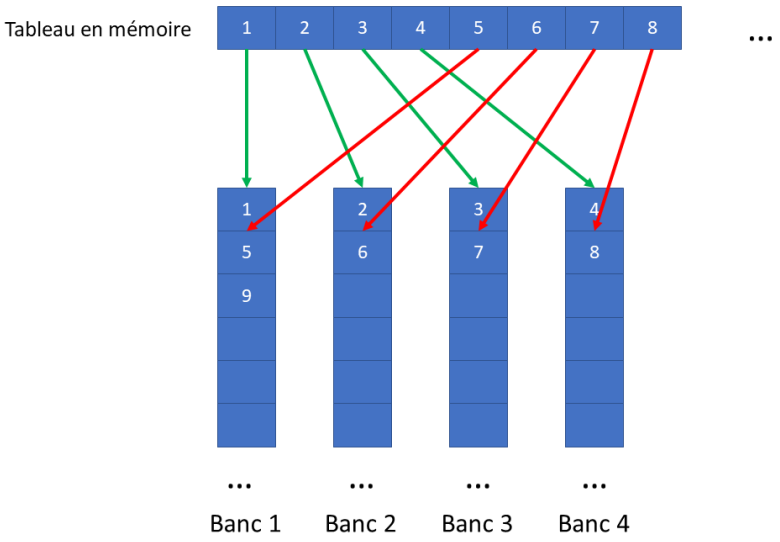


FIGURE 3.3 – Illustration de la répartition des éléments d'un tableau sur un système à 4 bancs mémoire. La lecture simultanée des éléments 1 et 5 n'est pas possible car ces deux éléments sont stockés dans le même banc. Il faudra deux lectures successives pour récupérer ces informations. Par contre, il est possible de récupérer les éléments 1 à 4 en un seul appel à la mémoire.

La table 3.2 illustre cette technique qui, si elle fait perdre de la mémoire, permet de conserver des accès en mémoire efficaces lors des parcours de tableaux en colonne. Le dilemme du développeur de code sera de choisir la bonne structuration de ses données si les tableaux sont parcourus en ligne ET en colonne. Une réorganisation temporaire peut alors être utile, même si elle a un coût.

banc 1	banc 2	banc 3	banc 4
tab(1,1)	tab(1,2)	tab(1,3)	tab(1,4)
tab(1,5)	tab(2,1)	tab(2,2)	tab(2,3)
tab(2,4)	**tab(2,5)**	tab(3,1)	tab(3,2)
tab(3,3)	tab(3,4)	**tab(3,5)**	

TABLE 3.2 – Introduction de padding dans un tableau 4x3 pour éviter les conflits de banc sur un accès par colonne. Les éléments en rouges (padding) ne seront pas utilisés lors des calculs. Tab(1,1) et tab(2,1) pourront être lus en mémoire en une seule opération.

La vision plate de la mémoire, l'absence de caches et la vectorisation ont eu comme conséquence que les codes pouvaient balayer l'intégralité du domaine de calcul à chacune des boucles, pratique qu'il fallu remettre en cause plus tard pour rester dans les caches.

L'effort principal de codage s'est porté sur la vectorisation. Les algorithmes ont été écrits de manière à :

- optimiser les accès mémoire comme nous venons de le voir ;
- s'assurer de travailler sur des indices consécutifs pour pouvoir traiter les données par paquets de

8 éléments (mots de 64 bits) ;
- d'éviter les branchements à l'intérieur des boucles. Cette préoccupation restera d'actualité avec les GPUs. Pour contourner cette difficulté, l'assembleur CRAY disposait d'instructions de masque accessibles depuis FORTRAN telle que $\vec{z} = CVMGT(\vec{x}, \vec{y}, \vec{p})$ où le vecteur $\vec{z} = \vec{x}$ si $\vec{p}$ est vrai et $\vec{z} = \vec{y}$ sinon. En jouant sur ces opérateurs, il était possible d'éviter tout branchement et donc d'assurer un traitement complètement vectoriel des boucles, surtout dans le cas où le compilateur ne disposait pas d'une vectorisation automatique poussée (CFT [1]).

Il est à noter que les machines CRAY, y compris le T90, étaient utilisées par des logiciels non parallélisés, même si la machine disposait de plus d'un CPU [2]. Les 24 CPUs du T90 permettaient de faire tourner 24 programmes de front. Seules quelques expérimentations de parallélisme ont été menées sur ces machines ; les vrais développements parallèles ont été conduits sur T3D et T3E qui seront décrits plus loin à la section 3.3.1.

Les machines CRAY ont clairement démontré que la performance maximale d'un programme ne peut être atteinte que si l'on a une connaissance fine du matériel sous-jacent. Cela peut nécessiter l'utilisation de fonctions intrinsèques comme le `CVMGT` dans les algorithmes. Heureusement, les compilateurs évoluant (CFT77), l'écriture naturelle utilisant des conditionnelles peut être (plus ou moins) automatiquement convertie en masques.

Des machines CRAY nous retiendrons donc que la vectorisation était nécessaire pour atteindre la meilleure performance et que la vision plate de la mémoire a influencé la façon de parcourir les données.

3.2 Les clusters de SMP

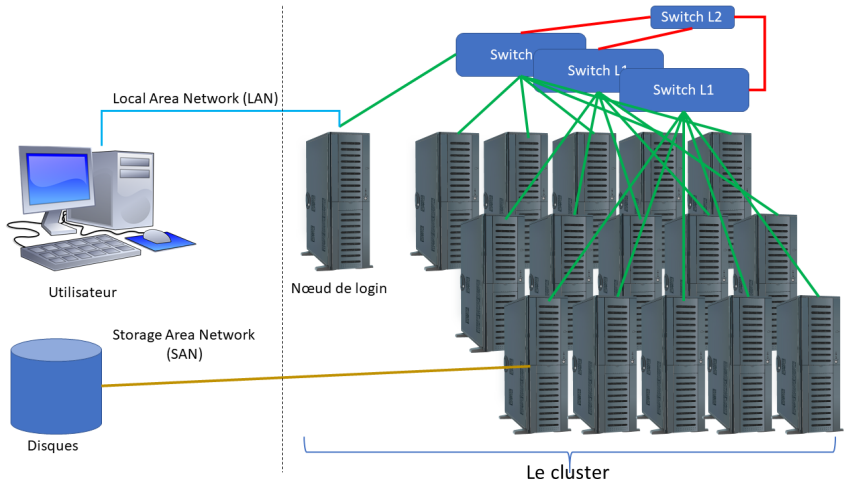


FIGURE 3.4 – Un cluster dans son environnement. dans son environnement. Les traits verts et rouges matérialisent le réseau d'interconnexion rapide. Diverses topologies de réseau sont possibles. Les machines du CEA/DAM ont, pour l'instant, toujours privilégié des topologies en arbre élagué (≪ pruned fat tree ≫).

Les années 2000 ont marqué un tournant dans l'architecture des ordinateurs de production. Le CRAY T90 était le symbole d'une technologie très chère, poussée à son extrême. Les machines à base de composants du commerce de grande diffusion [3] avaient le potentiel d'offrir des performances crêtes bien supérieures pour un coût moindre. La machine Tera 1 a démontré que ce pari était gagnant, ouvrant ainsi la voie à une série de supercalculateurs toujours plus efficaces.

1. Cray Fortran
2. CPU : Central Processing Unit ou processeur en français.
3. classiquement dénommé COTS pour ≪ Component Of The Shelf ≫ – composants sur étagère

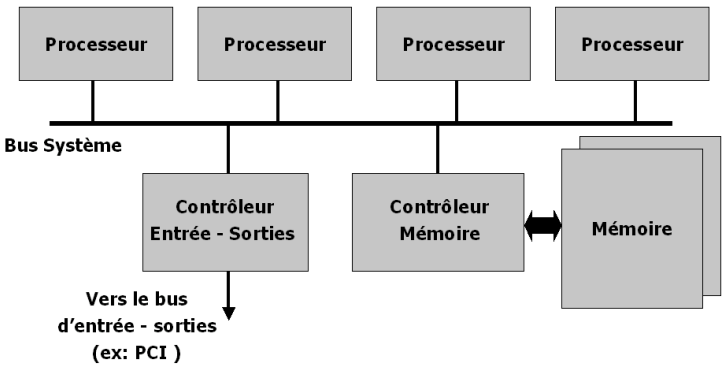


FIGURE 3.5 – Architecture d'un SMP

Cette section va donc être consacrée à ce type de supercalculateurs qui entrent dans la catégorie des clusters de SMP, rompant avec l'aspect monolithique des ordinateurs CRAY.

Un cluster est une agrégation d'un ensemble de machines, reliées ensemble par un réseau comme illustré sur la figure 3.4. Les premiers clusters de type Beowulf [Bec+95] étaient composés de simples PC reliés entre eux par des liens Ethernet. Néanmoins, pour réaliser un supercalculateur, des réseaux d'interconnexion (liens verts et rouges de la figure 3.4) de hautes performances sont nécessaires. La performance d'un réseau se mesure en débit (volume de message transmis par seconde) [4] et en latence (nombre de μs pour initier l'envoi d'un message). Le réseau prend toute son importance dans les schémas numériques implicites qui nécessitent de résoudre des systèmes linéaires répartis sur l'ensemble des nœuds. La résolution de ces grands systèmes est si commune dans les codes scientifiques que les méthodes associées ont conduit à la création de benchmarks qui seront décrits au chapitre 3.5.

Un SMP (pour Symmetric Multi Processor) est une machine où tous les éléments de calcul ont une vision unifiée de la mémoire au sein d'une seule instance du système d'exploitation (figure 3.5). Cette vision théorique est en pratique le plus souvent implémentée sous la forme ccNUMA (cache coherent Non Uniform Memory Access) où chaque processeur n'aura pas le même temps d'accès à la mémoire comme illustré par la figure 2.10 de la page 22. Le tableau T1, alloué sur la mémoire 0 liée au processeur 0, sera accédé par ce dernier bien plus vite que par le processeur 3. Le placement des données en mémoire devient un réel enjeu pour la performance globale du programme. Ce sera tout particulièrement vrai pour les programmes utilisant le standard OpenMP pour un parallélisme de threads : ceux-ci peuvent se répartir sur l'ensemble des processeurs et sans vigilance peuvent utiliser des données placées aux mauvais endroits à leurs créations.

La table 3.3 résume les caractéristiques des grands clusters de production du CEA/DAM. La comparaison avec la table 3.1 montre le saut important de performance crête obtenu au passage à l'architecture cluster.

3.2.1 Tera 1-10-100

Le supercalculateur Tera 1 est le premier supercalculateur de production reposant sur des composants de grande diffusion (COTS) au CEA/DAM. Cette nouvelle architecture a eu un impact important sur les codes car il a été nécessaire de prendre en compte le parallélisme à mémoire distribuée qui était à la base de la performance globale de la machine. La programmation par décomposition de domaine a donc fait son apparition en s'appuyant sur la bibliothèque d'échange de message MPI [mpi]. Cela a été la première rencontre avec la loi d'Amdahl lors de la mise en place de ce type de parallélisme.

4. le nombre de messages émis par seconde est aussi une caractéristique importante.

	Tera 1	Tera 10	Tera 100
Mise en service	2002	2006	2011
Constructeur	Compaq/HP	BULL	BULL
Processeur	DEC Alpha EV68	Intel Itanium 2	Intel Nehalem
cœurs / nœud	4	16	32
Nombre de cœurs	2560	10000	140000
Nombre de nœuds	640	624	4370
Taille mémoire/cœur	1 Go	3 Go	2 Go
Fréquence	1 GHz	1.5 GHz	2.27 GHz
Performance	5 Tflops	52 Tflops	1,25 Pflops
Consommation	0.6 MW	1.5 MW	5 MW
Réseau rapide	Quadrics Elan 3	Quadrics Elan 4	Infiniband QDR
			

TABLE 3.3 – Caractéristiques principales des ordinateurs Tera 1, Tera 10 et Tera 100.

Contrairement au T3D (voir section 3.3.1) le processeur Alpha EV68 de Tera 1 est un RISC [5] ayant une microarchitecture [6] « out-of-order » (OoO). L'OoO apporte une souplesse significative au compilateur qui n'a plus à faire attention au séquencement des opérations car celui-ci sera pris en charge par le processeur de manière automatique, comme illustré par la table 3.4. L'utilisateur voit naturellement augmenter les performances de son programme grâce à ce mécanisme, qui en contre-partie complique notablement la conception du processeur (sa microarchitecture).

Processeur in order			Processeur out-of-order		
1	*move $3, 100($4)*	en exécution, provoque un défaut de cache sur $4	1	*move $3, 100($4)*	en exécution, provoque un défaut de cache sur $4
2	*add $2, $3, $4*	en attente que le défaut de cache soit résolu	2	*sub $5, $6, $7*	peut s'exécuter pendant le défaut de cache
3	*sub $5, $6, $7*	attends que l'instruction **add** soit terminée	3	*add $2, $3, $4*	en attente que le défaut de cache soit résolu

TABLE 3.4 – Comparaison processeur in-order (exécution dans l'ordre) et out-of-order (exécution dans le désordre) : dans le cas in-order, la ligne 3 ne peut être exécutée qu'après le 2, qui elle-même est en attente de la ligne 1. Dans le cas out-of-order, la ligne 2 est indépendante de la 1 et de la 3 donc elle peut être effectuée immédiatement après la ligne 1 sans attendre que la référence à $4 soit satisfaite. Si la référence à $4 n'est pas trop loin dans la hiérarchie de cache, la ligne 3 pourra s'exécuter rapidement après la ligne 2. Charge ensuite au processeur OoO de remettre les résultats dans l'ordre attendu par le programme (respect de sa sémantique).

Tera 1 est la première machine de production dont le processeur met en jeu le mécanisme de cache. Contrairement aux CRAY et à leurs visions plates et uniformes de la mémoire, les machines à base de microprocesseurs ont une architecture hiérarchique de la mémoire qui essaye de compenser la différence de vitesse qui existe entre les circuits de calcul et la vitesse des mémoires. Pour remédier à ce déséquilibre, les concepteurs de circuits ont introduit la notion de cache décrite par la figure 2.11 (page 22).

L'apparition des caches a eu un impact direct sur les codes : le balayage complet des tableaux de grandes dimensions rendait inopérant les mécanismes de cache. Il est donc nécessaire de prendre en compte cette

5. Reduced Instruction Set : jeu d'instruction réduit

6. Dans le monde des processeurs, on appelle **architecture** le jeu d'instruction – le langage machine – par lequel utiliser le processeur. On appelle **microarchitecture** la façon dont est implémenté le jeu d'instruction au sein du processeur, à l'aide de transistors.

contrainte dans la programmation et de ne travailler que sur des portions de tableau (localité spatiale), le plus longtemps possible (localité temporelle) pour utiliser à plein les caches de plus petit niveau (idéalement L1$, le plus classiquement L2$ ou L3$). Cela a conduit à la création de méthodes telles que le blocking, étudié par exemple par Lam, Rothberg et Wolf [LRW91]. De plus, comme illustré sur la figure 2.11, les caches ne sont pas remplis avec une granularité de l'octet mais par blocs de 64 octets (un bloc est appelé une ligne de cache). Cela impactera directement la structuration des codes comme cela sera détaillé dans la section 3.3.3.1 (page 45) sur la vectorisation SIMD.

La conception de Tera 10 a été réalisée en partenariat avec la société Bull (depuis rachetée par la société Atos). Aujourd'hui nous parlerions de « codesign ». Elle a repris les éléments positifs de Tera 1 et a innové par l'utilisation du processeur Itanium.

L'itanium est un processeur 64 bits développé par Intel en coopération avec différents constructeurs informatiques (HP, Bull, etc.), visant initialement à remplacer l'architecture x86/CISC. L'architecture Itanium, nommée IA-64, repose sur la technologie EPIC (Explicitly Parallel Instruction Computing), considérée comme le successeur du RISC. Le challenge de cette architecture repose sur la capacité du compilateur à extraire du parallélisme d'instruction du programme, performance qui n'a pas été au rendez-vous à l'arrivée de la machine.

Tera 10 a vu l'abandon des systèmes d'exploitation propriétaires (ceux de CRAY et d'HP) au profit de l'open source. Nous avons donc vu arriver Linux (fourni par la société Red Hat) et du système de fichier parallèle Lustre.

Tera 100, dans la lignée de Tera 1 et 10, est la première machine pétaflopique conçue et fabriquée en Europe. Une description complète de la machine a été publiée dans la référence [Ami+13] et explique notamment sa méthode de refroidissement originale. Nous retiendrons de cette machine son utilisation d'un processeur très courant (utilisé dans les stations de travail), le Nehalem d'Intel, ainsi que sa forte consommation électrique, comparée aux ordinateurs de générations précédentes.

La taille mémoire impressionnante de cette machine a permis de développer des codes toujours plus performants et conduire des simulations toujours plus précises.

3.2.2 Tera 1000

Tera 1000, dont les principales caractéristiques sont données dans la table 3.5, représente la limite des technologies classiques et marque un point de bascule vers les technologies du futur. En effet la consommation électrique devient un sujet de préoccupation majeure et elle sera financièrement impossible à soutenir pour une machine de taille exaflopique, à technologie constante. Cette discussion sera abordée en détail à la section 9.1 sur l'exascale.

La machine Tera 1000 est composée de deux partitions, l'une de taille intermédiaire (Tera 1000-1) pour assurer la compatibilité avec le passé, dans la ligne droite de Tera 100, et l'autre (Tera 1000-2) plus résolument tournée vers l'avenir mettant en avant l'efficacité énergétique de son processeur, qui sera décrit dans la section sur les manycores (section 3.3.3).

Avec Tera 1000-2, on atteint la limite en programmabilité en ne se reposant que sur la programmation par échange de message MPI. Il devient impératif d'utiliser au maximum les trois niveaux de parallélismes proposés par cette génération de supercalculateur : parallélisation à mémoire distribuée (MPI), parallélisme à mémoire partagée reposant sur les threads (OpenMP ou TBB) et le parallélisme de données (ou vectorisation) SIMD.

Les instructions vectorielles SIMD sont enfin pleinement exploitées que ce soit l'AXV2 sur Tera 1000-1

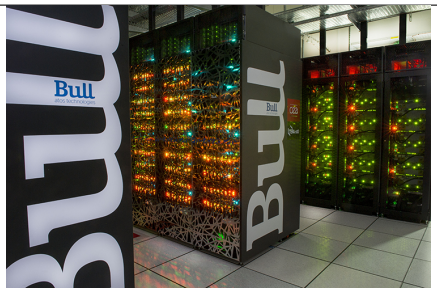
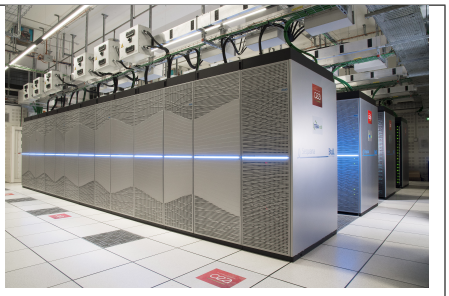
	Tera 1000-1	Tera 1000-2
Mise en service	2016	2017
Processeur	Intel Haswell	Intel KNL
cœurs / nœud	32	68
Nombre de cœurs	70272	576000
Nombre de noeuds	2196	8000
Taille mémoire/cœur	4 Go	2.7 Go
Fréquence	2.3 GHz	1.4 GHz
Performance crête	2.58 Pflops	25 Pflops
Consommation	1 MW	5 MW
Version SIMD	AVX2	AVX512
Réseau rapide	Infiniband EDR	Bull BXI
		

TABLE 3.5 – Caractéristiques principales des partitions de Tera 1000.

ou l'AVX512 du KNL sur Tera 1000-2. En effet, sur la machines Tera 100 la fonctionnalité de vectorisation n'était pas utilisée car elle n'apportait qu'un bénéfice maigre au regard des efforts nécessaires à la vectorisation des algorithmes (gain moyen observé d'environ 1.5X sur 2 théorique du SSE en double précision).

On constate aussi une montée en puissance de la programmation OpenMP au sein d'un nœud ouvrant la voie à une programmation hybride mélangeant plusieurs types de parallélismes. Ce mélange complique sensiblement les tâches de développement. Il devient nécessaire de disposer d'outils toujours plus efficaces, tels que les débogueurs (par exemple DDT de la société Arm), les outils de profilage (tel que VTune d'Intel) ou d'analyse de qualité de vectorisation comme MAQAO [maq]. La programmation hybride oblige aussi à revoir une partie de l'architecture des codes ainsi que leurs structures de données internes. Ces difficultés seront très marquées pour les codes non structurés qui, par nature, n'exhibent pas facilement de localité spatiale des données.

Débit	100 (4×25) Gbit/s)
Temps de latence	$1\mu s$
Messages émis par secondes	100. 10^6
Taille maximale d'une machine	64 000 nœuds

TABLE 3.6 – Caractéristiques techniques du BXI.

Tera 1000-2 est la première machine au monde à utiliser le réseau BXI [7] développé par la société Atos. La particularité de ce réseau est de prendre en charge au niveau matériel des opérations essentielles pour le support efficace de la bibliothèque MPI, assurant de ce fait un excellent niveau de performance de la machine. La table 3.6 donne les principales caractéristiques de ce nouvel équipement.

De cette section sur les clusters de SMP nous retiendrons que :

- l'open source est une solution viable pour exploiter les supercalculateurs ;
- l'architecture de type cluster est la seule permettant d'atteindre de très grandes puissances de calcul. Elle est donc appelée à perdurer ;

7. BXI = **B**ull E**x**ascale **I**nterconnect.

- le parallélisme se décline désormais sur trois niveaux : un macro parallélisme entre nœuds avec MPI et nécessitant un support matériel *ad hoc* tel que le BXI, un méso parallélisme en intra-nœud avec des threads (OpenMP ou TBB) et un micro parallélisme avec les unités SIMD.

3.3 Les supercalculateurs spécifiques

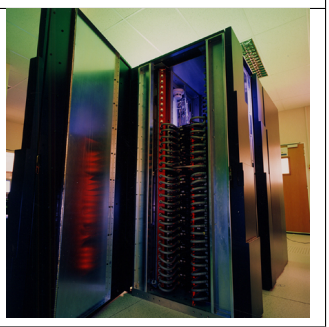
	Cray T3D	Cray T3E
Mise en service	1994	1996
Processeur	DEC EV4	DEC EV5
Nombre de CPU	128	192
Nombre de noeuds	64	96
Taille mémoire/proc	128 Mo	256 Mo
Fréquence	150 MHz	300 MHz
		

TABLE 3.7 – Caractéristiques principales des ordinateurs CRAY d'investigation du parallélisme du CEA/DAM.

3.3.1 T3D - T3E

Les machines T3D et T3E, dont les caractéristiques sont données dans la table 3.7, ont permis de s'intéresser au parallélisme à mémoire distribuée, bien que ces machines implémentaient une mémoire partagée distribuée par une combinaison de matériel spécifique et de support du système d'exploitation, et étudier comment faire évoluer les codes pour prendre en compte ce modèle de parallélisme.

Sur le T3D, l'approche maître-esclave (MPMD) était la plus naturelle, quoique son extensibilité forte soit limitée (par la saturation du maître si le nombre d'esclaves est très grand). Sur le T3E au contraire, le modèle SPMD s'est imposé car il permet une meilleure extensibilité. La parallélisation par décomposition de domaine, présentée au chapitre 2 est un bon représentant de la famille SPMD. Le T3E a montré l'importance du réseau d'interconnexion pour réaliser efficacement les échanges de maille fantômes mais aussi sur les opérations collectives comme la recherche du pas en temps global (une opération de réduction réalisée sur l'ensemble des processus). Ces constatations ont servi lors de la définition de la machine Tera 1.

La programmation sur T3D était essentiellement FORTRAN étendue par l'utilisation d'une bibliothèque d'échange de messages pour implémenter le modèle SPMD ou MPMD. Des expérimentations à base de CRAFT (« Cray Research Adaptive Fortran ») (vers 1995 au CEA sur le T3D) furent tentées mais rapidement abandonnées lors de l'arrivée de Tera 1. La première bibliothèque d'échange de messages disponible fut PVM [pvm] (Parallel Virtual Machine), créée en 1989 mais disponible seulement en 1991 au CEA. Si PVM permit d'apprendre la programmation parallèle, cette bibliothèque a été rapidement remplacée en 1992 par MPI [mpi] (Message Passing Interface) et disponible en 1995 au CEA. PVM n'a pratiquement pas été utilisé sur T3E car la migration de PVM à MPI était simple, les concepts de base étant les mêmes.

MPI prédomine aujourd'hui le monde du HPC et s'est largement enrichi depuis la version 1.0 de 1992. De nombreuses implémentations existent en Open Source et les constructeurs de supercalculateurs proposent aussi leurs versions optimisées pour leurs matériels. Il n'est pas rare de disposer de plusieurs souches de MPI sur une même machine. MPI propose (liste non exhaustive) :

- des communications point à point synchrones et asynchrones ;
- des opérations collectives synchrones et asynchrones ;
- la possibilité de créer des bibliothèques reposant sur MPI qui n'interfèrent pas avec les communications du code (sous communicateur) ;
- des fonctions d'entrées-sorties parallèles (sous-ensemble appelé MPI-IO).

3.3.2 Machines GPU

2007 a été une année charnière en matière de calcul à haute performance. Elle a vu l'arrivée des GPUs NVIDIA comme une option crédible en tant que source de puissance de calcul. Si des expérimentations d'utilisation de cartes graphiques pour le calcul avaient déjà été faites de par le passé, comme décrit par Strzodka dans [SDK05], seule l'arrivée concomitante du matériel et de l'environnement CUDA [cud] ont permis un réel décollage de l'usage des GPUs pour du calcul « généraliste », si tant est que l'algorithme s'y prête.

Des machines expérimentales ont permis de valider que les GPUs étaient utilisables en HPC, ce qui a conduit le CEA à s'équiper de la machine Titane au CCRT, Centre de Calcul Recherche et Technologie du CEA, première machine européenne de production utilisant des GPUs NVIDIA Tesla S1070 et dont les caractéristiques sont données dans la table 3.8. L'expérience positive acquise avec Titane a conduit à intégrer des partitions GPU dans Tera 100 et Tera 1000, contenant les meilleurs GPUs disponibles pour chacune de ces générations de supercalculateur. Ces GPUs servent tant à la visualisation scientifique qu'au développement de codes.

	Titane
Mise en service	2009
Processeur	Intel Nehalem-EP
cœurs / nœud	8
Nombre de noeuds	1068
GPU	48 serveurs NVIDIA Tesla S1070
Réseau rapide	Infiniband DDR
	

TABLE 3.8 – Caractéristiques principales de Titane, première machine européenne de production utilisant des GPUs pour le calcul scientifique.

Contrairement à un CPU, un GPU ne pourra pas être utilisé pour n'importe quel type de programme. En effet, un GPU diffère notablement d'un CPU par sa conception. Comme un GPU a été conçu dès l'origine pour le graphique, sa structure interne très régulière, illustrée par les figures 3.6 et 3.7, va imposer des contraintes fortes sur la manière de les programmer. Si un CPU permet une vue assez linéaire du parcours des données, même si l'usage des unités fonctionnelles SIMD impose une vue plus « groupée » selon la taille des vecteurs, un GPU réclamera de plaquer le problème sur une hiérarchie de blocs répartis sous forme de grille et d'un ensemble de threads indivisibles (appelés « warp » chez NVIDIA et « wavefront » chez AMD), donc synchrones (modèle SIMT).

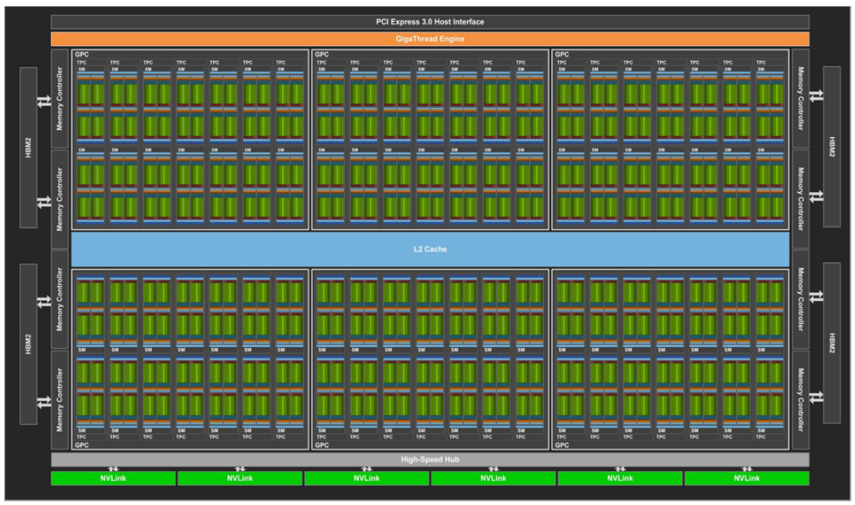


FIGURE 3.6 – Le GPU de dernière génération de chez NVIDIA : le Volta sorti en 2017. La régularité de la conception de cette carte graphique est évidente. Un zoom sur les unités fonctionnelles la composant est donné en figure 3.7.

Cette structuration matérielle va être très adaptée à des problèmes très réguliers tels que les codes structurés, des problèmes d'algèbre linéaire dense et sera nettement plus difficile à utiliser sur des codes non structurés. En effet, pour les deux premières catégories, il sera facile de subdiviser le domaine en blocs qui auront une correspondance directe avec ceux du matériel.

Une autre limitation des GPUs apparaît dans le traitement des branchements (conditionnelles). Si un CPU dispose de mécanismes sophistiqués pour la prédiction des branchements (qui sont classiquement des sections de programme comme :

```
1 if (condition) {
2         // branche 1
3 } else {
4         // branche 2
5 }
```

chaque branche pouvant contenir d'autres tests, un GPU est désarmé pour traiter efficacement celles-ci. Les branches sont qualifiées de divergence et doivent être minimisées pour ne pas gréver les performances du code. Cela conduit généralement à une restructuration des programmes pour faire sortir les tests en dehors des parties calculatoires. La gestion des divergences sera difficile avec des algorithmes de type Monte Carlo au comportement intrinsèquement aléatoire.

La dernière particularité des GPUs impactant la programmation est la notion de coalescence des accès aux données. Pour que le système mémoire du GPU fonctionne à plein régime, il faut que l'ensemble des threads accèdent à des données contiguës en mémoire de façon à fusionner l'ensemble des requêtes en une seule transaction. C'est cette particularité qui rend les GPUs difficiles à utiliser dans les codes non structurés car ces derniers reposent essentiellement sur des indirections, peu ou pas propices à la coalescence des accès mémoire.

La connaissance fine du GPU est requise pour en tirer la performance maximale et impactera la manière de coder. Si l'on prend un simple exemple de réduction, dont les sources C sont donnés table 3.9, on constate que ni la version CUDA[cud] (table 3.10), ni la version OpenCL[opeb] (table 3.11) ne sont réellement simples et devront être adaptées à chaque nouvelle génération de matériel.

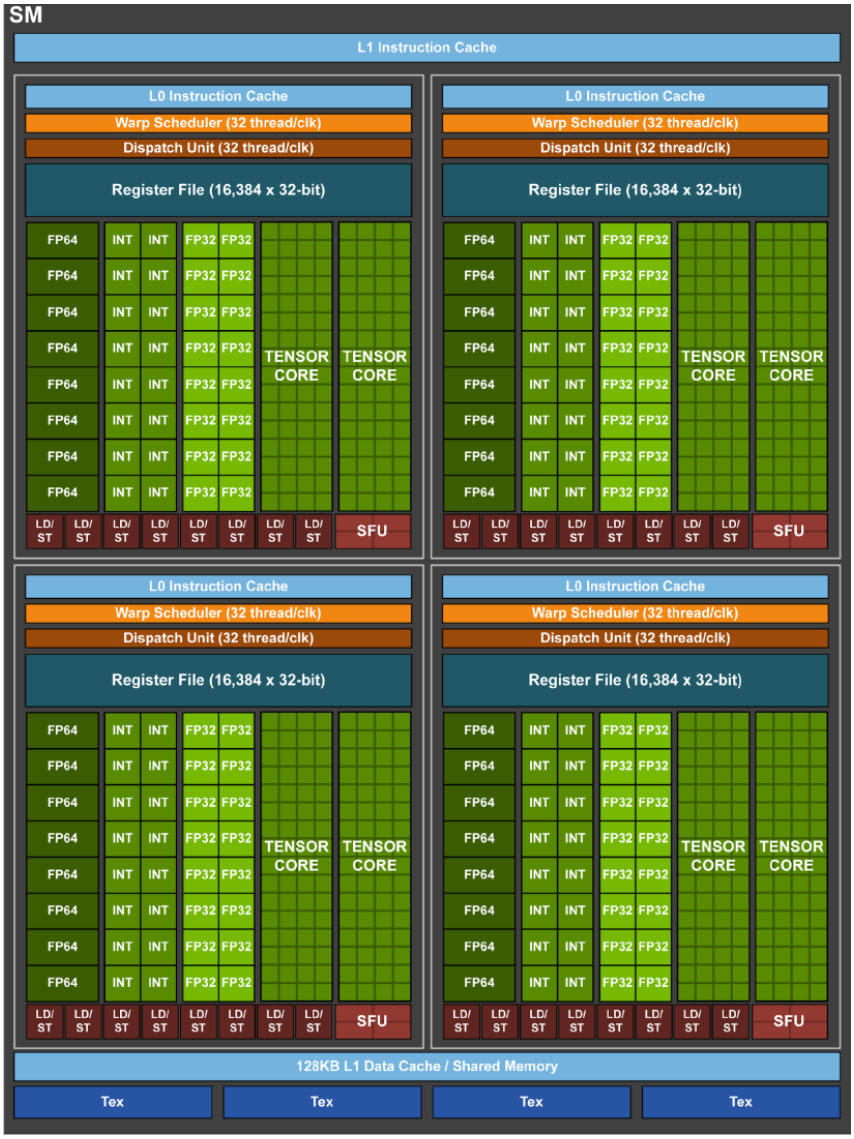


FIGURE 3.7 – Détail de l'architecture d'un SM (Symmetric Multiprocessor) du GPU Volta de 2018. L'aspect symétrique de cette architecture apparaît clairement. Ce qui est nouveau par rapport à la génération des GPUs de Titane est l'apparition des unités de calcul FP64, inexistantes dans les S1070, et des « tensor core » spécialement conçus pour accélérer les calculs d'intelligence artificielle (IA).

```
1 sum = 0.0;
2 for (i = 0; i < N; i++) {
3         sum += array[i];
4 }
```

TABLE 3.9 – Une simple réduction en C. Cette opération est très utilisée dans les codes scientifiques, par exemple pour sommer les contributions de particules à l'énergie contenue dans une maille.

Les GPUs se programment de deux façons : soit en utilisant un langage de bas niveau spécialement conçu, soit en utilisant des annotations du C ou du FORTRAN.

En 2009, les deux options disponibles étaient CUDA, pour le langage spécifique, et HMPP pour la voie annotation. En 2019, le choix est plus vaste car en plus de CUDA, il y a OpenCL en langage spécifique et OpenACC [opea] et OpenMP [oped] pour les annotations.

CUDA est le langage propriétaire de NVIDIA, spécialement adapté à ses matériels. Très vite, il est apparu qu'une option standardisée était nécessaire pour assurer une portabilité des programmes. C'est ainsi qu'est né OpenCL [opeb]. Ce dernier reprend les concepts de base de CUDA mais en diverge tant sur la mise en oeuvre que sur certains concepts, tels que les queues de commandes (« command queues ») et le pilotage du programme par événements (« events »).

HMPP, développé par la défunte société CAPS Entreprise, a été précurseur de l'approche par annotation pour les GPUs, reprenant le modèle d'OpenMP pour les CPUs. Un exemple de portage d'application sous HMPP est donné dans l'exposé [Gui10]. En fin de vie HMPP a fortement influencé OpenACC, poussé par les sociétés CAPS, CRAY et NVIDIA. OpenACC est une initiative visant initialement à prototyper ce que devaient être les extensions d'OpenMP pour les GPUs puis éventuellement disparaître au profit d'OpenMP. En pratique, OpenACC reste une option bien vivante car plus simple qu'OpenMP.

L'atout des annotations est de ne pas (trop) exposer au programmeur les impératifs de programmation liés au matériel. Notre exemple de réduction exprimé en OpenACC (table 3.12) voire OpenMP (table 3.18 bien qu'ici la directive soit aussi valable pour un CPU) montre la simplicité de la mise en œuvre des annotations.

HMPP présentait des options assez pratiques pour le portage de codes CPU sur GPU sans modification du source original mais permettant d'atteindre la performance comme dans l'exemple de la table 3.13 ou dans la table 3.14 pour reprendre notre exemple de réduction. La tendance actuelle est de faire confiance au compilateur pour détecter automatiquement de telles situations et de restructurer le code pour l'utilisateur. HMPP sera discuté dans la section 5.2.1 dans un contexte d'utilisation pour un code de physique.

A la difficulté d'exprimer l'algorithme en respectant les contraintes liées au GPU, s'ajoute une autre contrainte pour le programmeur : gérer astucieusement les mouvements de données. En effet le GPU (surtout en 2009) ne partage pas le même espace d'adressage que le CPU (ce qui n'est plus vrai en 2019 pour le couple IBM Power 9 – NVIDIA Volta qui utilise pour cela les capacités du lien NVLINK 2). Le programmeur doit d'une manière ou d'une autre « monter » les données à traiter sur le GPU et « redescendre » les résultats. Il dispose pour cela de fonctions en CUDA et OpenCL ou de directives spécialisées en HMPP – OpenACC – OpenMP. Comme le GPU est relié au CPU via le PCI-Express, qui est un lien lent comparé aux performances des mémoires, ces opérations de déplacement des données

```
template <unsigned int blockSize>
__global__ void reduce6(int *g_idata, int *g_odata, unsigned int n)
{
    extern __shared__ int sdata[];
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*(blockSize*2) + tid;
    unsigned int gridSize = blockSize*2*gridDim.x;
    sdata[tid] = 0;
    while (i < n) {
        sdata[tid] += g_idata[i] + g_idata[i+blockSize];
        i += gridSize;
    }
    __syncthreads();
    if (blockSize >= 512) {
        if (tid < 256) {
            sdata[tid] += sdata[tid + 256];
        }
        __syncthreads();
    }
    if (blockSize >= 256) {
        if (tid < 128) {
            sdata[tid] += sdata[tid + 128];
        }
        __syncthreads();
    }
    if (blockSize >= 128) {
        if (tid < 64) {
            sdata[tid] += sdata[tid + 64];
        }
        __syncthreads();
    }
    if (tid < 32) {
        if (blockSize >= 64) sdata[tid] += sdata[tid + 32];
        if (blockSize >= 32) sdata[tid] += sdata[tid + 16];
        if (blockSize >= 16) sdata[tid] += sdata[tid +  8];
        if (blockSize >=  8) sdata[tid] += sdata[tid +  4];
        if (blockSize >=  4) sdata[tid] += sdata[tid +  2];
        if (blockSize >=  2) sdata[tid] += sdata[tid +  1];
    }
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

TABLE 3.10 – Une réduction de sommation en Cuda. Cette implémentation est expliquée en détail sur le site `http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/reduction/doc/reduction.pdf`. Nous constatons qu'une connaissance fine du matériel utilisé est requise pour atteindre les meilleures performances : les boucles ont été déroulées et la structure d'exécution des threads a été exploitée pour limiter le recours aux synchronisations – sur un GPU NVIDIA tous les threads d'un ≪ warp ≫ (un groupe de 32 voire 64 threads) sont exécutés en mode SIMT donc il n'y a pas besoin de les synchroniser. Nous pouvons aussi remarquer la longueur de la version GPU d'un programme très simple (voir table 3.9).

```
__kernel
void reduce(__global float* buffer,
            __local float* scratch,
            __const int length,
            __global float* result) {

  int global_index = get_global_id(0);
  float accumulator = INFINITY;
  // Loop sequentially over chunks of input vector
  while (global_index < length) {
    float element = buffer[global_index];
    accumulator = (accumulator < element)? accumulator: element;
    global_index += get_global_size(0);
  }

  // Perform parallel reduction
  int local_index = get_local_id(0);
  scratch[local_index] = accumulator;
  barrier(CLK_LOCAL_MEM_FENCE);
  for(int offset = get_local_size(0) / 2;
      offset > 0;
      offset = offset / 2) {
    if (local_index < offset) {
      float other = scratch[local_index + offset];
      float mine = scratch[local_index];
      scratch[local_index] = (mine < other) ? mine : other;
    }
    barrier(CLK_LOCAL_MEM_FENCE);
  }
  if (local_index == 0) {
    result[get_group_id(0)] = scratch[0];
  }
}
```

TABLE 3.11 – Une réduction de sommation en OpenCL. Le site `http://developer.amd.com/documentation/articles/Pages/OpenCL-Optimization-Case-Study-Simple-Reductions_6.aspx` détaille cette implémentation. Il faut remarquer que celle-ci n'est pas très éloignée de la version CUDA (voir table 3.10), les quelques différences apparaissent sur la synchronisation des threads lors de la phase parallèle.

```
#pragma acc parallel loop
for(int i=0;i<num_rows;i++) {
   double sum=0;
   int row_start=row_offsets[i];
   int row_end=row_offsets[i+1];
#pragma acc loop reduction(+:sum)
   for(int j=row_start;j<row_end;j++) {
      unsigned int Acol=cols[j];
      double Acoef=Acoefs[j];
      double xcoef=xcoefs[Acol];
      sum+=Acoef*xcoef;
   }
   ycoefs[i]=sum;
}
```

TABLE 3.12 – Exemple de réduction de données stockées au format CSR tirée du guide de programmation OpenACC (`https://www.openacc.org/sites/default/files/inline-files/OpenACC_Programming_Guide_0.pdf`).

Avant	Après

`!$hmppcg permute k, i, j`
`DO I = 1, N`
`  DO J = 1, N`
`  DO K = 1, N`
`  A(I,J,K) = B(I,J,K)*1.2`
`  ENDDO`
`  ENDDO`
`ENDDO`
	`!`
	`DO K = 1, N`
	`  DO I = 1, N`
	`  DO J = 1, N`
	`  A(I, J, K) = B(I,J,K)*1.2`
	`  ENDDO`
	`  ENDDO`
	`ENDDO`

TABLE 3.13 – Cet exemple illustre la réorganisation de code reposant sur une directive HMPP. Le code « Avant » est efficace sur un CPU et lent sur un GPU tandis que celui généré par le compilateur HMPP qui est donné dans la colonne « Après » est bien adapté aux GPUs. Faire ce type de transformation manuellement est fastidieux sur un grand code, présente un grand risque d'introduire des erreurs et de conduire à une version sous-optimale sur CPU. L'usage de telles directives permet de conserver la portabilité **et** la performance aux changements d'architectures.

```
1 #pragma hmppcg gridify, reduce (+:ssx,+:ssy)
2 for (i = 0; i < NK; i++) {
3    if (qqprim2[i]) {
4       qq[qqprim[i]] += 1.0;
5       ssx = ssx + qqprim3[i]; /* sum of Xi */
6       ssy = ssy + qqprim4[i]; /* sum of Yi */
7    }
8 }
```

TABLE 3.14 – Une réduction en HMPP. La directive `hmppcg` précise les conditions de génération de code, ici qu'il faut utiliser une grille de threads sur le GPU et prendre en compte l'opération de réduction sur les variables `ssx` et `ssy`. La simplicité des directives HMPP se retrouve dans celles d'OpenMP (voir l'exemple de la table 3.17).

doivent être attentivement étudiées et optimisées pour ne pas effondrer les performances globales du programme. Un ping-pong continuel entre CPU et GPU représente une situation souvent catastrophique mais inévitable dans les premières itérations de portage sur GPU. Dans le cas d'un programme s'exécutant sur plusieurs nœuds de calcul disposant de GPUs, il sera nécessaire de pouvoir échanger des données (les mailles fantômes par exemple) entre les GPUs. Ce sera souvent une occasion de perte de performance car il faudra que les données suivent le chemin :

$$[GPU source - PCIe - CPU] - réseau - [CPU - PCIe - GPU destination]$$

Il existe néanmoins un court-circuit possible si la machine est équipée d'un réseau d'origine Mellanox. C'est le produit GPUdirect qui autorise un GPU à envoyer des messages MPI sur le réseau à destination d'un autre GPU sans passer par la mémoire du CPU ni solliciter ce dernier.

3.3.3 Intel manycore

Face à la montée en puissance des GPUs, Intel a essayé de disposer de son propre GPU de haute performance et a lancé le projet Larrabee, dont la première version est sortie en 2008. Ce fut un échec cuisant. Néanmoins cette carte graphique avait le potentiel pour servir au calcul. Une version améliorée a vu le jour en 2012 sous le nom de KNC (pour Knights Corner). Le KNC restait dans la philosophie de la carte additionnelle (sur le PCI-Express) servant à accélérer les calculs. En 2016 est apparu le KNL (Knights Landing) qui cette fois-ci était un processeur à part entière et qui corrigeait les nombreux défauts du

KNC.

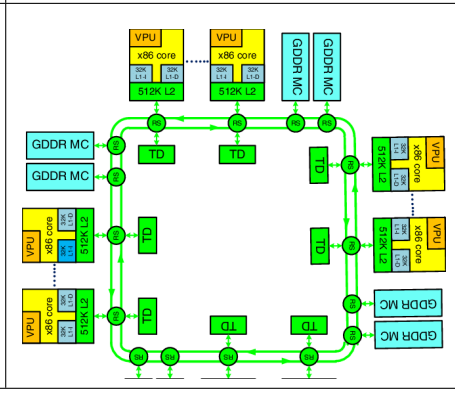
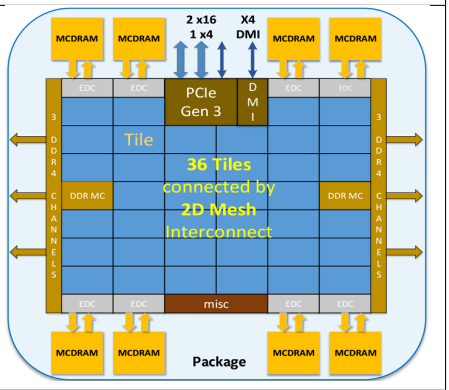
	KNC	KNL
Mise en service	2012	2016
Taille mémoire/proc	16 Go	16 Go MCDRAM / 384 Go DDR4
Nombre de coeurs	61	68
Fréquence	1.24 GHz	1.4 GHz
Version SIMD	MIC512	AVX512
		

TABLE 3.15 – Les manycore de chez Intel : KNC et KNL

Selon Intel, KNC et KNL sont des « manycore » car ils possèdent un grand nombre de cœurs de calcul par rapport aux processeurs de l'époque (entre 4 et 8 classiquement). La table 3.15 en donne les principales caractéristiques. KNC et KNL partagent des points communs :

- ils utilisent des cœurs x86 ;
- la puissance de calcul provient de leurs unités SIMD permettant d'opérer sur des vecteurs de 8 flottants double précision en une seule opération.

Par contre, ils divergent sur plusieurs points tels que :

- les cœurs du KNC sont reliés entre eux par un anneau qui limite les performances des accès à la mémoire (forte contention). Le KNL, au contraire, a une structure interne sous forme de grille (« mesh ») qui améliore fortement les performances des mouvements de données internes au processeur ;
- le KNC a peu de mémoire (16 Go de GDDR5) alors que le KNL dispose de 16 Go de MCDRAM à la bande passante élevée et jusqu'à 384 Go de DDR4 ;
- pour le KNC, le mode d'utilisation préconisé par Intel est le mode « offload » (déchargement du travail et des données sur le KNC) comme pour un GPU. Le KNL étant un vrai processeur, cette gestion des mouvements de données n'a pas lieu d'être.

Le KNC a surtout servi à travailler la vectorisation des algorithmes. Il ne sera disponible au CEA que dans quelques prototypes. Le KNL, déployé massivement dans Tera1000, poursuit cette voie mais y ajoute la possibilité de la programmation hybride MPI + threads, le plus souvent MPI + OpenMP.

SIMD et OpenMP ont été cités plusieurs fois plus haut. Comme ce sont deux notions centrales pour la programmation sur KNL, mais aussi pour les processeurs classiques dont le nombre de cœurs ne cesse de croître, il est utile de détailler leurs implications sur la programmation dans les deux sections suivantes.

3.3.3.1 La programmation SIMD

Pour programmer efficacement un processeur moderne possédant une ou plusieurs unités SIMD, le programmeur doit faire attention aux notions fondamentales suivantes : la structuration de ses données, la prise en compte des caches, la gestion des dépendances entre les données. Si ces conditions sont réunies, l'utilisation des instructions SIMD sera réellement efficace et conduira à une accélération notable du programme.

Une instruction SIMD telle que l'addition prend deux registres vectoriels en entrée et produit le résultat dans un registre vectoriel. Selon le processeur, ces registres peuvent avoir des tailles variables, mais fixées par l'ISA [8] du processeur. SSE [9] manipule des registres de 128 bits donc peut travailler sur 2 flottants double précision (FPDP) ou 4 flottants de simple précision (FPSP), AVX2 de 256 bits soit 4 FPDP ou 8 FPSP, AVX512 de 512 bits 8 FPDP ou 16 FPSP. SVE [10], récemment défini par Arm, permet de manipuler des vecteurs de 128 bits jusqu'à 2048 bits selon l'implémentation matérielle.

En théorie, une boucle utilisant une opération SIMD sera exécutée N fois plus vite que son équivalent scalaire, si N est le nombre d'éléments contenu dans un registre vectoriel. En pratique l'accélération constatée est toujours moins importante (1.5 / 2 en SSE, 3 / 4 en AVX2, 6 / 8 en AVX512 sur les FPDP). Les gains apportés par la vectorisation ne sont sensibles qu'à partir d'AVX2, ce qui explique le regain récent d'intérêt des développeurs pour la vectorisation.

RAW (Read after Write)	WAR (Write after Read)
`for (i = 0; i < N; i++) { a[i] = a[i-1] + b[i]; }`	`for (i = 0; i < N; i++) { a[i] = a[i+1] + b[i]; }`
Ce type de boucle n'est pas vectorisable car le résultat de l'itération courante dépend du résultat de l'itération précédente (dépendance de type « *flow* » dans le langage des compilateurs).	Cette boucle est vectorisable car la valeur `a[i+1]` n'est que lue à l'itération `i` et ne sera modifiée qu'à l'itération suivante (`i + 1`).
RAR (Read after Read)	**WAW** (Write after Write)
`for (i = 0; i < N; i++) { a[i] = b[i % 2] + c[i]; }`	`for (i = 0; i < N; i++) { a[i % 2] = b[i] + c[i]; }`
Malgré la non linéarité de la variation des index du tableau b, cette boucle est vectorisable car n'affecte en rien les affectations de a. Il n'y a pas vraiment de dépendance ici.	Plusieurs itérations écrivent sur le même élément de tableau. Il est donc impossible de vectoriser ce genre de boucle.

TABLE 3.16 – Les 4 types de dépendances entre itérations d'une boucle pouvant affecter la vectorisation. Les deux cas indiqués en rouge ne sont pas vectorisables.

Quelles sont les conditions pour une utilisation optimale des unités SIMD et donc atteindre les meilleures performances du processeur ? Le programmeur devra veiller à quatre points.

Pour remplir les registres il est nécessaire d'accéder aux données le plus vite possible. Comme nous l'avons vu dans la section 3.2.1 (page 33), les caches sont lus par lignes entières. Le programmeur devra donc veiller à ce que ses données manipulées dans une boucle à vectoriser utilisent au mieux les caches. Une structure C telle que :

```
1 typedef struct _particule {
2         long num;
3         double x, y, z;
4         double t, m;
5 } Particule_t;
```

(conduisant à l'utilisation d'un tableau de structures ou AOS pour « Array Of Structure ») ne sera pas adaptée pour des opérations vectorielles ne travaillant que sur **un** des champs, même si elle est une

8. Instruction Set Architecture ou jeu d'instruction du processeur
9. Simd Streaming Extension. L'équivalent de SSE est NEON chez Arm
10. Scalable Vector Extension

expression naturelle de ce que doit être une particule pour le problème à simuler. En effet, si l'on parcourt un tableau de type `Particule_t` uniquement sur la variable `t`, les données ne seront pas contiguës en mémoire. Les lignes de caches seront « polluées » par des données inutiles (ici `num, x, y, z, m`). Il faudra donc faire plus de transactions mémoire que si l'on ne manipulait qu'un tableau ne contenant que les `t`, d'où une efficacité moindre de cette section de code. Cela poussera donc le programmeur à réorganiser son code, par exemple en utilisant une structure telle que :

```
typedef struct _tabparticule {
        long *num;
        double *x, *y, *z;
        double *t, *m;
} TabParticule_t;
```

dans laquelle toutes les grandeurs `t` seront contiguës (SOA : « Structure Of Arrays » ou structure de tableaux).

Les codes non structurés (voire même les codes AMR) présentent une autre difficulté pour la vectorisation : leur nécessité de recourir aux indirections pour accéder aux informations casse la possibilité d'un accès séquentiel aux données. Pour préparer des tableaux contigus en mémoire, le programmeur devra réaliser un compactage des données dans un tableau de travail par une opération de type `gather`[11] puis après traitement, ventiler les résultats dans les structures de données du code par un `scatter`[12]. Sans support matériel de ces deux opérations, le code ne sera pas vectorisable directement si de telles indirections sont présentes au sein de la boucle de travail.

Une fois résolu l'accès efficace aux données, le programmeur devra vérifier si la boucle à vectoriser ne tombe pas dans les cas RAW ou WAW par une analyse fine des enchaînements des opérations. Les différents types de dépendances des données, impactant ou non la vectorisation, sont rappelés dans la table 3.16. Souvent le compilateur aide le programmeur en expliquant pourquoi la vectorisation n'est pas possible et il indique (plus ou moins clairement) sur quelle variable se trouve le problème. Charge ensuite au développeur de modifier sa programmation pour contourner le problème tout en respectant la sémantique de son programme. Dans les cas de boucles complexes, il est souvent nécessaire d'aider le compilateur en ajoutant au source du programme des indications aidant la vectorisation.

```
sum = 0.0;
#pragma omp simd reduction(+: sum)
for (i = 0; i < N; i++) {
        sum += array[i];
}
```

TABLE 3.17 – Une réduction en OpenMP utilisant l'extension vectorielle « simd »

En 2019, avec l'arrivée d'OpenMP 5.0, il est désormais recommandé d'utiliser les directives `simd`, pour préciser les intentions du programmeur afin d'aider le compilateur, au lieu d'utiliser des directives propriétaires telles que « `#pragma ivdep` » qui force la vectorisation au risque d'introduire des bugs dans le programme. La table 3.17 donne un exemple de réduction vectorisée utilisant une annotation `simd`.

En résumé, les conditions d'une bonne vectorisation sur les processeurs modernes sont :

- traiter (réutiliser) des données déjà présentes dans les caches (localité temporelle) ;
- accéder à des données proches en mémoire (localité spatiale) ;

11. l'opérateur « gather » réalise la boucle suivante : `for (i = 0; i < N; i++) temp[i] = a[index[i]];`
12. l'opérateur « scatter » réalise la boucle suivante : `for (i = 0; i < N; i++) b[index[i]] = temp[i];`

- éliminer les dépendances entre variables inhibant la vectorisation ;
- aider le compilateur en précisant ses intentions à l'aide de directives.

3.3.3.2 La programmation OpenMP

```
1 sum = 0.0;
2 #pragma omp parallel for shared(sum, array) reduction(+: sum)
3 for (i = 0; i < N; i++) {
4         sum += array[i];
5 }
```

.

TABLE 3.18 – Une réduction en OpenMP

OpenMP est un jeu de directives pour le parallélisme intra-nœud, sous la forme d'annotation du source du programme (un exemple est donné en table 3.18), qui permet de tirer avantage des ordinateurs possédant plusieurs processeurs par nœud. L'avantage de cette approche repose sur le fait que, si ces annotations sont ignorées par le compilateur, le programme fonctionne normalement mais plus lentement. OpenMP a été peu utilisé initialement car les accélérations n'étaient pas bonnes à cause de l'immaturité des exécutifs (« runtime ») et n'offraient que peu d'intérêt face à la programmation tout MPI. En effet, cette dernière permet aux programmes de s'exécuter dans des espaces d'adressage différents et donc de ne pas tomber dans les pièges propres à la programmation par thread (conflit de cache, false sharing, Amdahl, ...). Depuis l'arrivée des processeurs à nombre de cœurs élevé, la programmation OpenMP prend une grande importance.

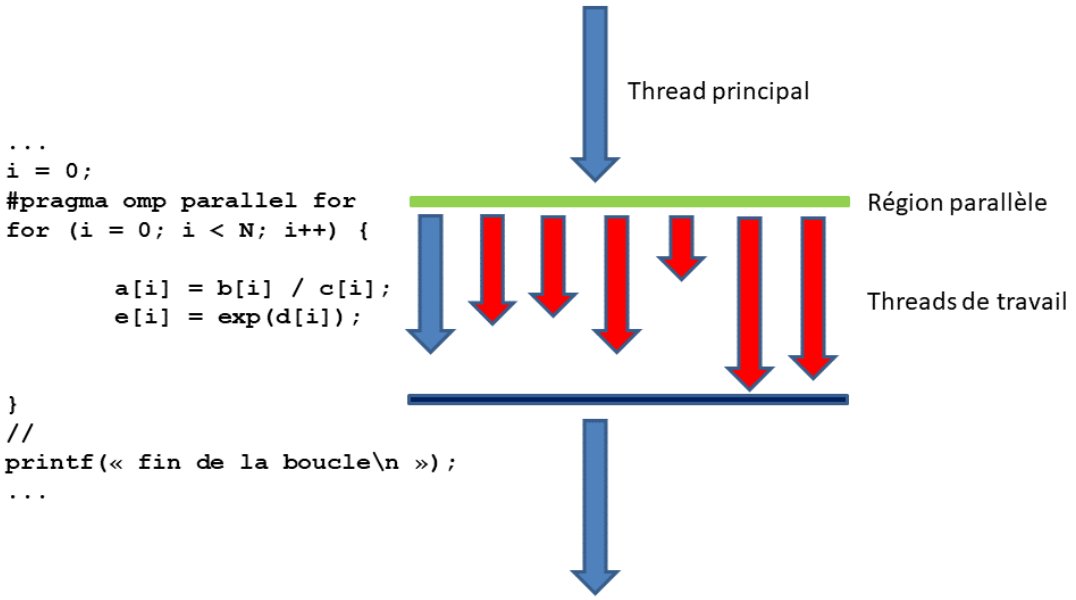


FIGURE 3.8 – Le modèle fork-join : à l'entrée de la boucle, au niveau de la barre verte, des threads de travail (en rouge) sont créés [fork]. Les différentes itérations de la boucle sont réparties sur les threads et exécutées. Quand toutes les itérations sont terminées, les threads sont tous synchronisés [join] (barre noire) pour attendre le thread le plus lent et les threads de travail sont détruits. Seul reste le thread principal (flèche bleue) qui continue le déroulement du programme jusqu'à la prochaine section parallèle.

OpenMP repose sur un modèle de thread (ou fil d'exécution) de type fork-join comme illustré dans par la figure 3.8. Ce modèle d'exécution est très sensible à la loi d'Amdahl décrite précédemment, si les sections parallèles comportent peu d'instructions. Ce sera le cas de l'annotation systématique des boucles de calcul

pouvant être exécutées en parallèle. On parlera alors d'une parallélisation à grain fin qui est souvent la plus facile à mettre en place mais dont les accélérations seront faibles.

Pour améliorer le passage à l'échelle des programmes OpenMP, il est nécessaire d'étendre la granularité des sections parallèles. On ne s'intéressera plus aux simples boucles mais aux sous-programmes entiers, lesquels seront exécutés sur des tranches de données affectées aux threads. Cette parallélisation à gros grain offre de bien meilleurs comportements sur les processeurs à grand nombre de cœurs. OpenMP offre aussi la possibilité de manipuler des tâches et depuis la version 4.5 de déporter une partie des calculs sur des accélérateurs de type GPU.

La taille croissante des simulations sur des supercalculateurs ayant des processeurs à grands nombre de cœurs conduit à la montée en puissance de la programmation hybride de type MPI + X, où X est le plus souvent OpenMP, mais cela peut être aussi un modèle de thread de type TBB [tbb]. La gestion du parallélisme inter-nœud est assurée par MPI, et l'intra-nœud est pris en charge par OpenMP tant pour la parallélisation de thread que pour la vectorisation, comme nous l'avons vu à la section précédente. Une telle approche permet d'économiser la mémoire nécessaire à MPI pour sa gestion des communications mais aussi apporter une économie en mémoire à l'application par réduction du nombre de zones de mailles fantômes à établir. En effet, plus un domaine 3D à traiter est volumineux, plus sa surface est faible en proportion (resp. surface et contour en 2D). Or les communications MPI sont proportionnelles à la surface (resp. contour) d'un sous-domaine. D'où l'intérêt de maximiser les volumes à traiter et donc de minimiser le nombre de sous-domaines traités par MPI au profit d'un parallélisme de thread à l'intérieur des sous-domaines.

Les notions qui ont été détaillées dans cette section sur les supercalculateurs spécifiques vont servir lors de la section 5.2 sur mes expériences relatives au massivement parallèle. Les points les plus importants qui se dégagent sont :

- hégémonie de MPI pour la programmation à mémoire distribuée ;
- simplicité de la programmation par annotation ;
- montée en puissance d'OpenMP et de la programmation hybride MPI+X ;
- les GPUs sont des options intéressantes face aux CPUs pour l'efficacité énergétique mais ils ne sont pas encore des processeurs complètement généralistes.

3.4 Classification des calculateurs CEA

Pour faire une synthèse des diverses machines présentées ci-dessus, en se reposant sur la taxonomie de Flynn décrite à la section 2.2.2, la table 3.19 offre une classification des supercalculateurs du CEA selon leurs modes principaux d'utilisation. Toutes les machines récentes (T3D, T3E, Tera[1..1000], Titane) utilisent des microprocesseurs ayant des unités SIMD ; il n'en est donc pas fait mention dans le tableau pour l'alléger.

Le type SISD ne présente pas d'intérêt ici et le MISD n'est pas représenté à l'heure actuelle au CEA. Le MPMD n'est pas utilisé en production sauf pour implémenter le dépouillement *in situ* à l'aide de co-processus. On constate donc que tous les modes ont été expérimentés au cours du temps pour finalement converger vers le SPMD pour la plus grosse partie de la production et du SIMT pour les partitions GPU dédiées au calcul comme à la visualisation des résultats.

	Mémoire partagée	Mémoire distribuée	Mémoire partagée distribuée
SIMD	Cray 1 Cray XMP Cray YMP Cray T90		
SIMT		Titane (GPU) Tera [100..1000] (GPU)	
MPMD			T3D
SPMD		Tera [1..1000]	T3D T3E

TABLE 3.19 – Ventilation selon la taxonomie de Flynn des supercalculateurs du CEA décrits dans ce manuscrit. La couleur verte signifie que ce type d'architecture est toujours en service au CEA. NB : la machine Titane du CCRT a été remplacée par une série de supercalculateurs possédant tous des GPUs.

3.5 La comparaison des supercalculateurs

Très rapidement s'est posée la question de la comparaison des supercalculateurs entre eux. L'aspect politique de la chose, important dans d'autres contextes, est hors de propos dans ce manuscrit. Ce qui va intéresser les vendeurs de supercalculateurs est de positionner leurs machines comme très puissantes (TOP500 [top]), très économes (GREEN500 [gre]) ou efficace pour traiter de grandes bases de données (Graph500 [gra]). Depuis peu, une nouvelle métrique se développe, le classement HPCG [hpc], qui vise à démontrer l'efficacité réelle des ordinateurs.

Cette section décrit ces quatre classements et ce qu'ils mesurent.

3.5.1 TOP500

Le classement du TOP500 repose sur la performance des machines mesurée par le HPL. Le benchmark HPL [hpl] (pour High Performance Linpack) est un programme que résout le système linéaire dense $A\vec{x} = \vec{b}$. La méthode de résolution est imposée. Elle effectue $O(N^3)$ opérations, si N est la taille de la matrice A. Le programme est optimisé pour faire la résolution en traitant la matrice par blocs de tailles compatibles avec les caches et minimiser les accès au réseau. En conséquence, le HPL va utiliser la performance maximale des processeurs et se rapprocher de la performance crête théorique de la machine. Le rendement Linpack, défini comme le rapport de la performance mesurée avec le HPL divisé par la puissance crête théorique, est une bonne mesure de la conception d'un supercalculateur pour sa capacité de calcul brut. Étant optimisé pour entrer dans les caches, le HPL est assez peu sensible à la performance des mémoires ; par contre plus la matrice sera grosse, plus le Linpack tirera de la performance de la machine, pour un temps de calcul de plus en plus important. Afficher un score au TOP500 est devenu un art : c'est trouver le bon jeu de paramètres pour que le temps de calcul reste raisonnable et que la performance soit la meilleure possible, souvent sur une machine instable car tout juste installée et non expurgée de ses composants défectueux ou fragiles.

La figure 3.9 montre que depuis 2013 environ, les performances des machines commencent à ne plus croître aussi vite que par le passé. Cette inflexion est fortement liée au ralentissement de la loi de Moore et à la recherche d'architectures plus performantes. La première machine hybride numéro un est apparue en novembre 2010 (voir à ce titre la discussion de la section 9.1.1 et la figure 9.1). Depuis, la majorité des machines les plus puissantes comporte des accélérateurs. Le HPL favorise donc les processeurs ou les accélérateurs les plus puissants.

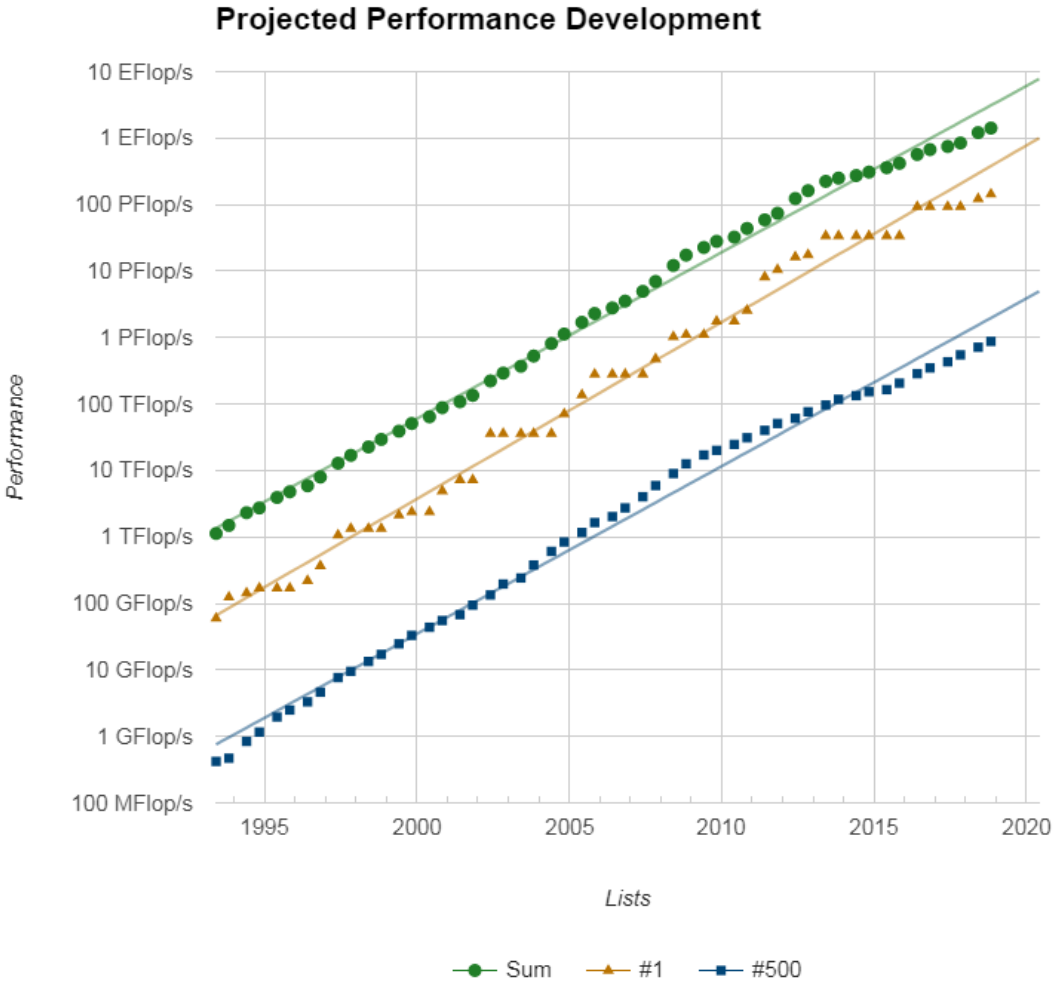


FIGURE 3.9 – Evolution de la puissance de calcul de la première machine du TOP500 au cours du temps, de la dernière du classement ainsi que la somme cumulée des performances des 500 machines de la liste. ©TOP500, liste de novembre 2018.

3.5.2 Green500

Dérivée de la liste TOP500, celle du GREEN500 classe les machines du TOP500 par efficacité énergétique exprimée en GFlops par watt. Cette liste ne reflète pas du tout celle du TOP500 car des machines peu performantes en calcul se révèlent être très efficaces selon cette grille d'analyse : sur la liste de novembre 2018, la machine numéro 1 au GREEN500 n'est que 374^e au TOP500 grâce à son accélérateur (le PEZY-SC2). Néanmoins, cette liste démontre une prise de conscience de la nécessité de concevoir des machines économes en énergie.

La table 3.20 indique clairement que les machines les plus performantes sont loin d'être celles qui sont les plus économes en énergie.

3.5.3 Graph500

A l'inverse du TOP500 qui essaye de déterminer la puissance de calcul maximale possible d'une machine, le Graph500 [gra] va essayer de solliciter quasi exclusivement les systèmes mémoire et réseau de l'ordinateur. Pour se faire le programme génère un graphe remplissant pratiquement toute la mémoire et va effectuer

# Green	# Top500	GFlops/W	nom
1	375	17.604	Shoubu system B
2	374	15.113	DGX SaturnV Volta
3	1	14.668	Summit
4	7	14.423	AI Bridging Cloud Infrastructure
5	22	13.704	TSUBAME3.0
6	2	12.723	Sierra
7	446	12.681	AIST AI Cloud
8	441	11.865	MareNostrum P9 CTE

TABLE 3.20 – Comparaison des classements Green500 et TOP500, liste de novembre 2018.

soit une traversée de ce graphe par un parcours exhaustif et récursif de tous les nœuds [13] soit chercher le chemin minimal d'un nœud vers tous les autres [14]. Ces types d'algorithmes sont utilisés pour l'exploitation de grandes bases de données.

BFS		
# Graph	# Top500	nom
1	18	K Computer
2	3	Taihulight
3	10	Sequoia
4	21	Mira
5	8	SuperMUC-NG

TABLE 3.21 – Comparaison des classements Graph500 et TOP500 pour la méthode BFS, liste de novembre 2018.

La table 3.21 montre bien que les machines conçues pour la performance de calcul ne sont pas forcément adaptées pour des charges de travail de type « Big Data ».

3.5.4 HPCG

Le Linpack résout un système dense. Or les codes ayant besoin de résoudre ce genre de systèmes sont rares. Il est plus fréquent de devoir résoudre des systèmes creux [15], c'est-à-dire étant majoritairement remplis par des zéros. Pour éviter de gaspiller de la mémoire, de telles matrices sont stockées dans des formats plus compacts, par exemple au format CSR [16]. Ces formats compacts reposent sur des indirections qui vont interférer avec les mécanismes de cache conduisant à une performance globale qui s'éloigne nettement de la performance crête atteinte avec le Linpack.

Le HPCG [hpc] est un benchmark qui va reproduire cette situation, par là-même mimant le comportement de nombreux codes scientifiques (voir table 3.22). Il résout le système $A\vec{x} = \vec{b}$, $\vec{x}$ étant l'inconnue, par la méthode du gradient conjugué. Dans le cas du HPCG, la matrice est creuse et le système est résolu de manière itérative. La particularité de ce benchmark est d'être limité par l'accès à la mémoire et de solliciter fortement le réseau.

13. Méthode BFS : Breadth-First Search ou recherche en largeur d'abord
14. Méthode SSSP : Single Source Shortest Path
15. Par exemple des matrices penta-diagonales où seules la diagonale principale et les deux sous-diagonales inférieure et supérieure sont non nulles. Ce type de matrice est généré par un schéma à 5 points sur un maillage structuré 2D.
16. Le format CSR est représenté par deux tableaux. Le premier contient toutes les valeurs non nulles les unes à la suite des autres et le deuxième contient les indices de début et de fin des zones non vides de chacune des lignes.

BFS		
# Rang HPCG	# Top500	nom
1	1	Summit
2	2	Sierra
3	18	K Computer
4	6	Trinity
5	7	AI Bridging Cloud Infrastructure

TABLE 3.22 – Les 5 premières entrées du classement du HPCG avec leurs positions dans le TOP500, liste de novembre 2018. On se référera à la table 3.23 pour une comparaison des valeurs de performances.

Comme le montre la table 3.23 l'écart entre la performance Linpack et HPCG est important et illustre que la plupart des codes traitant de données dispersées n'utilisent qu'une fraction de la performance théorique des machines.

Machine	rang	année	crête TFlop/s	HPL TFlop/s	HPL %crête	HPCG TFlop/s	HPCG %crête
Summit	1	2018	200794	143500	71.47	2925.75	1.46
Sierra	2	2018	125712	94640	75.28	1795.67	1.43
Sunway	3	2016	125435	93014	74.15	480.848	0.38
Tianhe-2A	4	2018	100678	61444	61.03	*NC*	*NC*
Piz Daint	5	2017	27154	21230	78.18	496.978	1.83
Trinity	6	2017	41461	20158	48.62	546.124	1.32
AI Bridging-C	7	2018	32576	19880	61.03	508.854	1.56
SuperMUC-NG	8	2018	26873	19476	72.47	207.844	0.77
Titan	9	2012	27112	17590	64.88	322.322	1.19
Sequoia	10	2011	20132	17173	85.30	330.373	1.64

TABLE 3.23 – Comparaison des performances HPL et HPCG des 10 premières machines du TOP500 de novembre 2018. Le Sunway est l'archétype de la conception se focalisant sur la puissance de calcul brut au détriment de la performance des codes.

Les sections sur le TOP500, le Green500, le Graph500 et le HPCG montrent qu'aucune de ces 4 mesures suffit à elle seule à caractériser un supercalculateur. Dans un monde parfait, le classement du TOP500 devrait se retrouver dans les trois autres listes indiquant ainsi que la machine est performante pour le calcul pur mais qu'elle est aussi efficace pour l'utilisation de sa mémoire et qu'elle est économe. Les tables 3.20, 3.21 et 3.22 montrent que dans le monde réel, un compromis doit être fait en fonction du type de production visée (machine généraliste comme au CEA ou machine spécialisée par exemple pour le traitement de bases de données), privilégiant une des caractéristiques au détriment des autres.

La communauté du HPC cherche encore le meilleur benchmark, faisant l'unanimité, qui permette de prendre en compte tous les aspects d'un supercalculateur ayant un impact sur la production des centres de calcul. La tendance actuelle est de disposer d'un banc de charge représentatif de la production locale, au détriment de la reproductibilité de celui-ci d'un centre de calcul à l'autre, qui permette de choisir la prochaine génération de supercalculateur réellement adapté aux besoins.

Au delà de la mesure globale de la performance de l'ordinateur, il est indispensable de mettre en place, dès l'origine des développements, les outils de mesure de la performance du code et ce de manière **pérenne**. L'accès au nombre d'opérations flottantes est souvent difficile et fonction du compilateur utilisé. Il est préférable d'utiliser une caractéristique propre du programme, comme par exemple le nombre de cellules ou d'atomes traités par seconde. Cette mesure permettra de vérifier dans le temps que des évolutions du codage ne viennent pas perturber la performance du programme et permettra de quantifier l'évolution de l'efficacité des versions successives. La caractéristique choisie sera un des indicateurs utilisés dans le banc de charge évoqué plus haut.

– ♦ ∞ ♦ ∞ ♦ ∞ ♦ ∞ ♦ ∞ ♦ ∞ ♦ –

4 Conclusion

Pour conclure cette première partie, si l'on reprend les 7 nains de Colella, leurs caractéristiques, au regard des différentes machines décrites, peuvent se résumer ainsi :

1. les algorithmes sur grilles structurées : régularité des données favorisant les opérations vectorielles ou sur GPU. Dans le cas particulier des grilles AMR, la régularité et la vectorisation sont possibles au prix d'un certain effort de codage ;
2. les algorithmes sur grilles non structurées : données très irrégulières sollicitant fortement les systèmes mémoires et reposant sur la performance en latence de ceux-ci ;
3. l'algèbre linéaire dense : à la base du Linpack, cette classe d'algorithme sollicite au mieux les capacités calculatoires des processeurs ;
4. l'algèbre linéaire creuse : la représentation des données est une difficulté importante pour obtenir un code très vectorisé, l'aspect creux conduisant forcément à des indirections et à la forte sollicitation du système mémoire ;
5. les transformées de Fourier rapides : les FFT 2D et 3D passent mal à l'échelle en multinœud à cause des transpositions de matrices mises en jeu (de type `MPI_Alltoall`) ;
6. les méthodes particulaires : par manque intrinsèque de structuration, les méthodes particulaires génèrent beaucoup d'indirections ;
7. les méthodes de Monte Carlo : ces méthodes au caractère aléatoire génèrent des branchements difficiles à prévoir par le matériel. Ils sont donc difficiles à vectoriser ou porter sur GPU.

Au final, nous retiendrons que l'architecture des supercalculateurs a un impact fort sur les codes que ce soit sur l'architecture du code, ses structures des données et sur les algorithmes employés.

Si l'on recherche la performance portable des logiciels, de nombreuses questions se posent. Quelles sont les bonnes méthodes à retenir ? Quels langages de programmation ? Quelle pérennité des développements ? La partie suivante va tenter d'apporter quelques éléments de réponses à ces questions aux travers de mes recherches effectuées tant sur les codes que pour les outils pour le graphique.

– ♦ ∞ ♦ ∞ ♦ ∞ ♦ ∞ ♦ ∞ ♦ ∞ ♦ –

Points de vigilance
- L'extensibilité réelle du programme
- La compatibilité avec les mécanismes de cache (localités spatiale et temporelle)
- La régularité des accès, si possible contigus (CPU et GPU)
- La vectorisation du code CPU

Bonnes pratiques
- Penser le parallélisme multi niveaux (inter / intra / SIMD ou GPU) dès l'origine du projet (MPI + OpenMP vectorisé)
- Réduire la proportion de codes séquentiels ou les sérialisations par un bon usage des synchronisations (loi d'Amdahl)
- Bien concevoir ses structures de données en pensant performance
- Éviter la structure maître-esclave qui ne passera pas à l'échelle
- Mettre en place dès l'origine les outils de mesure de la performance

– ♦ ∞ ♦ ∞ ♦ ∞ ♦ ∞ ♦ ∞ ♦ ∞ ♦ –

Deuxième partie

Recherche de l'adéquation matériel - logiciel

5 Contributions aux Codes de calcul

DANS la première partie de ce manuscrit ont été donnés quelques rappels sur les codes et une description des supercalculateurs que j'ai été amené à utiliser. Cette deuxième partie se place dans un premier temps sous l'angle des codes de calcul, tels que définis dans l'introduction générale, et en particulier dans la perspective de mes contributions à ceux-ci, puis, dans un deuxième temps sous l'angle des outils de dépouillement orientés graphique.

J'ai défini un code de calcul comme étant un logiciel complexe de grande taille. Les sources d'un grand code de calcul font souvent plusieurs millions de lignes de FORTRAN, de C et de C++. Cela inclut la partie résolvant les équations d'intérêt mais aussi toutes les bibliothèques nécessaires au bon fonctionnement de celui-ci (entrées-sorties diverses, constantes physiques, outils mathématiques de base...). C'est tout particulièrement le cas au CEA/DAM où les codes traitent de phénomènes physiques fortement couplés, présentant des dynamiques extrêmes , sur un très grand nombre d'itérations [1]. Le développement de tels codes prend de nombreuses années (en moyenne 10 ans), auxquelles il faut ajouter la période de validation par confrontation aux résultats expérimentaux (environ 5 ans) avant que le code ne soit déclaré apte à la production. Cette dernière phase peut durer une vingtaine d'années.

La durée de vie des codes du CEA/DAM implique de faire des choix (architecture, langages de programmation, algorithmes) qui devront rester pertinents pendant de nombreuses années car il est impossible de réécrire un code tous les 5 ans, durée de vie d'un supercalculateur, au vu du nombre de lignes à modifier. Une étude américaine [CM15] a cherché à mesurer la productivité effective des développeurs. Dans le contexte considéré (application de défense américaine), Clark évalue que 100 000 lignes de code source utiles [2] représentent 630 homme.mois de développement, soit environ 160 lignes par mois et par développeur ! Le potentiel de développement pour un code scientifique est très probablement dans ces ordres de grandeur, bien que jamais effectivement mesuré.

Relever cette gageure est difficile, surtout à l'approche de situations de rupture comme cela sera détaillé dans la partie III (page 89) développant les perspectives en matière de supercalculateurs, et demande beaucoup d'expérimentations pour anticiper les évolutions pouvant affecter les codes de calcul. L'apport de l'informaticien – un programmeur/expert HPC – sera donc de trouver les meilleures conditions logicielles et matérielles pour que le numéricien – un spécialiste en analyse numérique et schémas associés – puisse s'exprimer pleinement.

Dans ce chapitre, je m'intéresserai particulièrement à mes travaux dans le monde des codes de calculs, depuis ma première rencontre avec le monde de la vectorisation jusqu'au développement de codes massivement parallèles sur GPU en gardant comme constante la recherche de la performance portable.

1. Imposant de fait l'utilisation de processeurs ou de GPUs capables de traiter nativement des flottants en 64 bits.
2. Une ligne utile correspond à une instruction qui est complètement déverminée, testée et documentée.

5.1 Code laser

Le développement du code DXFCI2 a été mon activité de départ dans le monde professionnel. Cette première réalisation va jeter les bases de mon avenir : sensibilité aux besoins code (connaissance du matériel sous-jacent, programmation visant la performance), utilisation du monde du graphique haute performance pour comprendre et anticiper ce que devront être les architectures du futur.

5.1.1 Description du code DXFCI2

Le code DXFCI2 [G C88], développé sur CRAY 1 puis CRAY XMP, est un code de simulation des diagnostics X utilisés dans les expériences laser pour produire des images (dans la gamme des rayons X) de l'interaction des faisceaux lasers avec la cible étudiée. Ce code prend en entrée les résultats produits par un code simulant l'interaction laser – cible (le FCI2 qui est un code de **f**usion par **c**onfinement **i**nertiel bi-dimensionnel (2D) structuré, selon la classification des 7 nains du HPC présentée au chapitre 2). Un schéma de maillage utilisé par le FCI2 est donné figure 5.1.

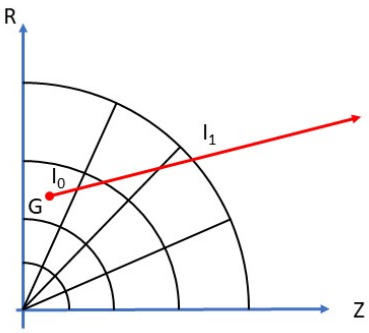


FIGURE 5.1 – Discrétisation du code FCI2 : le code fait ses calculs sur un maillage 2D. Celui-ci est de type RZ c'est à dire qu'il est de révolution autour de l'axe Z. L'axe OR est un plan de symétrie. Le quart de cercle de la figure est donc en réalité une sphère. Chaque maille, représentée par un quadrangle, est de fait un tore de révolution autour de Z. Le point G représente le centre de gravité de la maille et sert de point de départ au rayon (I_0) en direction du sténopé (I_1 en sortie de cible).

Le code DXFCI2 a permis de restituer des images expérimentales. Une telle expérience est décrite par la figure 5.2 et un exemple de restitution est illustré par la figure 5.7. Le couple FCI2 – DXFCI2 a été utilisé soit pour interpréter des expériences soit pour en dimensionner et prédire de nouvelles.

Le code DXFCI2 va produire en sortie une image représentant une photo du phénomène vu à travers un petit trou (appelé un sténopé) dont le principe est représenté sur la figure 5.3, soit simuler une caméra à balayage de fente, appelée « fentastix », qui permet de suivre le déroulement du phénomène le long d'une direction privilégiée matérialisée par la fente (figure 5.4). Le fentastix produit une image synthétique (figure 5.8) qui représente l'évolution de l'image d'une fente en fonction du temps.

Le code DXFCI2 cherche à évaluer I_2, intensité radiative émise par la cible arrivant sur le film (équations 5.1) pour tous les rayons [numériques], chaque rayon partant des centres de gravité de chacune des mailles et passant par le sténopé puis par un atténuateur évitant la destruction du film, et ceci pour chacun des groupes de fréquence discrétisant l'énergie radiative [3].

$$I_0 = \frac{C}{4\pi}(\mu_p^k a T^4 b_k + \sigma_{th} n_e E_k), \quad I_1 = I_0 e^{(-\sum_i (\mu_p^k + \sigma_{th} n_e)\lambda_i)}, \quad I_2 = I_1 e^{-\lambda_{att}} \tag{5.1}$$

3. Le spectre d'émission est découpé en N groupes de fréquence pour le calcul.

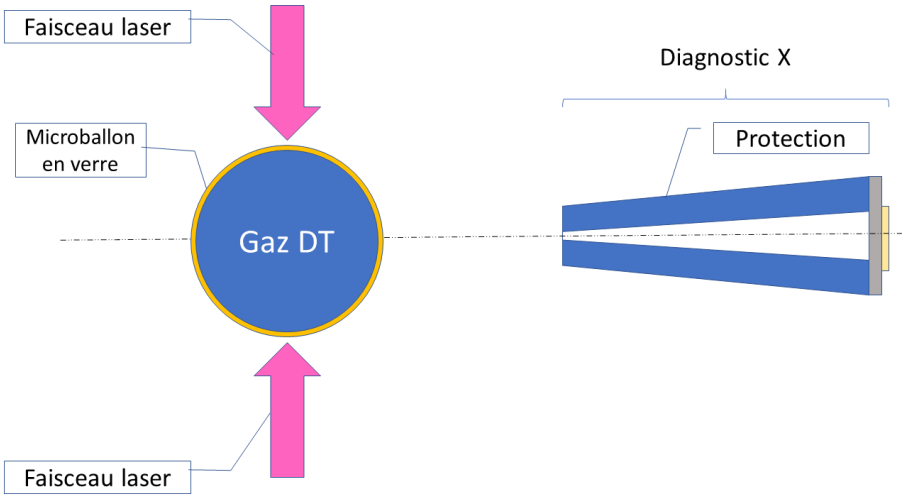


FIGURE 5.2 – Cette figure est une coupe d'une expérience laser simulée par le couple FCI2 – DXFCI2. Deux faisceaux laser vont déposer brutalement de l'énergie sur un microballon de verre contenant du gaz Deutérium Tritium (DT) et le faire imploser. La compression du gaz résultant de cette implosion va faire monter la température du gaz jusqu'à atteindre (éventuellement, selon conditions expérimentales) le point d'inflammation du mélange ou fusion nucléaire. Une image de l'énergie radiative émise par la cible est réalisée au moyen d'un équipement spécial appelé diagnostic X. Ce dernier peut être un sténopé ou un fentastix.

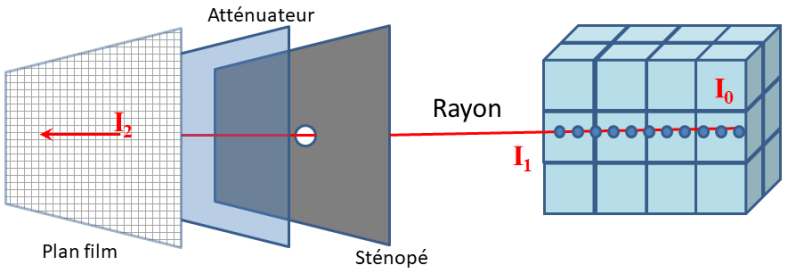


FIGURE 5.3 – Principe d'un sténopé : selon le même principe qu'un appareil photo, l'image de la cible est produite à travers un petit trou. Mais comme le rayonnement est dans la gamme de X et non pas dans le visible, il est nécessaire de protéger le film (en fait un détecteur CCD) d'une trop forte concentration d'X qui produirait une saturation, voire une destruction du détecteur, par un atténuateur permettant de sélectionner une gamme de longueur d'ondes. Le détecteur est simulé par une grille d'où l'effet de pixélisation des images numériques.

Les différents paramètres permettant de calculer I_2 sont : k groupe de fréquence considéré, i mailles traversées du centre de gravité au centre du sténopé, λ_i distances parcourues dans chaque maille i, μ_p^k opacité de Planck pour le k-ième groupe, σ_{th} section efficace de diffusion Thomson, n_e la densité électronique, T la température de la maille, b_k intégrale sur le groupe k de la fonction de Planck réduite, E_k la densité d'énergie radiative du groupe k dans la maille et enfin λ_{att} la profondeur optique de l'atténuateur utilisé. Ces différentes variables proviennent de la physique des plasmas, sujet hors de propos dans ce manuscrit.

La particularité du DXFCI2 est de traiter d'une géométrie 3D reconstituée à partir de la géométrie 2D d'entrée provenant du code FCI2. Le maillage résultant ne pouvait pas tenir en mémoire centrale s'il était stocké. Le code utilisait donc une matérialisation temporaire des mailles 3D (illustrée par la figure 5.5) se reposant sur la méthode de découpage des tores par pas constant.

Le suivi des rayons présente une difficulté liée à la prise de décision dans des cas pathologiques comme

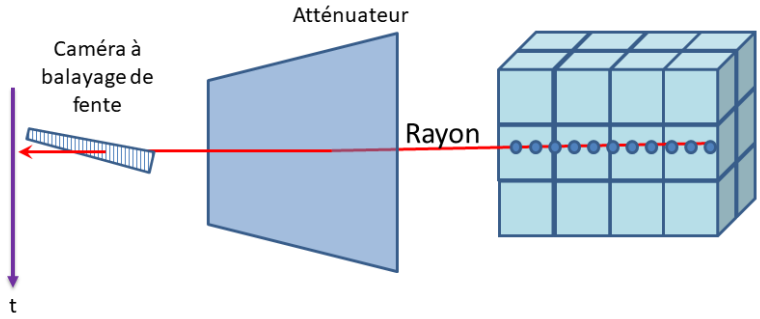


FIGURE 5.4 – Principe d'un fentastix : contrairement au sténopé, le détecteur du fentastix est une ligne de détecteurs qui enregistre des images à intervalle de temps régulier. Chaque ligne d'image collée à la suite l'une de l'autre constitue une image synthétique qui permet de suivre l'évolution temporelle des zones les plus émissives de la cible dans une direction choisie. Là encore, la présence d'un atténuateur est indispensable.

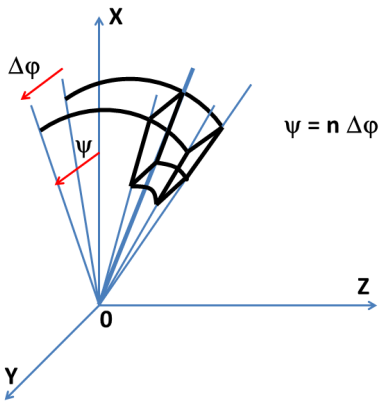


FIGURE 5.5 – Discrétisation du code DXFCI2 : le code travaille sur un maillage 3D reconstitué par découpage des tores par pas de $\Delta\phi$.

dans le cas où un rayon passe sur une frontière de maille (figure 5.6) rendant impossible le passage à la maille suivante sans faire de choix, donc des tests. On retrouve cette situation dans tous les codes effectuant un suivi de trajectoires (particules, rayons lumineux, ...) et particulièrement dans les codes où la trajectographie est de type Monte-Carlo où, de surcroît, la direction de sortie résulte d'un processus aléatoire.

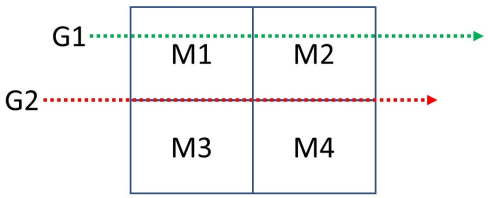


FIGURE 5.6 – Cas pathologique classique d'une trajectographie : un rayon passe dans l'axe de la séparation entre les mailles. Il est donc impossible de savoir dans quelle maille il va entrer (cas du rayon rouge). Le cas du rayon vert est beaucoup plus simple, surtout dans le cas structuré qui est celui du DXFCI2.

Celle-ci impacte la vectorisation car elle introduit des tests dans l'algorithme. Dans le cas du DXFCI2, l'utilisation des fonctions intrinsèques de type `cvmgt`, évoquée en page 32, a permis de contourner la difficulté au prix d'une certaine lourdeur de codage. Il est intéressant de constater que cette difficulté reste d'actualité avec les GPUs les plus récents.

De cette première expérience j'ai retenu la difficulté de programmer une algorithmie complexe en conservant portabilité et performance. J'ai été sensibilisé au fait qu'il était nécessaire de bien comprendre l'architecture du supercalculateur utilisé, constat qui ne s'est pas démenti depuis.

L'utilisation d'un compilateur faisant de la vectorisation automatique ne permet pas de vectoriser toutes les boucles sans (un peu d') aide du programmeur. Cette constatation, valable en 1988, est toujours d'actualité en 2019. Avec 30 ans de recul, je constate donc que l'écriture d'un bon compilateur reste un art très difficile et que toute solution matérielle se reposant sur le compilateur pour produire des logiciels performants est une chimère comme l'ont démontré l'Itanium et le KNC d'Intel. Au contraire le matériel doit prendre en charge une partie des difficultés comme par exemple en utilisant l'out-of-order (OoO) versus l'in-order (voir page 34).

Le développement du DXFCI2 a eu aussi une conséquence inattendue : mon introduction au monde du graphique. Les images produites par le code étaient imprimées sur des moyens maintenant disparus pour la plupart (diapositives par exemple) qui ont nécessité des développements spécifiques. J'ai donc assez naturellement basculé dans ce nouveau monde, comme cela sera détaillé dans le chapitre 6.

Les acquis en parallélisme, abordés au travers de mes recherches en graphique, m'ont fait me pencher à nouveau sur les codes de calcul avec comme objectif la haute performance (voir la figure 1.2). C'est ce dont je vais traiter dans la section suivante par soucis de cohérence thématique.

Image expérimentale

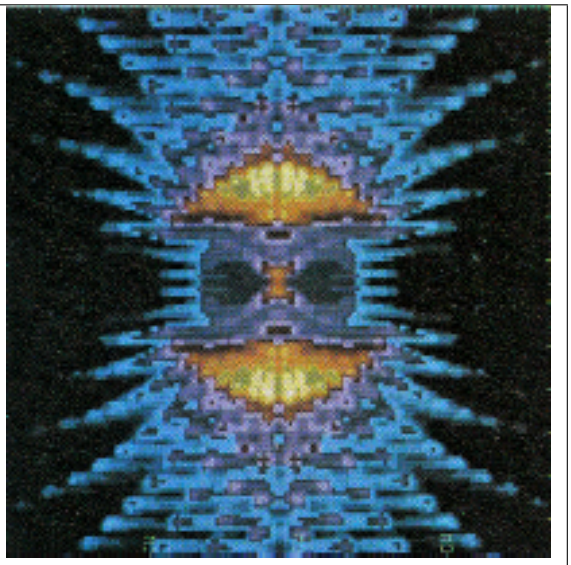


Image calculée par le DXFCI2

FIGURE 5.7 – Simulation d'un sténopé : comparaison du calcul et de l'expérience démontrant la bonne restitution des phénomènes physiques par la chaîne de calcul FCI2 + DXFCI2. L'expérience reproduite est l'implosion d'un micro-ballon contenant du gaz DT par l'attaque de deux faisceaux laser arrivant du nord et du sud sur l'image. L'image représente la somme de l'énergie émise par la cible au cours de l'expérience. On a donc superposition de plusieurs étapes de l'expérience : attaque du micro-ballon par les lasers puis mise en vitesse du micro-ballon et enfin émission par combustion du gaz DT. Ces différentes phases sont mises en évidence par le diagnostic dit fentastix, illustré sur la figure 5.8.

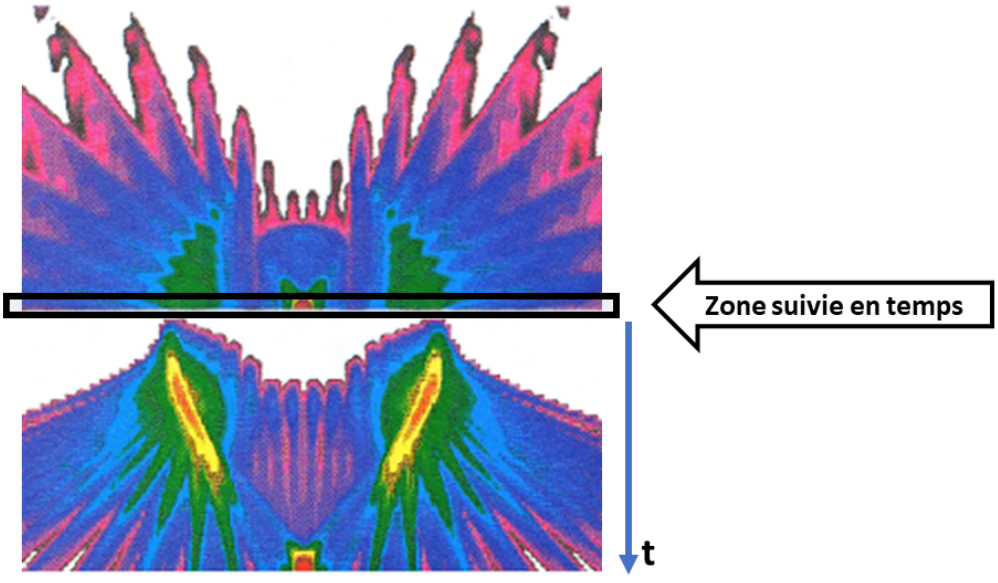


FIGURE 5.8 – Simulation d'un fentastix dans la partie basse de l'image et du 1/2 sténopé correspondant sur la partie haute. Le fentastix permet de voir quelles zones de la cible sont émissives durant l'expérience. On regarde la cible au travers d'une fente et au cours du temps. Le mouvement de la matière, lors de l'implosion du micro-ballon, est bien visible. Le centre de la cible n'émet que tardivement mais la source est très intense ce qui explique que le centre du sténopé soit le plus rouge.

5.2 Vers le massivement parallèle

Les codes de production du CEA/DAM ont une durée de vie d'environ 30 ans, soit au moins 5 générations d'ordinateurs si l'on compte une durée de vie moyenne de 6 ans par machine. Il est donc indispensable de réfléchir très tôt à ce que pourraient être les ordinateurs de demain (voire d'après-demain), d'en mesurer les implications tant en terme d'architecture de supercalculateur mais surtout de codes. Pour ces derniers, plusieurs aspects sont à considérer. Il faut entre autres répondre aux questions suivantes : les algorithmes seront-ils toujours les mieux adaptés, les structures des données actuelles seront-elles les plus pertinentes, les langages de programmation sont-ils pérennes, pourra-t-on facilement programmer des codes passant à l'échelle du pétaflop [4] puis de l'exaflop [5] ?

Avec ces questions comme trame, j'ai essayé de voir si les GPUs étaient une voie possible, comment les programmer et pour quels bénéfices. De plus, je me suis intéressé à la manière de développer des codes pour atteindre les meilleures performances tout en conservant la portabilité des programmes dans le temps. L'objectif de ce chapitre est de montrer les résultats acquis au travers des diverses publications réalisées sur ces sujets.

5.2.1 L'approche GPU

Dès 2007, les premiers prototypes incluant des GPU ont montré que l'on était face à une potentielle révolution en matière de calcul à hautes performances. Cette révolution a été rendue possible par l'introduction de CUDA [cud] par NVIDIA (annonce du 15 février 2007) qui apportait un environnement de programmation plus simple que ce qui avait été tenté auparavant en utilisant des langages dédiés au graphique tels qu'OpenGL[opec], comme décrit par Strzodka [SDK05].

L'arrivée de la machine Titane, première plate-forme européenne de production à disposer de GPU, a offert aux développeurs une réelle occasion de tester l'utilisation des GPUs à l'échelle. Mon article [G C09], publié dans High Performance Computing on Vector Systems de 2009, décrit la stratégie mise en place au CEA à cette époque :

- choix du matériel NVIDIA par ses capacités de calcul, la disponibilité de plusieurs langages de programmation et son intégrabilité dans une machine de production ;
- expérimentation des langages de programmation pour GPU en comparant CUDA, HMPP [DBB07], RapidMind [Rap] et OpenCL sur un ensemble de mini-applications. On remarquera qu'en 2019, HMPP et RapidMind n'ont pas survécu au temps ;
- mise en place de formations pour diffuser le savoir faire GPU.

J'ai publié dans les Comptes Rendus de Mécanique de l'Académie de Sciences [G C11] de 2011 une première synthèse des résultats acquis au bout de 2 ans. Sa principale conclusion n'est pas de savoir **si** les développeurs de code devront passer à l'utilisation des GPUs mais bien **quand** ! Néanmoins, cet article attire déjà l'attention du lecteur sur les points suivants, dont certains restent toujours d'actualité :

- les deux seuls acteurs sérieux du monde des cartes graphiques sont NVIDIA et AMD (qui a acquis ATI et ses GPUs). Intel n'est pas focalisé sur les GPUs en 2011 (mais le sera en 2020 – annonce faite en 2018) ;
- l'environnement logiciel de 2011, gravitant autour de CUDA, se développe bien par l'addition de bibliothèques optimisées telles que CuFFT ou CuBLAS. En 2019, cet environnement est beaucoup plus riche car l'utilisation des GPUs pour l'intelligence artificielle s'est développée entre temps ;
- en 2011, les langages de programmation disponibles sont CUDA, OpenCL – le favori d'AMD – et HMPP qui mature bien ;

4. 10^{15} opérations flottantes par secondes (FLOPS)
5. 10^{18} FLOPS

- les développeurs ont besoin d'outils pour trouver les bugs et analyser les performances des programmes. Des progrès restent à faire.

Cet article fait un certain nombres de remarques visant plus spécifiquement les codes et esquisse la méthodologie de portage des applications sur GPU, lesquels seront abordés un peu plus loin. Les programmes très orientés objets ainsi que les programmes à profil plat seront difficiles à porter sur GPU. Le profil d'un code est l'histogramme des temps effectifs passés dans chacun des sous-programmes. Un programme à profil plat est un programme où aucun sous-programme ne prend de temps significatif d'exécution, donc présentant un histogramme plat. Il est alors difficile de savoir où commencer l'optimisation / le portage sur GPU sans une réorganisation (souvent profonde) du logiciel. De même, les codes utilisant beaucoup d'indirections n'utiliseront pas les GPUs de manière efficace, ces derniers ayant besoin de régularité des données manipulées. Enfin, il est fort probable qu'il sera nécessaire de revisiter la méthode numérique employée comme par exemple favoriser les montées en ordre qui augmentent significativement le nombre d'opérations effectuées sur les données, utilisant à plein les capacités des GPUs. Dans tous les cas, porter un gros code (de l'ordre du million de lignes de source) sera coûteux en temps et en ressources humaines.

Les expérimentations m'ont conduit à l'élaboration de la méthodologie de portage suivante :

1. isoler les sous-programmes représentant une portion significative du temps calcul. En règle générale, 80 % du temps calcul est passé dans 20 % du code source. Il est donc souvent inutile de se pencher sur le reste. Dans le cas où il n'est pas possible d'isoler ces 20 %, il est recommandé de restructurer le code, ce qui peut être difficile. En vertu de la loi d'Amdahl [Amd67], le gain de performance maximal possible du code est fonction du poids de la partie non accélérée. Cette première étape est indispensable pour traquer les parties du code purement séquentielles, surtout si l'on cherche une extensibilité forte du code ;
2. transformer les routines ciblées pour les adapter au GPU. Cette étape se fera en négligeant temporairement les temps de transfert de données. Cette étape est souvent frustrante car on observe un ralentissement notable du code par excès de mouvement de données ;
3. par une étude approfondie des données du code, optimiser les mouvements de données vers et depuis le(s) GPU(s). L'objectif, ici est de faire en sorte de laisser les données actives (ou chaudes) le plus longtemps possible sur le GPU. Un exemple de directives modernes de placement/mouvement des données (OpenMP 4.5) est donnée en exemple dans le tableau 5.1 ;
4. s'il est possible d'utiliser des bibliothèques en lieu et place d'un codage manuel, privilégier celles qui existent aussi sur GPU. On trouvera classiquement les FFT, les générateurs de nombres aléatoires, les routines d'algèbre linéaire, etc.

Le cas des bibliothèques externes applicatives peuvent se révéler un obstacle majeur pour l'utilisation des GPUs. En effet, elles peuvent soit ne pas exister sous forme GPU soit manipuler des données beaucoup trop volumineuses. Ce dernier cas est fréquent avec les constantes physiques nécessaires pour décrire les états de la matière comme les opacités ou les sections efficaces de neutronique ou toute autre quantité finement tabulée. Le cas le plus favorable est celle où les fonctions de la bibliothèque peuvent être insérées directement dans le programme appelant (sous forme d'« `include` » en C par exemple).

Dès cette époque, il apparaissait nettement que la migration des codes du patrimoine (« legacy codes ») sur GPU serait un défi majeur.

La méthodologie précédente a été testée sur un code de physique simulant l'interaction d'un faisceau laser avec le verre d'une lentille, telle qu'elle est utilisée dans un laser de puissance comme le laser mégajoule du CEA/DAM. Cette étude a été récompensée par le deuxième prix Bull Joseph Fourier de 2013 et décrite dans les publications [BG11], [Ber+12] et [Mau+13] ainsi que dans la thèse de Mme Sarah Mauger [Mau11] dont j'ai suivi les travaux spécifiques aux GPUs et pour laquelle j'ai été membre invité du jury.

```
1  extern void init(float*, float*, int);
2  extern void output(float*, int);
3  void vec_mult(float *p, float *v1, float *v2, int N)
4  {
5          int i;
6          init(v1, v2, N);
7          #pragma omp target data map(to: v1[0:N], v2[:N]) \\
8                                  map(from: p[0:N])
9          {
10                 #pragma omp target
11                 #pragma omp parallel for
12                 for (i=0; i<N; i++)
13                         p[i] = v1[i] * v2[i];
14         }
15         output(p, N);
16 }
```

TABLE 5.1 – Ce petit exemple, tiré de la documentation officielle d'OpenMP (`https://www.openmp.org/wp-content/uploads/openmp-examples-4.5.0.pdf`), illustre la directive `map` (ligne 7 et 8) qui indique dans quelles directions copier les données.

Dans cette étude, nous voulions voir si les GPUs pouvaient apporter un gain significatif en temps de simulation et quelle était la meilleure option de programmation (efficacité versus temps de développement). Et naturellement, en cas de gain, accéder à des résultats de physique plus précis car obtenus sur des simulations de plus grandes tailles. Ce fut donc l'occasion de mettre en pratique nos acquis et de comparer une version CUDA à une version HMPP, ces deux approches étant radicalement différentes.

Pour le portage en CUDA, la difficulté majeure était que le code d'origine était écrit en FORTRAN. Il a été nécessaire de réécrire des portions significatives du code en C/C++ et créer les interfaces entre les deux langages. Au final, le code est passé d'environ 900 lignes à 5500 lignes soit une inflation significative. Pour HMPP, il n'a fallu ajouter que 198 lignes de directives au FORTRAN, montrant tout l'intérêt de la programmation par annotation.

Le tableau 5.2 illustre les gains de performances obtenues. Même si les gains HMPP ne sont pas tout à fait ceux de CUDA, l'investissement réduit en développement par l'utilisation des directives est largement récompensé par une réelle accélération de l'application.

# rangs MPI	Temps CPU	Temps CUDA	Gain	Temps HMPP	Gain
1	23318 s	2296 s	10.16	2611 s	8.93
2	12005 s	1208 s	9.94	1421 s	8.45
4	6784 s	655 s	10.36	845 s	8.03
8	3453 s	379 s	9.11	536 s	6.46
16	1778 s	240 s	7.41	362 s	4.91
32	931 s	168 s	5.54	288 s	3.23

TABLE 5.2 – Performances comparées de la version CUDA et de la version HMPP du code SBS pour un nombre croissant de tâches MPI. Chaque rang MPI dispose d'un GPU NVIDIA Tesla S1070. Le temps donné est le temps total d'exécution.

L'analyse détaillée des gains par fonction présentée dans l'article, montre que l'on peut obtenir des accélérations importantes localement. Par exemple pour la fonction FLUX, on observe un gain de 68.2 X en CUDA et 86.6 X en HMPP par rapport au temps de référence CPU. Dans d'autres cas, l'avantage est à CUDA : pour la fonction KERSBS le gain est de 44.7 X en CUDA pour 32.6 X en HMPP. De tels gains valident l'intérêt des GPUs pour une utilisation en calcul scientifique de type HPC.

En 2012, les meilleurs choix possibles pour programmer étaient donc CUDA, OpenCL [opeb] ou HMPP.

En 2019, seuls CUDA et OpenCL sont encore présents. HMPP a été remplacé par OpenACC, qu'il a influencé, ou par OpenMP [oped] 4.5 (et bientôt 5.0 qui a été annoncé à Supercomputing de novembre 2018). Compte-tenu de la durée de vie de nos codes mentionnées en introduction de ce chapitre, la question du choix du langage de programmation, donc de sa pérennité et de sa portabilité, s'est rapidement posée. A ces questions s'est adjointe celle sur la manière de transformer les codes pour viser la plus haute performance. La section suivante va tenter d'apporter des éléments de réponses à ces questions, toujours ouvertes, au travers de deux études et d'une synthèse.

5.2.2 La performance portable

Pour tester la vraie portabilité d'un code, il est nécessaire d'utiliser des architectures radicalement différentes. Les deux études qui suivent ([BDG13] et [GS]) sont parties de la même mini-application HydroC [6] que je développe depuis 2008 pour tester différents langages et toutes les machines auxquelles je peux avoir accès. Le code d'origine est la mini-application « hydro » extraite du code Ramsès 2D développé par Romain Teyssier [Ram]. Hydro est écrite en FORTRAN et maintenue par Pierre-François Lavallée de l'IDRIS. J'en ai fait une version C (d'où son nom HydroC) pour tester CUDA, HMPP et OpenCL et C++ pour tester OpenMP. Romain Dolbeau a contribué au développement de la version OpenACC. Cette mini-application se prête bien à ce genre d'exercice par sa compacité (environ 5000 lignes) bien que se comportant comme une vraie application (car disposant de sorties graphiques et plus récemment d'un mécanisme de protection-reprise dans les dernières versions C++). Une partie de ce travail a été réalisée dans le cadre de PRACE [7] First Implementation Phase [1IP] (2010 - 2012) et a fait l'objet d'un rapport [Lav+12] détaillant les divers portages réalisés.

OpenCL est apparu très vite comme l'alternative possible à CUDA car non propriétaire et adaptée à de nombreux matériels. L'article [BDG13] « One OpenCL to Rule Them All ? » de 2013 étudie OpenCL sous l'angle de la portabilité de la performance en comparant le comportement de la mini-application HydroC sur plusieurs types d'architectures (un GPU d'AMD, un KNC et un NVIDIA K20C) dans l'optique particulière d'utiliser OpenCL en « backend » [8] d'un codage OpenACC. L'idée de base repose sur la technologie utilisée par le compilateur OpenACC de CAPS dite de conversion source à source. Le code OpenACC d'origine est transformé en un nouveau source OpenCL spécialement adapté à l'architecture ciblée. Par ce mécanisme, on peut éviter au programmeur une multitude de variantes spécifiques aux différents matériels rencontrés, charge à OpenCL de produire le meilleur code possible pour la cible. Nos résultats démontrent la validité de cette approche si l'on considère qu'une variation de 12 % dans les performances est un écart acceptable, ce qui sera le cas dans la majorité des situations car les utilisateurs sont plus sensibles à des gains en centaines de pourcents qu'à ce genre d'écart.

Gabriel Noaje a cherché un chemin similaire lors de ses travaux de thèse [Noa13], pour laquelle j'ai été membre invité du jury, en essayant de convertir automatiquement du code OpenMP en CUDA optimisé. Il a focalisé ses efforts sur les nids de boucles reposants sur les directives OpenMP **#pragma omp parallel for**, un parallélisme à grain fin. Ces travaux seraient à étendre pour prendre en compte les tâches OpenMP, construction de plus en plus utilisée dans les codes pour mettre en place un parallélisme à gros grain.

A ce jour, OpenCL n'a pas su remplacer CUDA et tarde à décoller. Plusieurs facteurs sont en cause. Tout d'abord OpenCL est surtout poussé par AMD qui reste un acteur minoritaire dans le monde du

6. `https://github.com/HydroBench/Hydro/tree/master/HydroC`
7. `http://www.prace-ri.eu/`
8. Une **compilation** passe 3 étapes : un « **frontend** » se charge de lire un langage de programmation donné et produit une représentation intermédiaire du programme, un optimiseur applique des transformations d'optimisation du programme indépendantes de l'architecture cible et un « **backend** » génère le langage machine adapté à la machine où sera exécuté le programme. Le backend prend en charge les optimisations spécifiques à la machine cible.

calcul sur GPU. Ensuite la conception d'OpenCL fait qu'il y a une séparation forte (au niveau du texte du source) entre le programme appelant et le code OpenCL appelé, au contraire de CUDA qui offre une vue unifiée des deux. Cela ne facilite ni la maintenance ni le déverminage des programmes. Peut-être que SYCL [syc] (bonne intégration d'OpenCL dans un source de type C++) sera la réponse pour le développement d'OpenCL ? Néanmoins, OpenCL est présenté comme très utile pour programmer les FPGA et dispose enfin d'un compilateur Open Source (llvm [llv]).

L'approche que je viens de décrire est à bas niveau car prise en charge par le compilateur, en espérant une implication minime du développeur. Une autre approche consiste à se placer du côté du programmeur et voir si l'on peut imaginer une approche de la programmation qui puisse réconcilier vectorisation sur CPU et programmation efficace de GPU, donc produire un source portable, même au prix d'un peu de C++ complexe. D'un premier abord, les deux notions peuvent paraître contradictoires. En 2018, la publication [Ote+18] propose d'en faire la preuve sur une mini-application d'aérodynamique reposant sur la méthode dite de Galerkine Discontinue sur grille structurée. Cette méthode a la particularité d'être très gourmande en calcul donc, *a priori*, favorable tant à la vectorisation qu'à l'usage d'un GPU. Pour cela nous avons proposé une méthode de codage hybride dérivée de l'implémentation GPU qui est la plus contraignante à concevoir.

2D CUDA-GPU code (GPU.cu)

```
#define VSIZ 1
//Include the kernel
#include "kernel.h"
void __global__ gpu_function(
  double *__restrict__ val)
{
  //Thread indexes
  int tidx = ...;
  //Logical condition
  //to make computations
  if(tidx<Nx)
  {
    kernel<VSIZ>(val,tidx);
  }


}
```

Common vectorized kernel (kernel.h)

```
template<const int VSIZ>
//Conditional compilation
#ifdef DEFGPU
__device__
#endif
__inline void kernel(
  double *__restrict__ val,
  const int tidx)
{
  //Vectorized loop
  #pragma vector always
  #pragma unroll
  for(vec=0; vec<VSIZ; vec++)
  {
    ...
    val[VSIZ*tidx+vec] = ...;
  }
}
```

2D CPU code (CPU.cpp)

```
#define VSIZ 32
//Include the kernel
#include "kernel.h"
void cpu_function(
  double *__restrict__ val)
{
  //CPU loop
  for(int tidx=0;
      tidx<Nx;
      tidx+=VSIZ)
  {
    kernel<VSIZ>(val,tidx);
  }


}
```

TABLE 5.3 – Exemple de programmation hybride permettant de générer un exécutable GPU ou CPU selon l'option de compilation retenue par définition de la macro `DEFGPU`.

La table 5.3 montre un exemple de code extrait de l'application. Une simple compilation conditionnelle permet de générer un programme de type CPU ou GPU. L'utilisation de la directive `template` du C++ permet d'adapter automatiquement l'algorithme au bon paramétrage matériel, garantissant la meilleure efficacité du code spécialisé. Une telle approche a été vérifiée sur de nombreux matériels et a permis, entre autres, de comparer le KNL d'Intel au Pascal NVIDIA, ces deux options étant disponibles dans Tera 1000. La comparaison a été faite tant sur les performances que sur l'aspect énergétique qui sera abordé au chapitre 9.1. La limitation de la méthode décrite est qu'elle a été démontrée sur un code structuré qui se prête bien par nature à la programmation GPU. Il faudrait maintenant l'étendre à des codes non structurés.

La performance portable peut aussi être atteinte par une programmation prenant en compte les caractéristiques du matériel. La mini-application HydroC donne un bon exemple du gain que cette démarche peut apporter. Le parallélisme OpenMP est implémenté de deux manières différentes et les résultats sont comparés (voir table 5.4).

Dans un premier cas, le domaine de calcul est considéré comme une succession de bandes sur lesquelles sera appliqué le parallélisme de type OpenMP à grain fin (partie gauche de la figure 5.9). Chaque thread va donc traiter une ou plusieurs lignes complètes de maille selon la largeur de la bande souhaitée. Ce type de découpage est facile à implémenter et est tout à fait compatible avec une approche GPU.

Dans l'autre cas, le domaine sera découpé selon un pavage de tuiles (carrées) et chaque thread traitera

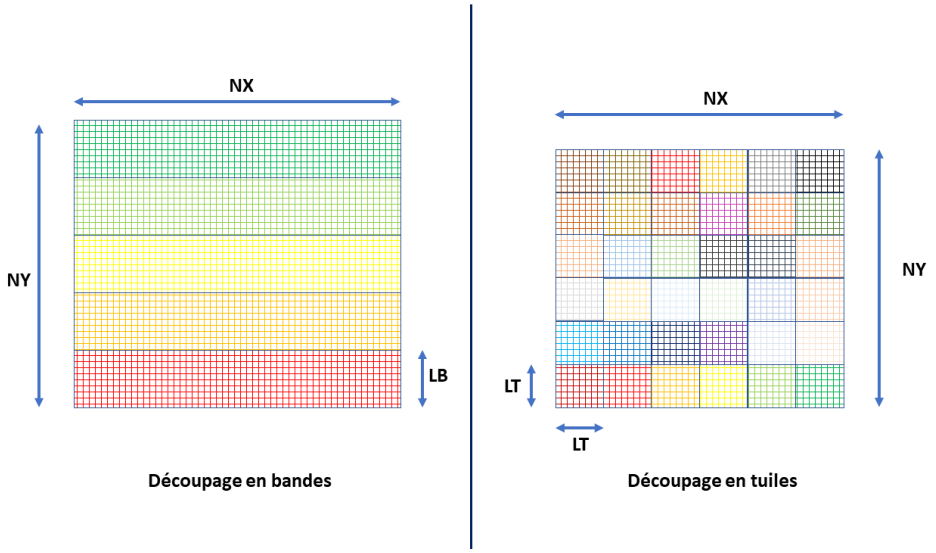


FIGURE 5.9 – Différents découpages possibles d'HydroC en threads. A gauche, le domaine est traité par bandes de largeur LB mailles, un multiple du nombre de thread. Le parallélisme OpenMP est donc réalisé au sein de chaque bande successivement. A droite, le domaine est découpé en tuiles carrées de largeur LT mailles. Chaque tuile est affectée à un thread OpenMP. Un thread traitera plusieurs tuiles.

Bande (v1)	MC/s: moy 48.693 min 32.723, max 53.099, sig 4.364
Tuile (v2)	MC/s: moy 96.891 min 43.849, max 105.67, sig 11.769
Tuile + Morton (v2b)	MC/s: moy 101.73 min 87.585, max 106.75, sig 5.236

TABLE 5.4 – Comparaison de performances de deux types d'implémentation d'HydroC. Machine de test : Intel(R) Xeon(R) Platinum 8168 CPU @ 2.70 GHz. Fréquence fixée à 1.90 GHz pour éviter les perturbations du mode turbo. Cas test réalisé avec 46 threads OpenMP, NX=NY=8192, LB=92, LT=64 (voir figure 5.9).

autant de tuiles que de besoin, maximisant la quantité de travail à réaliser par thread (parallélisme à gros grain). La taille des tuiles est un paramètre réglable, à optimiser selon les architectures – donc leurs tailles de cache. La mise en place de ce découpage implique de plus profondes modifications du code.

La mesure de performance (vitesse de calcul) est donnée en nombre de millions de cellules traitées par seconde (MC/S) et par itération. Le code reporte la vitesse de traitement moyenne, la vitesse minimale et maximale et l'écart type de ces vitesses. Pour éviter les incertitudes liées au démarrage du code, les 5 première mesures (itérations) sont ignorées. La taille du domaine (NX et NY) est choisie pour qu'une ligne (resp. colonne) de mailles ne rentre pas dans les caches. La fréquence du processeur est forcée à 1.9 GHz (au lieu des 2.7 GHz) pour ne pas activer le mode turbo qui introduit du bruit dans les mesures de performances, car il est impossible de contrôler son déclenchement (c'est une décision du matériel).

Nous constatons (table 5.4) que la version 1 est plus lente que la version 2, ce qui illustre l'intérêt à passer du parallélisme à grain fin au parallélisme à gros grain. Ceci s'explique par le fait que la v1 ne prend pas en considération le mécanisme de caches et donc doit constamment aller chercher les données en mémoire centrale, pénalisant les performances. Au contraire, la v2 travaille sur des tuiles qui peuvent rentrer dans les caches et dont la durée d'utilisation est plus longue que dans le cas de la V1. On constate une accélération d'un facteur 2.

Il faut aussi regarder l'écart type des mesures par itération. La v1 a un comportement plus stable que celui de la v2 dû à son comportement très régulier (écart type plus faible). La v2 a un comportement plus chaotique car les tuiles sont affectées aux threads aléatoirement, en fonction de la progression de ceux-ci. Pour remédier à cela, la version 2 (v2b) est légèrement modifiée pour que l'affectation des tuiles aux

```
#ifndef MORTON_HPP
#define MORTON_HPP
#include <stdint.h>   // for the definition of uint
static int32_t morton1(int32_t x_)
{
        int32_t x = x_;
        assert(x <= 0xFFFF);
        x = (x | (x << 8)) & 0x00FF00FF;
        x = (x | (x << 4)) & 0x0F0F0F0F;
        x = (x | (x << 2)) & 0x33333333;
        x = (x | (x << 1)) & 0x55555555;
        return x;
};
static int32_t umorton1(int32_t x)
{
        x = x & 0x55555555;
        x = (x | (x >> 1)) & 0x33333333;
        x = (x | (x >> 2)) & 0x0F0F0F0F;
        x = (x | (x >> 4)) & 0x00FF00FF;
        x = (x | (x >> 8)) & 0x0000FFFF;
        return x;
};
static void umorton2(int32_t * x, int32_t * y, int32_t m)
{
        int32_t z1;
        *x = umorton1(m);
        z1 = m >> 1;
        *y = umorton1(z1);
};
static int32_t morton2(int32_t x, int32_t y)
{
        return morton1(x) | (morton1(y) << 1);
};
#endif
```

TABLE 5.5 – Code source des fonctions d'encodage (`morton2`) / décodage (`umorton2`) des coordonnées (x, y) d'une matrice 2D vers/depuis son indice sur la courbe de Morton (figure 5.10). On notera la simplicité de mise en œuvre par un simple fichier d'inclusion qui permet au compilateur d'optimiser (vectoriser) les changements de coordonnées. Note pour un éventuel utilisateur : la courbe de Morton parcourt une matrice carrée. Il faut donc vérifier que les coordonnées (x, y) retournées par `umorton2` soient valides ou alors stocker la coordonnée de Morton. L'équivalent 3D existe pour parcourir un cube.

threads ne soit plus aléatoire mais, en s'inspirant des méthodes peu sensibles aux caches (dites « cache oblivious »), les tuiles sont traitées selon une courbe de Morton [mor] (voir figure 5.10 et le code source correspondant dans la table 5.5). Cette simple modification va stabiliser les écarts de mesures mais aussi conduire à un gain supplémentaire de performance. L'utilisation des courbes de remplissage de surface (resp. d'espace) de type Morton peuvent aussi être utilisées pour répartir des données en inter nœud, comme le montre la thèse de Julien Loiseau [Loi18], pour laquelle j'étais membre invité du jury. Dans ce cas, il s'agit de répartir des particules sur les nœuds de calcul en assurant une bonne localité des données, à l'instar de ce qui est fait pour les caches, limitant ainsi le recours à des communications.

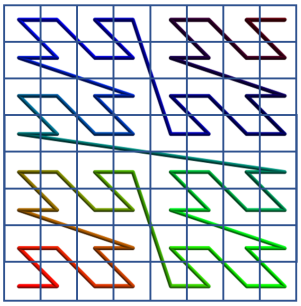


FIGURE 5.10 – Parcours d'un maillage 8x8 selon une courbe de Morton pour le cas 2D. Une coordonnée (x, y) est équivalente à son abscisse curviligne le long de la courbe.

Au final, la réorganisation du parallélisme en prenant en compte le mécanisme de cache a accéléré le code d'un facteur 2.09, sans pour autant dégrader la portabilité du code, ce qui est le but recherché.

La recherche de la performance portable se fait aussi au travers du développement de plates-formes telle que FleCSI décrit dans la thèse de Julien Loiseau ou Kokkos [CTS14], deux initiatives du Département de l'Energie (DOE) américain. L'environnement Kokkos prend en charge les structures de données pour les adapter automatiquement à l'architecture cible et produire le programme le plus efficace possible, simplifiant de ce fait le travail du développeur de code. Cette approche, qui repose lourdement sur des mécanismes avancés de C++, est très intrusive. Si elle est appliquée à un code du patrimoine C++, elle impose une réécriture profonde. De plus, reposant sur le C++, elle exclu les codes FORTRAN. L'avenir dira si cette voie est la plus pertinente pour le développement de grands codes efficaces sur des calculateurs de classe exaflopique.

5.2.3 La performance extrême

Compte-tenu du contexte des codes du CEA/DAM, nous avons toujours essayé de porter une grande attention à la portabilité tout en visant une performance décente. En collaboration avec un expert d'Intel (qui se qualifie de programmeur « ninja »), nous avons essayé de quantifier la différence entre performance décente et performance maximale et surtout de mesurer l'impact sur le code de la recherche de cette dernière. Dans l'article [GS], nous sommes partis d'HydroC et nous avons appliqué toute une série de transformations qui, additionnées les unes aux autres, ont conduit à une accélération d'un facteur **12** sur un KNC 7210P par rapport au code d'origine.

Cette étude montre qu'un programmeur motivé et **expert** peut tirer des performances d'un code existant mais au prix d'une grande complexité du source, obérant sa maintenabilité dans le temps, un facteur impossible à négliger au CEA. Néanmoins des leçons positives émergent de cette étude : l'application de techniques simples comme le blocking (voir par exemple [LRW91]), si elles sont introduites dès la conception du code, peuvent apporter un surplus significatif de performances sans toutefois menacer la lisibilité et la maintenabilité de ce dernier. De même, bien garder en mémoire les notions de localités spatiales et temporelles dans l'utilisation des variables ou tableaux minimisera les pertes de performances dues à des mouvements inutiles de données.

L'ensemble des travaux sur OpenCL, sur le code laser et sur l'optimisation extrême m'ont permis d'écrire une synthèse [G C15] à destination des acteurs de la physique nucléaire et plus généralement à tout développeur de codes scientifiques. En brossant un panorama des évolutions des matériels et des environnement logiciels, j'ai démontré que, dès 2015, nous étions à un point d'inflexion dans l'usage des supercalculateurs qui va obliger **tout** développeur de codes à modifier ses habitudes (trop souvent attendre la prochaine génération de machines qui calculeront encore plus vite sans avoir besoin de toucher aux programmes) et commencer à programmer en visant la performance, tout en maintenant la portabilité du source.

– ♦ ∞ ♦ ∞ ♦ ∞ ♦ ∞ ♦ ∞ ♦ ∞ ♦ –

6 Contributions au Graphique

La génération d'images à partir de données scientifiques est souvent coûteuse en temps calcul. Les bases de données peuvent être de grande taille, les images peuvent être de très haute résolution soit pour être affichées sur un mur d'images soit pour réaliser des posters de grandes dimensions. Souvent les deux se combinent : l'utilisateur souhaite un poster à partir d'une simulation très détaillée. Si, de plus, il souhaite explorer interactivement ses données, il sera nécessaire de calculer les images dans un temps réduit, souvent inférieur à la seconde. Enfin, si l'image doit être vue sous forme d'animation et en stéréo, il faut calculer deux images (une par œil) en moins de $\frac{1}{60}s$ pour que l'utilisateur ne ressente aucune gêne.

Deux voies complémentaires sont possibles pour accélérer la production des images : utiliser un matériel adapté et/ou introduire le parallélisme dans les outils graphiques à l'instar de ce qui a été fait dans les codes de calcul scientifiques.

Dans ce chapitre consacré au graphique, la première section décrira mes recherches de l'intérêt du parallélisme sur les logiciels graphiques, plus spécifiquement pour le rendu volumique, et la suivante son apport sur la voie matérielle au travers de la réalisation de murs d'images.

6.1 Rendu volumique parallèle

Les images produites à partir de simulations numériques 3D sont majoritairement composées d'éléments surfaciques représentés par des polygones colorés. La figure 6.1 est un exemple de représentation surfacique. Ces polygones décrivent, par exemple, les surfaces des objets visualisés ou représentent des iso-valeurs [1] d'un phénomène physique. Si l'on superpose plusieurs polygones, celui qui est le plus proche de l'observateur va masquer ceux de derrière. Il peut être nécessaire d'avoir une vision plus globale du volume étudié en reproduisant une vue de type radiographique : c'est le rendu volumique.

Si les cartes graphiques ont depuis longtemps été optimisées pour afficher extrêmement rapidement des polygones (rasterisation [2] accélérée par le matériel), le rendu volumique reste une opération très coûteuse en calcul. Le recours au parallélisme s'impose donc pour produire les images en un temps raisonnable.

Il existe deux grandes familles de visualisation de volumes reposant sur l'utilisation de rayons : le « ray tracing » et le « ray casting ».

Le ray tracing consiste à lancer des rayons depuis les différentes sources de lumières et de les faire réfléchir sur les surfaces rencontrées jusqu'à ce qu'ils rencontrent l'œil de l'observateur (figure 6.2). L'ancêtre des logiciels open source de ray-tracing est POV-Ray [pov] ou plus récemment YafaRay [yaf], lui aussi open

1. iso-valeur = lieux dans la géométrie où une grandeur physique prend une valeur donnée. En 2D, une iso-valeur est représentée par des segments de droites plus ou moins jointifs. En 3D une iso-valeur est le plus souvent un ensemble de triangles qui peuvent former des surfaces.

2. La rasterisation consiste à plaquer un élément géométrique sur une grille 2D de pixels. C'est ce qui donne l'effet de marche aux droites obliques affichées sur un écran, si l'on regarde de près.

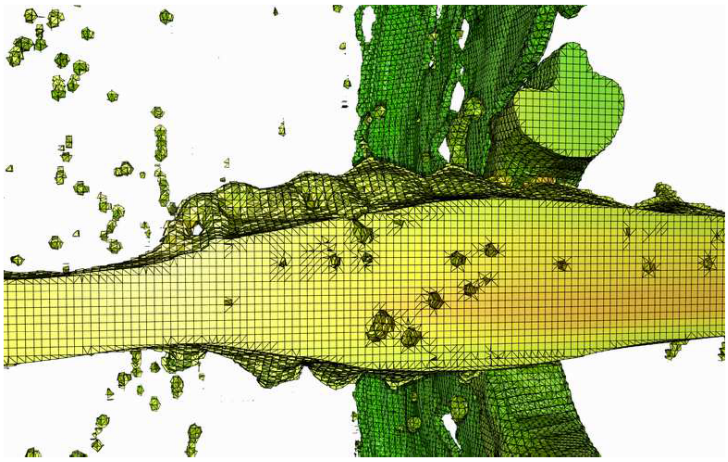


FIGURE 6.1 – Visualisation surfacique composée de lignes et de polygones colorés. Cet exemple représente la reconstruction des surfaces des matériaux d'une simulation de CFD.

FIGURE 6.2 – Un exemple d'image calculée en ray tracing avec YafaRay. Cette technique est très utilisée pour sa capacité à produire des images photo-réalistes en architecture ou pour le design automobile. ©Jimmac @ yafaray.org

source.

Dans le cas du ray casting (figure 6.3) on lance, depuis l'œil de l'observateur, un rayon qui va pénétrer tout droit à l'intérieur de l'objet et être plus ou moins atténué selon les propriétés optiques de la matière traversée.

L'atténuation le long de la trajectoire du rayon est décrite par l'équation 6.1 (qui est de la même famille que l'équation 5.1).

$$C(P) = \int_0^T e^{-\int_0^T \sigma(s)ds} C(t)dt \tag{6.1}$$

Sa version discrète, utilisée par le logiciel de rendu volumique étudié, est donnée par l'équation 6.2, où σ représente l'opacité courante au point utilisé dans l'intégration [3].

$$C(P) = \sum_{i=0}^{n} \left[\prod_{j=0}^{i} e^{-\int_j^{j+1} \sigma(s)ds} \right] C(i) \tag{6.2}$$

Si désormais les outils de rendu volumique sont devenus classiques comme par exemple dans VTK [vtk],

3. La valeur de l'opacité est déterminée par interpolation tri-linéaire au sein de chaque voxel.

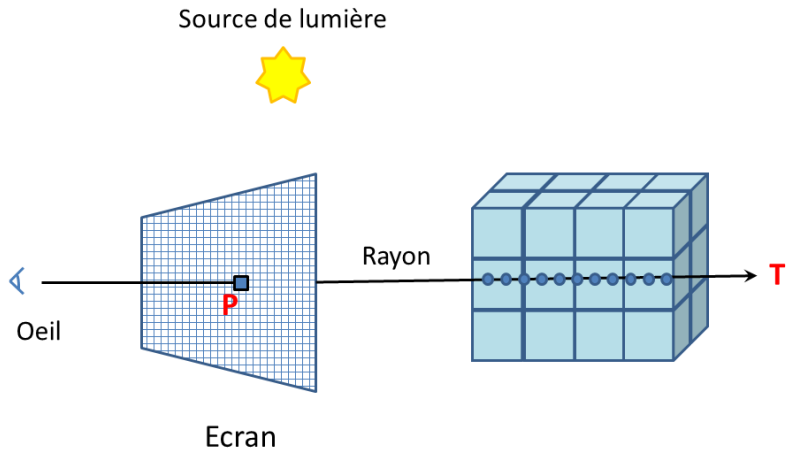


FIGURE 6.3 – Le principe du ray casting.

en 1994 ils étaient peu développés car demandant beaucoup de ressources calcul. Le programme sur lequel j'ai travaillé, lors de mon séjour au Los Alamos National Laboratory, avait été développé sur CM5. J'ai réalisé son portage en C++ sur T3D en conservant la méthode de parallélisme d'origine adaptée à l'utilisation d'une bibliothèque de communication par échange de messages.

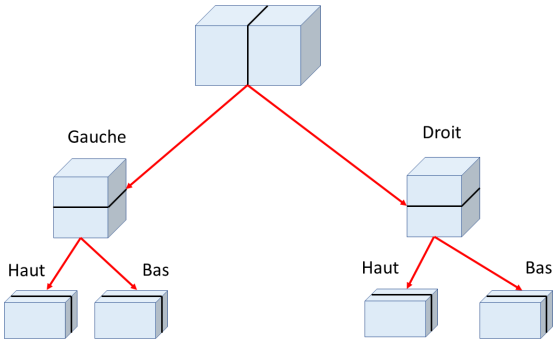


FIGURE 6.4 – Découpage en arbre kD (nécessite un nombre de tâches qui soit une puissance de 2)

Ce programme, décrit en détail dans [Han+95] et [Gui97], fonctionne de la façon suivante : le volume est tout d'abord découpé selon un arbre kD (illustré sur la figure 6.4). Chaque petit volume est pris en charge par une tâche qui va résoudre l'équation 6.2 et produire une image partielle du volume.

Le volume utilisé ici est un parallélépipède rectangle découpé en petits cubes (appelés voxels). Les valeurs affectées aux voxels sont comprises dans l'intervalle $[0 - 255]$ (après normalisation) et peuvent provenir par exemple d'un scanner tridimensionnel. Nous verrons plus loin que la technique du rendu volumique peut s'appliquer à d'autres types de maillages.

Les pixels calculés par le programme sont codés sur 4 composantes entières, forcées dans l'intervalle $[0 - 255]$: le rouge (Red), le vert (Green), le bleu (Blue) et la transparence (appelé canal Alpha), d'où le nom de RGBA pour ce codage. Une opacité, qui est l'inverse de la transparence, est le terme σ de l'équation 6.2 et représentera une des valeurs possibles affectées aux voxels. L'ensemble des valeurs des opacités pour les 255 cas possible est appelé fonction de transfert.

En appliquant l'opérateur « **over** » de Porter-Duff [PD84], qui satisfait l'équation 6.3, sur la composante d'opacité et en parcourant l'arbre kD depuis l'arrière vers l'avant, on est en mesure de reconstituer l'image

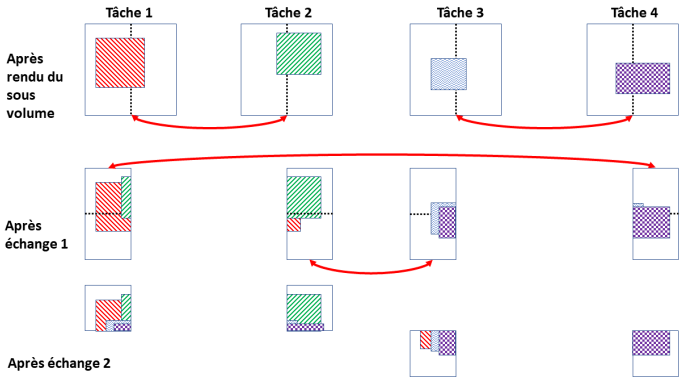


FIGURE 6.5 – Le binary swap ou échange binaire : les différentes tâches échangent deux à deux des portions d'images (flèches rouges). A la fin chaque tâche possède une fraction de l'image définitive.

de façon parallèle. C'est l'algorithme du binary-swap, illustré par la figure 6.5, dans lequel toutes les tâches sont actives en même temps. Il n'y a donc pas besoin de les synchroniser artificiellement.

$$a \textbf{ over } (b \textbf{ over } c) = (a \textbf{ over } b) \textbf{ over } c \tag{6.3}$$

Régler les paramètres de rendu volumique est difficile et nécessite beaucoup d'expérimentations. Pour faciliter ces essais erreurs, j'ai développé une petite interface utilisateur (figure 6.6) permettant de régler les paramètres du rendu volumique. Elle permet par exemple de jouer sur la fonction de transfert (les opacités) pour soit donner une idée globale du volume (les opacités sont alors faibles pour toutes les composantes de couleur) soit extraire une iso-surface correspondante à une des valeurs du volume (opacité fixée à 255 pour les valeurs désirées).

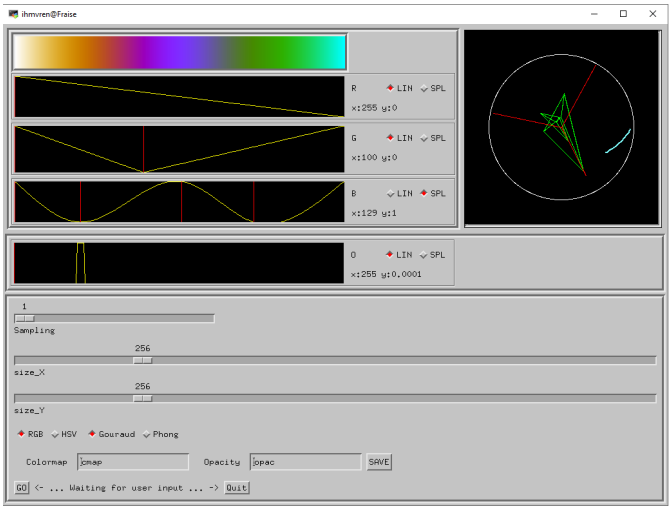


FIGURE 6.6 – Interface homme machine permettant de faire varier les divers paramètres du rendu volumique. Les cartes de couleurs et la fonction de transfert sont réglables interactivement de même que la rotation de la géométrie, qui est pilotée par un quaternion.

Pour accélérer les calculs, j'ai ajouté la possibilité de ne traiter qu'un éclairage de type Gouraud [Gou71] qui est moins coûteux qu'un éclairage de Phong [Pho73], au prix d'une dégradation de l'image (voir figure 6.7). En effet la méthode de Gouraud se contente d'une simple interpolation linéaire de l'intensité lumineuse à partir de celle calculée aux sommets du polygone à afficher. La méthode de Phong est plus

coûteuse car elle calcule en chaque point une illumination reposant sur les composantes ambiante, diffuse et spéculaire de la lumière.

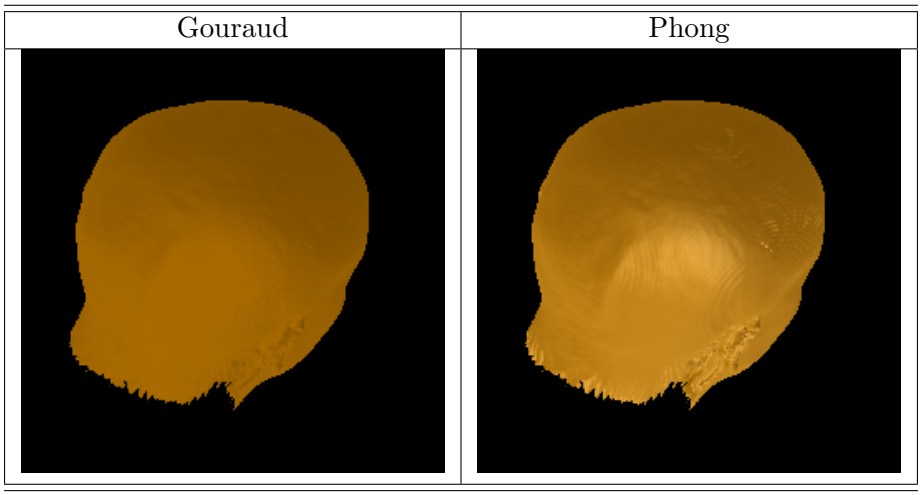


FIGURE 6.7 – Comparaison des illuminations de Gouraud et Phong.

En 1994, les outils du parallélisme par échange de message étaient peu utilisés dans les grands codes de production. Le plus utilisé était PVM [pvm], précurseur de MPI. Ma première version du logiciel de rendu volumique reposait sur PVM. Néanmoins l'implémentation disponible sur CRAY T3D n'était pas aboutie et donc pour contourner les difficultés (mauvaise barrière), la bibliothèque ACLMPL [Pai+95] a été développée et utilisée. ACLMPL a permis de passer à MPI facilement par la suite (sur T3E et T90 puis Tera). Ce développement a mis en évidence l'importance des couches de communication dans la performance globale des codes de calcul et des outils de visualisation parallélisés.

Mes travaux sur le rendu volumique ne se sont pas limités à ce programme car il ne prenait en compte que des maillages structurés. Dans [MG09], nous montrons qu'il est possible d'utiliser efficacement le rendu volumique sur maillage AMR en utilisant une méthode de pré-intégration capable de prendre en compte l'ombrage. La figure 6.8 illustre le résultat de ces travaux. Il est à noter que l'aspect parallélisme n'est pas pris en compte dans cette publication et reste donc à faire. Néanmoins, la structure en arbre utilisée par l'AMR se rapproche fortement de celle utilisée dans le cas structuré détaillé dans ce chapitre. Il est donc probable qu'au prix d'une adaptation, l'algorithme de composition reposant sur l'opérateur *over* soit utilisable dans le cas de l'AMR.

Ces travaux sur le parallélisme et le rendu volumique prennent une nouvelle importance avec la montée en puissance de l'*in situ* qui est constatée depuis le milieu des années 2010. Les simulations étant de plus en plus détaillées, les données produites sont de plus en plus volumineuses. Le temps d'entrées-sorties (« I/O ») devient une fraction importante du temps calcul voire prédominant. Le contournement, minimisant les volumes écrits, consiste à intégrer le processus de dépouillement au code en exécution et d'assurer un partage des données entre eux, d'où le nom d'*in situ*. Comme les simulations sont réalisées par des codes parallèles, il est indispensable de disposer de briques élémentaires de visualisation compatibles avec le parallélisme pour profiter à plein des ressources disponibles. Les implémentations peuvent être diverses :

- la visualisation est réalisée par une bibliothèque appelée par le code (« co-processing »). Les espaces mémoires sont donc partagés. Il y aura compétition sur l'utilisation de la mémoire et le calcul sera suspendu le temps de la construction des images ;
- le processus de visualisation partage une partie des nœuds de calcul avec le code. Les espaces mémoires sont séparés, d'où la mise en place d'un mécanisme de partage des informations (mémoire partagée unix ou fenêtre MPI). Les deux processus peuvent travailler de façon concurrente, mas-

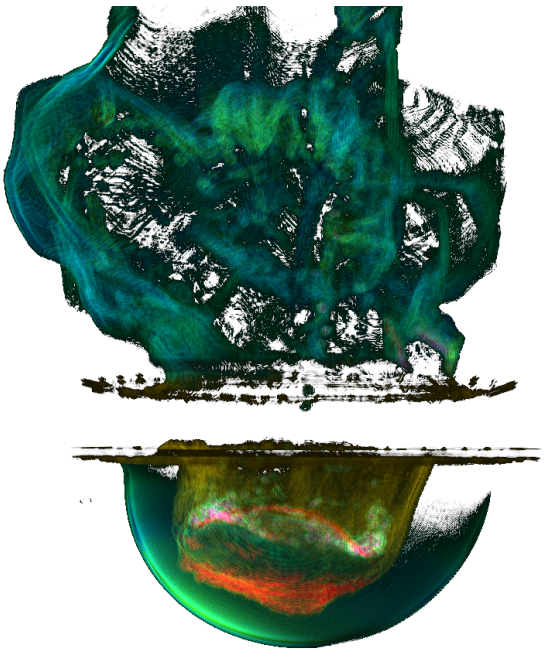


FIGURE 6.8 – Rendu volumique d'une simulation utilisant un maillage AMR

quant ainsi le temps de production des images ;
- on peut même envisager de faire tourner la visualisation sur une machine spécialisée en parallèle du supercalculateur principal. Cette solution est néanmoins plus lourde à mettre en place dans le cadre de la production d'un grand centre de calcul.

Au terme de cette section, nous retiendrons les points suivants :
- la composition des images étudiées ici se révélera indispensable pour la réalisation des grands murs d'images, tels que décrits dans la prochaine section ;
- les volumes à traiter étant toujours plus grands et les résolutions recherchées plus élevées, le passage au parallélisme est indispensable pour les logiciels graphiques ;
- les pratiques évoluent : l'analyse des résultats qui se fait classiquement en post-traitement va se faire de plus en plus *in situ*, mode d'exploitation indispensable pour les simulations de grande taille.

6.2 Le parallélisme matériel

Comme évoqué en introduction de cette partie, l'utilisateur a souvent besoin de plus de résolution et/ou plus d'images par seconde, souvent les deux ensemble. Pour atteindre de bons niveaux de performance, nous avons vu que l'on pouvait appliquer le parallélisme au logiciel de dépouillement à l'instar de ce qui est fait dans les codes de calcul. Il est aussi possible de le faire au niveau matériel pour la partie d'affichage de grandes dimensions. Le parallélisme est donc présent tout au long de la chaîne de simulation [4] comme décrit dans [Bre+03], dès la génération Tera 1, et [Agu+07] pour la génération Tera 10. Ce modèle est toujours d'actualité car il s'est poursuivi pour Tera 100 et Tera 1000 et sera conservé pour la future machine EXA 1 prévu pour 2023.

4. Les simulations conduites au CEA/DAM font appel à une suite de codes qui s'enchaînent pour décrire les différentes phases de phénomènes d'intérêt, d'où la dénomination de chaîne de simulation. Les logiciels de dépouillement / visualisation sont partie intégrante de cette chaîne de simulation.

nom	année	résolution	géométrie	recouvrement
MIRAGE 1	2001	3200×2400 $(3.2m \times 2.4m)$	4×4	non
MIRAGE 2	2006	4970×2730 $(5.5m \times 3m)$	4×3	oui

TABLE 6.1 – Les spécifications des deux **M**urs d'**I**mages à **R**ésolution **A**ugmentée et **G**rande **E**chelle (MIRAGE). Pour mémoire, en 2018 sont sortis les premiers écrans dit 8K d'une résolution de 7680 x 4320 pixels et tout un chacun peut acheter un écran 4K (3840 x 2160) pour un prix raisonnable !

6.2.1 Les grands murs d'images

Les écrans de station de travail du début des années 2000 disposaient de moins d'un million de pixels. Ces gammes de résolution ne permettaient pas de voir les détails des simulations sans recourir à des zooms, ces derniers faisant perdre la vue globale de l'objet visualisé.

Pour contourner cette limitation, il apparaît naturel de juxtaposer des écrans sous forme de pavage jusqu'à atteindre la résolution globale désirée. Les écrans peuvent être des écrans classiques (CRT dans le passé, LCD aujourd'hui) ou des images projetées ou rétro-projetées à l'aide de projecteurs numériques. Cette approche nécessite de prendre en compte plusieurs points :

1. comment sont générées les différentes parties des images ?
2. comment se fait la jonction des diverses images ?
3. comment se fait le calibrage colorimétrique ?
4. quelle compatibilité avec la stéréo ?

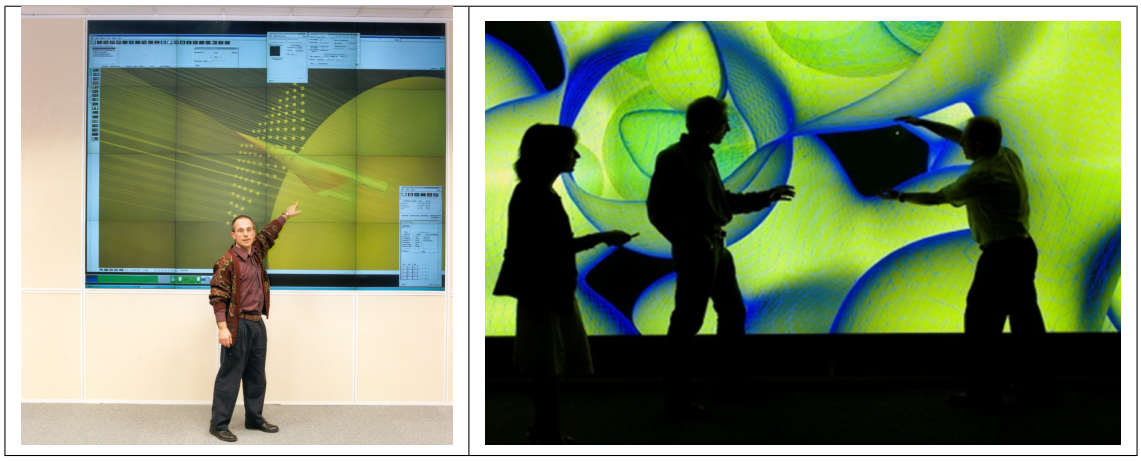


FIGURE 6.9 – Deux générations de murs d'images MIRAGE au CEA/DIF, à gauche la version de 2001 et à droite celle de 2006.

Reprenons chaque point en détail à la lumière de l'expérience acquise au travers des murs MIRAGE 1 et 2 illustrés par la figure 6.9 et dont les caractéristiques sont rappelées par la table 6.1.

Le mur de 2001 utilisait un seul ordinateur SGI ONYX Infinite Reality qui savait découper l'image produite par le logiciel en quatre parties. Un système électronique découpe chaque 1/4 d'image en 4 sous-images (donc 4x1/16 de l'image totale). Cette approche concentre le parallélisme sur le matériel et non pas sur le logiciel amont. Le mur de 2006 utilise un cluster de PC pour générer les 12 images à l'aide de 12 projecteurs. Le parallélisme n'est plus géré par le matériel et donc doit être pris en charge par le logiciel de visualisation. Nous retombons alors sur des situations bien traitées par les codes de calcul (synchronisation obligatoire de l'affichage par « `MPI_Barrier()` » par exemple pour ne pas produire des

images incohérentes : sans cette synchronisation des parties de l'image affichée pourraient correspondre à des temps ou des angles / facteurs de zooms différents).

FIGURE 6.10 – Vue arrière et avant du grand mur (MIRAGE 2006) à son installation, pendant le réglage initial. Les zones de recouvrements des images sont clairement visibles ainsi que les défauts d'alignement qui devront être corrigés.

La jonction entre deux images est fonction du système retenu. La solution de 2001 reposait sur des cubes indépendants mis côte-à-côte. Un espace de 0.9 mm séparait deux écrans entre eux, formant une ligne noire permanente sur l'image globale visible sur la figure 6.9 à gauche. A l'usage, les utilisateurs finissaient par ne plus voir ces lignes [5]. Comme le montre la figure 6.10, l'usage de projecteur sur une dalle unique provoque un recouvrement des images qu'il faudra gérer tant du point de vue matériel que logiciel, sous peine de produire des effets de décalages désagréables.

Le calibrage des couleurs est un problème difficile car il est fonction des caractéristiques de la source de lumière qui va produire l'image. Les solutions retenues en 2001 et 2006, à base de projecteurs à lampes, présentent des variations visibles mais faibles comme illustré sur la figure 6.11. Ce problème existe toujours en 2019 !

On peut espérer que les murs construits à partir de panneaux de LED soient plus homogènes du point de vue colorimétrique. Il est possible de compenser légèrement ces différences en appliquant une correction en sortie de carte graphique dans le cas du cluster mais un tel réglage est très fastidieux à réaliser car à appliquer à chaque pixel, réglage souvent visuel.

Enfin, la vision en stéréo réclame :

- de choisir le type de stéréo que l'on veut produire active ou passive et donc de choisir les équipements associés. En stéréo passive, un procédé physique sépare les images droite et gauche qui sont affichées simultanément. En stéréo active, des lunettes, synchronisées sur l'affichage des images, occultent alternativement l'œil droit puis le gauche, donc avec un affichage de l'œil gauche puis de l'œil droit ;
- de produire les images avec un taux de rafraîchissement suffisant pour le confort physiologique de l'utilisateur. La stéréo n'a pratiquement pas été utilisée avec le mur de 2001 car l'installation ne pouvait pas dépasser les 30 images/s (i/s) alors qu'il fallait au moins 120 i/s (60 i/s par œil pour une image stable) en stéréo active. Le mur de 2006, par contre, est parfaitement utilisable en stéréo.

5. Ce type de technologie est toujours utilisée de nos jours, notamment pour afficher les grandes images derrière les présentateurs des journaux télévisés.

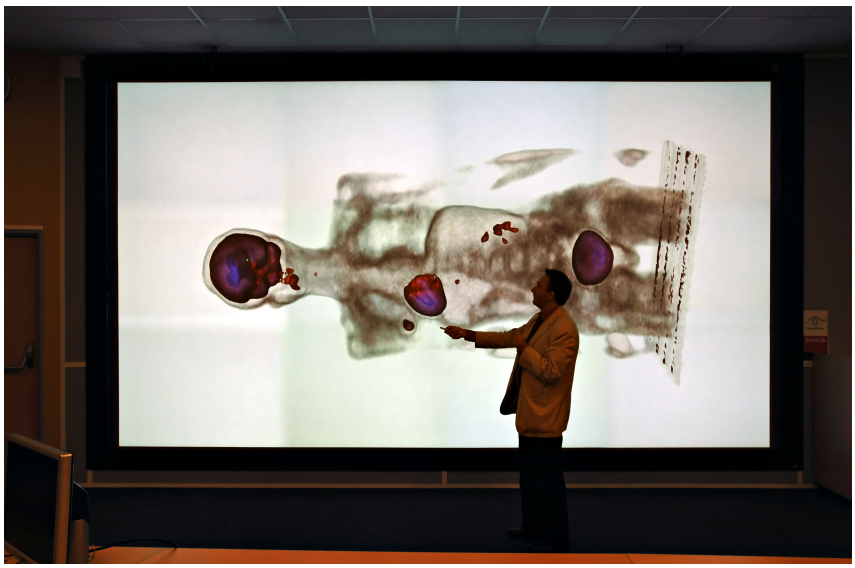


FIGURE 6.11 – Rendu volumique sur le grand mur d'images MIRAGE (version de 2006) du CEA/DIF. Ce qui est affiché sur cette photo est en fait un film construit image par image et visualisable en stéréo. Les variations colorimétriques sont clairement visibles sur les aplats blancs.

Le parallélisme intervient donc aussi dans la visualisation haute performance. Une vision analytique du processus associé fait apparaître 4 phases :

1. la lecture des données (I/O) ;
2. la phase de pré-rendu où les primitives 3D sont transformées en primitives 2D + Z. Conserver l'information de profondeur (Z) sera utile dans les phases suivantes pour réaliser une occultation correcte des (parties des) éléments les plus éloignés qui doivent être masqués ;
3. une phase de rendu où les primitives 2D + Z sont transformées en pixels + Z ;
4. de composition des images où les pixels sont triés selon Z.

Donc nous disposons de différents endroits où l'on peut trier les primitives comme le décrit Molnar dans [Mol+94]. Les deux tris les plus usités sont le « Sort First » et « Sort Last ».

Dans le mode Sort First (figure 6.12), les processus échangent un volume de données qui dépend de la géométrie initiale et selon un découpe spatial *a priori*. Un déséquilibre de charge est possible.

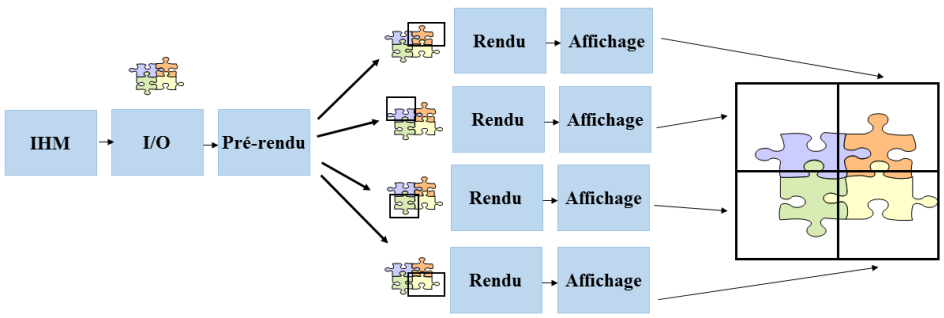


FIGURE 6.12 – Sort First : le tri se fait sur les primitives 3D.

Le Sort Last (figure 6.13) n'échange que des pixels. Les volumes échangés correspondent à la taille de l'image finale. Pour les hautes résolutions ce volume est important mais fixe (et donc connu à l'avance).

Le cas général, représenté par la figure 6.14, est idéal car il est parallèle de bout en bout et ne nécessite

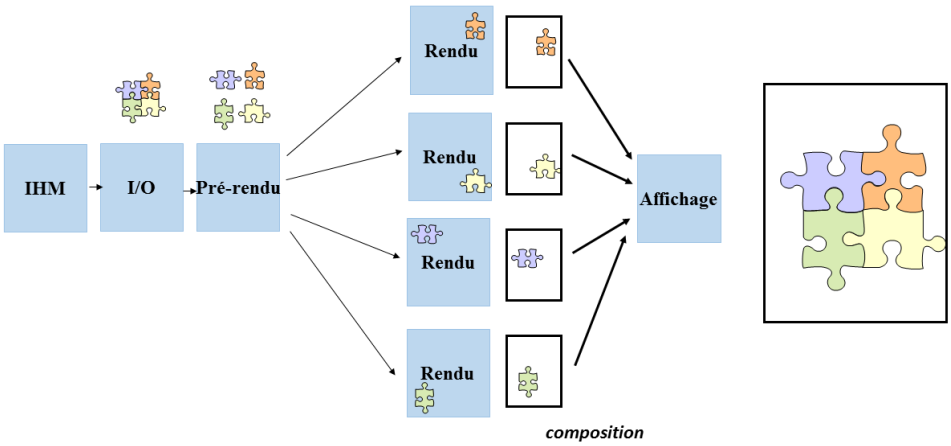


FIGURE 6.13 – Sort Last : le tri se fait sur les pixels+Z .

pas de phases de tris. C'est celui qui sera privilégié dans le mode *in situ* où la phase d'I/O sera remplacée par une lecture des données directement dans la mémoire du code (parallèle).

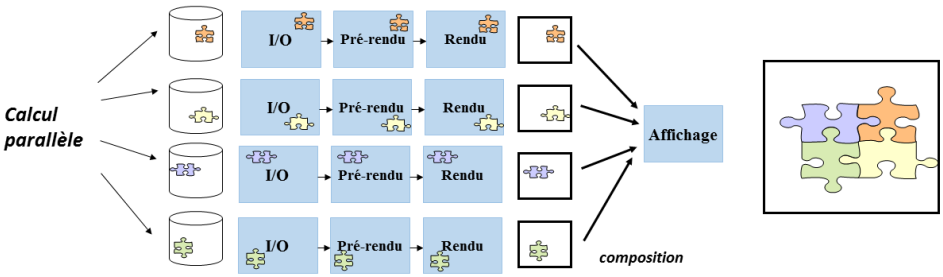


FIGURE 6.14 – Illustration du pipeline graphique, tel qu'il peut être utilisé dans le mode *in situ* ou sur un grand mur d'images. Le traitement graphique est parallèle de bout en bout.

Si l'on veut afficher un film sur un équipement tel que MIRAGE, sachant qu'un film parait fluide à partir de 24 ou 25 i/s, le recours au parallélisme prend tout son sens. Un logiciel de visualisation de films a été spécifiquement développé pour permettre la visualisation d'images en stéréo, à partir d'un cluster, à des vitesses supérieures à 30 i/s. Dans ce cas, les phases de pré-rendu et de rendu sont réalisées une fois pour toutes (pré-traitement des images) et le logiciel se contente d'afficher le plus rapidement possible chaque sous-image de manière synchrone. La photo de la figure 6.11 est une bonne synthèse de ce qui a été vu dans cette partie sur le graphique : une simulation de tomographie par émission de position (TEP) est visualisée sous forme d'un film, pouvant être affiché en stéréo, créé à partir d'images de rendu volumique.

6.2.2 L'haptique

Dès que l'on a un afficheur de haute résolution capable de faire de la stéréo, il est tentant d'essayer d'interagir avec la simulation au travers de périphériques adaptés au 3D. L'interaction peut être simple comme avec une souris 3D ou bien avec un retour d'effort comme illustré sur la figure 6.15. Dans ce dernier cas, on peut essayer de renvoyer une information complémentaire à l'utilisateur comme par exemple une résistance proportionnelle à une grandeur (en plus de la vue spatiale de la géométrie et de la couleur associée à une autre grandeur). L'utilisateur va déplacer un pointeur virtuel dans la scène 3D et se faire une idée des valeurs de la grandeur par la résistance plus ou moins grande qu'il ressentira.

Or un individu n'aura une perception réaliste de résistance que si le périphérique réagit à une fréquence

FIGURE 6.15 – Périphérique 3D à retour d'effort (ici un Phantom) utilisé par un doctorant au CEA/DAM.

d'environ 1 kHz. Cela impose donc d'être capable de localiser la donnée dans la simulation, d'afficher la scène 3D et mettre à jour la force de retour du périphérique à une vitesse au moins équivalente ($< 1/1000s$). Si cela reste possible avec de petites simulations (comme les quelques atomes de la photo), une simulation de grande taille reste impossible à explorer par cette méthode, même si l'on utilise un parallélisme massif, la limitation provenant bien souvent des entrées-sorties ou des algorithmes mis en œuvre (comme les iso-valeurs).

L'idée d'utiliser un retour haptique pour analyser les grandes simulations a donc été abandonnée à la suite de ces contraintes physiologiques. L'autre contrainte physiologique (affichage à plus de 60 i/s) sur la stéréo s'est finalement révélée moins difficile à réaliser et la stéréo est maintenant utilisée de manière routinière.

Ce chapitre illustre que l'approche du parallélisme pour le matériel graphique partage des similitudes fortes avec celui des logiciels en mettant en jeu des techniques analogues telles que le tri des éléments à calculer ou la composition des fragments d'images.

Même si les résolutions des afficheurs modernes sont de plus en plus élevées, la contrainte en temps de restitution des images impose le recours au parallélisme pour des bases de données de type HPC. Les techniques d'optimisation utilisées pour les codes de calcul trouvent un prolongement naturel dans les logiciels graphiques visant la performance extrême.

– ♦ ∞ ♦ ∞ ♦ ∞ ♦ ∞ ♦ ∞ ♦ ∞ ♦ –

7 Conclusion

Le chapitre de cette deuxième partie, focalisé les codes de calcul, montre que Vectorisation et GPU ne sont pas des notions incompatibles. Même si la voie GPU semble, *in fine*, la plus probable, il ne faut pas exclure la voie CPU et donc maintenir une programmation adaptable aux deux situations. Pour ce faire, la programmation par directives permet de maintenir la portabilité tout en offrant une performance correcte. OpenMP (4.5 et suivant) va clairement dans cette direction. Pour anticiper l'arrivée des machines fortement hétérogènes, il est grand temps de revoir en profondeur les codes du patrimoine pour les rendre compatibles avec les nouvelles architectures.

Sans pour autant chercher à devenir un programmeur « ninja », la compréhension fine des machines doit influer sur la conception des structures de données des codes.

De son côté, le chapitre consacré au graphique logiciel et matériel montre que le parallélisme doit se penser de manière globale et se retrouve à tous les niveaux. Comme le montre l'article [Ami+13] le parallélisme se pense dès la conception du centre de calcul puis dans les divers matériels utilisés (comme les murs d'images) pour être ensuite pris en compte dans les logiciels. De ces derniers, ceux traitant de graphique ou plus généralement de dépouillements des résultats de codes ne font pas exception et utilisent des méthodes et outils analogues à ceux des codes de calcul. Cette similitude est flagrante lorsqu'il s'agit d'utiliser des GPUs non plus pour produire des images mais pour accélérer les codes de calcul, faisant écho à ce qui a été abordé dans le premier chapitre de cette partie.

– ♦ ∞ ♦ ∞ ♦ ∞ ♦ ∞ ♦ ∞ ♦ ∞ ♦ –

Points de vigilance

- Les messages du compilateur, surtout pour améliorer la vectorisation.
- Le post-traitement des données produites par les codes, surtout elles sont massives, ne doit pas s'affranchir du parallélisme
- Bien choisir où et quand se fera le post-traitement des données : en différé ou *in situ*
- Attention à la physiologie de l'utilisateur dans la conception des outils graphiques (matériels et logiciels)

Bonnes pratiques

- Vérifier la bonne adhésion aux standards en utilisant plusieurs souches de compilateurs et des machines d'architecture différentes (au sens ISA).
- N'utiliser que des directives de vectorisation portables, surtout depuis l'arrivée de la clause SIMD d'OpenMP.
- Inclure le « penser parallèle » dans tous les maillons de la chaîne de calcul, y compris pour la chaîne de post-traitement

– ♦ ∞ ♦ ∞ ♦ ∞ ♦ ∞ ♦ ∞ ♦ ∞ ♦ –

Troisième partie

Conclusion et perspectives

8 Conclusion

La simulation numérique est l'un des piliers du programme Simulation conduit par le CEA/DAM. Elle met en œuvre des codes de simulation qui sont exécutés sur des supercalculateurs. Les simulations réalisées sur ces derniers sont de grandes tailles et durent longtemps. La recherche de la performance optimale des codes sur ces matériels exceptionnels est indispensable. Mon expérience professionnelle, résumée au sein de ce manuscrit, a été guidée par la recherche de la meilleure performance des logiciels, des codes et des matériels utilisés dans le programme Simulation.

La première partie de ce document est un état de l'art. Un rappel de quelques notions relatives aux codes de calcul est fait dans le chapitre 2. Il sert à préciser le contexte des chapitres suivants. Y sont abordés les différents types de maillages utilisés dans les codes. Les modèles mémoire et le parallélisme présent au sein des codes ou des logiciels de dépouillement sont définis. Le chapitre 3 décrit les divers supercalculateurs qui ont croisé ma carrière. Nous y montrons comment leurs diverses particularités architecturales ont été prises en compte dans la programmation.

La deuxième partie de ce manuscrit se focalise sur l'ensemble de mes travaux de recherche. Ces dernières concernent principalement deux activités : une composante code d'une part et une composante plus orientée graphique d'autre part. Au chapitre 5 est décrit le code DXFCI2 développé sur machine vectorielle. Sa vectorisation était indispensable et a mis en place des bases qui se révéleront utiles bien plus tard. Puis le parallélisme massif est abordé sous le triple angle de l'utilisation des GPUs, de la portabilité et de la performance extrême. Le chapitre 6, quant à lui, montre que le graphique, lui aussi, requiert l'utilisation du parallélisme que ce soit dans le logiciel comme dans le cas du rendu volumique ou dans le matériel comme pour les grands murs d'images.

Dans cette thèse, nous avons donc montré que l'algorithmique et la programmation, dans un contexte HPC, ne peuvent être envisagées sans tenir compte de l'architecture matérielle des supercalculateurs et sans « penser parallèle » et que pour cela, il est nécessaire de mettre en œuvre des stratégies de développement adaptées. Au fil des chapitres des points de vigilance et de bonne pratique sont apparus. Ils sont essentiels pour assurer la transition vers des ordinateurs encore plus puissants.

Les programmeurs devront être vigilants à l'extensibilité réelle de leurs programmes et à bien mettre en place les conditions de la vectorisation (régularité des accès si possible contigus, compatibilité avec les mécanismes de cache en privilégiant les localités spatiale et temporelle). Ils devront aussi garder en mémoire que le post-traitement des données produites par les codes, surtout elles sont massives, ne doit pas s'affranchir du parallélisme. Cela imposera de bien choisir où et quand se fera le post-traitement des données : en différé ou *in situ*. L'accès à la performance ne se fera donc pas sans un certain nombre de bonnes pratiques. Cela commencera par le fait de penser le parallélisme multi niveaux (inter et intra nœud, SIMD et/ou GPU) dès l'origine du projet en ayant pour objectif de minimiser les effets de la loi d'Amdahl (réduire la proportion de codes séquentiels ou les sérialisations par un bon usage des synchronisations).

Il faudra ensuite concevoir les structures de données en pensant performance et mettre en place les outils de mesure de la performance dès la programmation des premières lignes de code. Enfin, on devra vérifier la bonne adhésion aux standards en utilisant plusieurs souches de compilateurs et des machines d'architecture différentes (au sens ISA).

Le cycle de renouvellement des supercalculateurs du CEA/DAM ne s'est pas arrêté avec Tera1000. Le calculateur EXA1 est dores et déjà commandé et ses premiers éléments seront livrés dès 2021. Le chapitre suivant esquisse quelles peuvent être les perspectives d'avenir pour le HPC au-delà des besoins du CEA.

– ♦ ∞ ♦ ∞ ♦ ∞ ♦ ∞ ♦ ∞ ♦ ∞ ♦ –

9 Perspectives

9.1 Enjeux de l'exascale

Les deux parties précédentes ont couvert les évolutions des architectures de supercalculateur disponibles au fil du temps au CEA/DAM et leurs implications sur les codes de calcul, la poursuite de la performance de ces derniers, et enfin le parallélisme au service des logiciels graphiques, mettant en relief que ces trois domaines partagent des considérations communes. Ces deux parties montrent qu'il est indispensable de connaître les architectures utilisées pour le calcul pour choisir le bon modèle de programmation pour une parallélisation efficace. Donc d'avoir une prise de recul permettant d'avoir une vision la plus globale possible de l'ensemble machine – programme – utilisation et de ses évolutions.

Dans le cadre de cette vision globale, mes recherches actuelles sont naturellement tournées vers l'architecture des futurs supercalculateurs qui permettront au CEA/DAM d'assurer ses missions dans les années à venir. L'objectif est de disposer d'une machine de classe exaflopique (EXA 1) dès 2023. La contrainte énergétique va conditionner les études à poursuivre comme expliqué dans les deux prochaines sections.

9.1.1 L'énergie

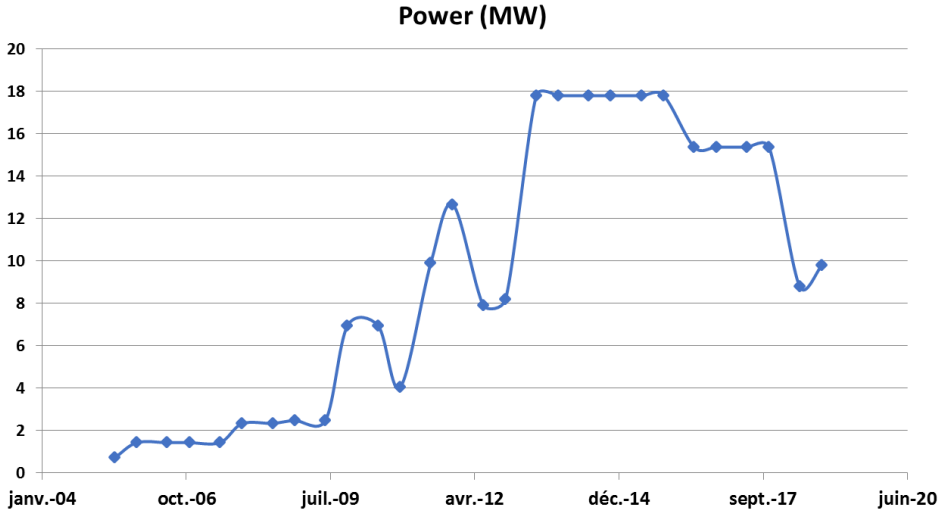


FIGURE 9.1 – Puissance dissipée par les machines numéro 1 du TOP500 au cours du temps.

La performance énergétique des supercalculateurs a pris une importance croissante, depuis quelques années, dans le monde du HPC et tout particulièrement au CEA/DAM. En effet le coût de l'électricité varie, en France, entre 600k€ et 800k€ le MW par an. Le prix de l'électricité devient une part importante dans le budget de fonctionnement d'une installation, laquelle doit être pensée en coût complet. La recherche de machines toujours plus puissantes s'accompagne désormais d'une recherche sur l'amélioration

de la performance énergétique. La communauté internationale espère réaliser un ordinateur d'1 EFlop [1] consommant entre 20 et 40 MW.

La figure 9.1 trace la puissance des supercalculateurs en fonction du temps. On y voit clairement qu'une dérive s'était installée avant 2013. La figure 9.2 permet de comprendre que seul un changement d'architecture peut éviter une inflation dans la consommation d'électricité surtout si, à architecture constante (la gauche du trait rouge), on essaye d'atteindre les 1000 PFlops. C'est pour cela qu'à partir de 2013, les supercalculateurs à base de GPU vont dominer le Top500 et être majoritaires dans le Green500. On notera l'excellente position du supercalculateur ROMEO de l'URCA dans le Green500 de novembre 2013, en position 5 avec une performance de 3 $GFlops/W$, le numéro 1 faisant 4.503 $GFlops/W$. En juin 2019 le numéro 1 du Green500 ne fait que 15.113 $GFlops/W$, ce qui est clairement insuffisant pour atteindre l'objectif d'1 EFlop dans 20 – 40 MW.

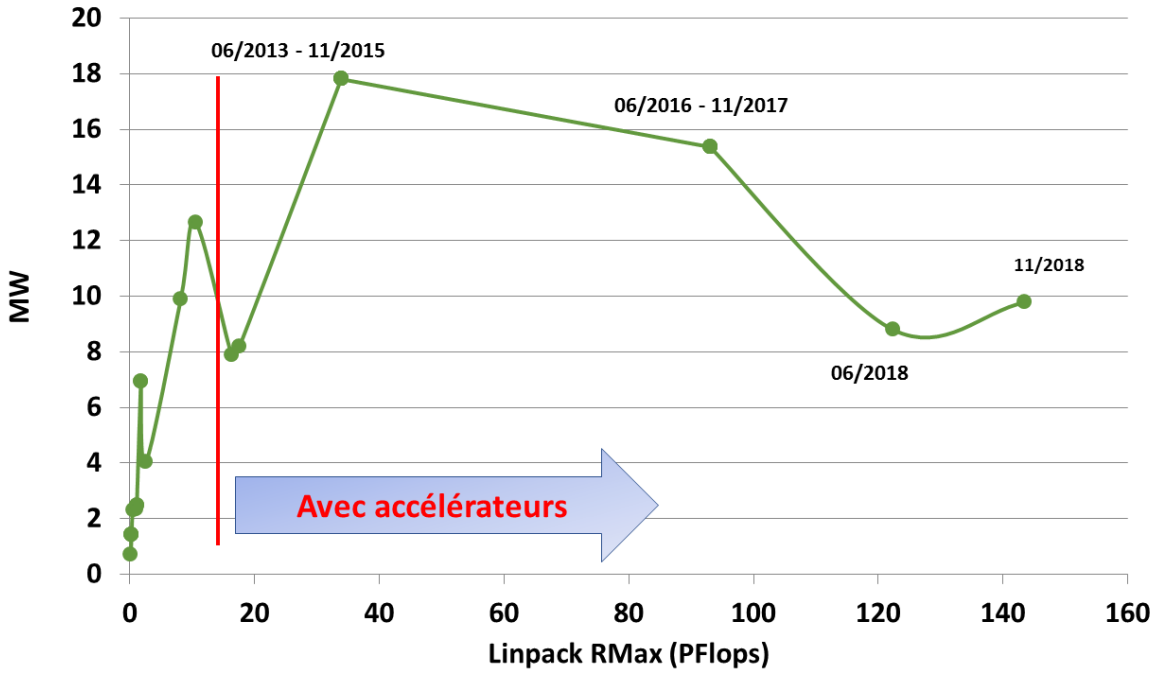


FIGURE 9.2 – La relation énergie performance des machines numéro 1 du TOP500. Il y a clairement une prise de conscience qui s'est faite autour des années 2013 - 2015, la tendance précédente ne pouvant pas conduire à des réalisations raisonnables économiquement.

Pour contenir la consommation énergétique de nombreuses études ont été lancées qui doivent être suivies ou auxquelles il est nécessaire de participer.

9.1.2 Les options matérielles possibles

Comme mentionné dès 2013 dans le Strategic Research Agenda [G C13] de l'ETP4HPC, un supercalculateur exaflopique sera un équilibre entre solution matérielle économe énergétiquement et programmabilité correcte. La recherche de cet équilibre passe par un effort de R&D qu'il est désormais impossible de conduire seul (au sens d'une seule institution). Cela passe par une participation à des projets européens comme Exanode [exa], Mont-Blanc 2020 (MB2020) [mb2] ou l'European Processor Initiative (EPI) [epi], pour ne citer que les trois projets les plus récents auxquels je participe, dans lesquels seront étudiés une architecture possible pour l'exascale (Exanode), des briques de base pour un processeur européen (MB2020) ou la réalisation d'un processeur européen (EPI) à faible consommation et hautes performances.

L'un des goulots d'étranglement des ordinateurs moderne est la mémoire dont la performance est très en deçà de celle du processeur. Ce décalage est mesuré par le rapport bande passante mémoire (GB/s)

1. Soit 10^{18} opérations flottantes par seconde.

par la vitesse de calcul ($GFlop/s$), exprimée en $byte/flop$. S'il est communément admis qu'une machine est équilibrée à un $byte/flop = 1$, force est de constater que l'on est loin de cette valeur aujourd'hui (Un Skylake d'Intel de 1460 Gflops possède une bande passante de 128 GB/s donc un $byte/flop$ de 0.09, donc loin de la cible de 1). L'arrivée de nouvelles mémoires rapides comme la HBM (environ 1.5 TB/s) permettra d'améliorer le $byte/flop$ des prochaines générations de processeurs (0.5 pour un processeur de 3 TFlops) et donc d'espérer un gain significatif sur la performance des codes, comme illustré sur la figure 9.3. Les mémoires non volatiles, offrant des capacités de stockage importantes, vont aussi bouleverser les usages, par exemple pour l'analyse de grandes bases de données qui pourront résider plus facilement en mémoire principale voire persister au redémarrage de l'ordinateur.

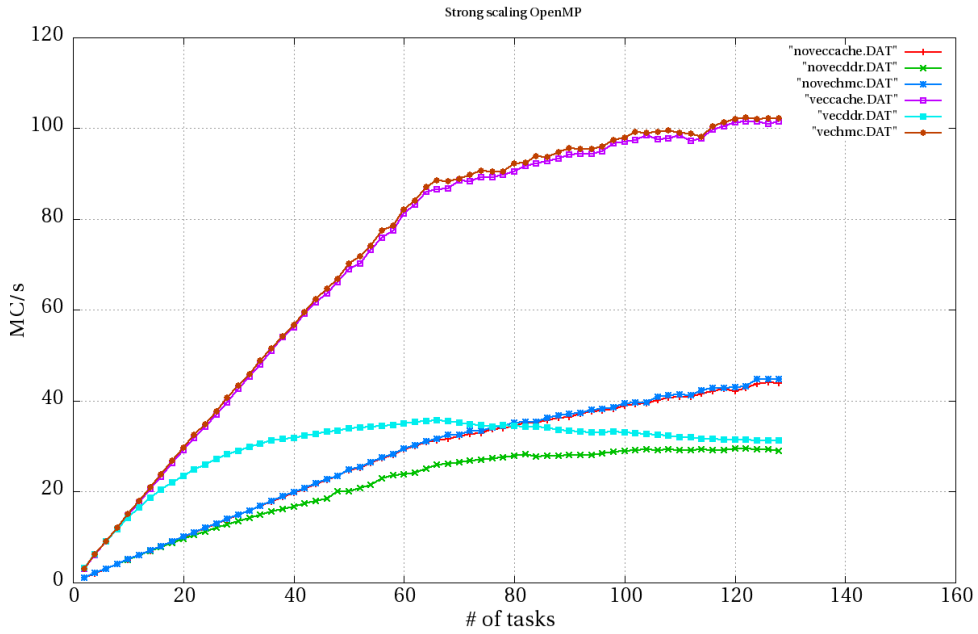


FIGURE 9.3 – Exemple de l'apport de la mémoire rapide du KNL sur le benchmark HydroC. Pour un même cas test, le code est compilé sans aucune vectorisation (courbes novec*) ou avec (courbes vec*). En jouant sur le placement des données allouées via `numactl`, on peut comparer les exécutions utilisant soit la DDR4 (bande passante maximum de 90 GB/s [Stream Triad]), soit la MCDRAM (qui est en fait de la HMC® de Micron Technologies d'une bande passante maximale de 470 GB/s) ou dans le mode où la MCDRAM sert de cache à la DDR4. On constate que 1) l'apport de la mémoire rapide est significatif pour un code vectorisé, que 2) le mode cache n'apporte pas une dégradation significative des performances et que 3) la latence de la DDR4 impacte très vite le passage à l'échelle. Ce petit exemple explique tout l'intérêt de disposer de HBM dans les futurs systèmes et montre que le mode cache serait à reconduire dans l'avenir – ce qui n'est pas certain.

Notre article [Rad+18], publié dans le cadre du projet européen ExaNode [2], conforte ces remarques. Il étudie différentes architectures sur la base du HPL et du HPCG en s'intéressant à la relation performance calculatoire par rapport à la performance mémoire. Notre étude montre, entre autres, que :

- les résultats mesurés sont très différents des valeurs théoriques affichées par les constructeurs de processeurs. Les exécutions du HPL et du HPCG sont indispensables pour comprendre les performances réelles d'une machine ;
- les latences mémoires diffèrent notablement selon les diverses architectures testées. Ces latences impactent fortement les performances observées. La qualité du contrôleur mémoire, qui conditionne la latence, sera donc un élément important à surveiller pour les prochaines générations de calculateur ;

2. The ExaNoDe research project is supported by the European Commission under the "Horizon 2020 Framework Programme" with grant Number 671578.

- le ratio $byte/flop$ tend à diminuer sur les architectures classiques (de type Xeon) mais croît pour les architectures émergentes et confirme l'importance d'équiper les prochaines générations de processeur de mémoire de type HBM.

L'arrivée des mémoires non volatiles (NVM [3]) de type 3D Xpoint^{TM}, permettant de disposer de tailles mémoires très importantes (plusieurs teraoctets par nœud), risque de modifier les usages : redémarrage immédiat des systèmes, pas de perte de données en cas de coupure électrique, bases de données en mémoire, systèmes de fichiers en mémoire, voire consommer moins d'électricité – ce qui reste à mesurer.

Les recherches sur l'efficacité énergétique sont très actives. Elles s'orientent tant sur la fabrication des circuits que sur les architectures utilisables pour le calcul. On ne s'intéressera ici qu'à la seconde voie. Si, comme nous l'avons vu plus haut, les GPUs sont une des options à considérer par leur bon rapport $GFlops/W$, d'autres options voient le jour. Il s'agit d'étendre les possibilités des unités SIMD (SVE) ou de rechercher de nouvelles architectures (CSA).

Nous avons vu à la section 3.3.3.1 (page 45) l'intérêt des instructions SIMD du point de vue programmation. Elles ont aussi un intérêt sur le plan énergétique car de telles instructions minimisent le nombre de circuits mis en jeu pour réaliser une opération sur des opérandes vectoriels. La tendance est donc à l'élargissement des vecteurs manipulés pour augmenter le rapport $GFlops/W$ comme dans le cas du KNL : en flottant double précision (FPDP) SSE opère sur 2 FPDP, AVX2 sur 4 FPDP et AVX512 sur 8 FPDP (cas du KNL ou du Skylake d'Intel).

SVE [sve], ou Scalable Vector Extension, est le nouveau jeu d'instructions SIMD proposé par la société Arm en remplacement du jeu d'instruction NEON qui est limité à 128 bits (2 FPDP). La particularité de SVE est de ne pas requérir de taille de vecteur à la compilation, contrairement à AVX512 par exemple. Le programme s'adaptera au matériel à l'exécution. SVE supporte des tailles de vecteurs de 128 bits jusqu'à 2048 bits. Les premières implémentations de SVE annoncées seront de 512 bits.

Mi 2019, il n'existe pas de processeur disponible pour le public qui implémente SVE (le processeur A64FX de Fujitsu, qui supporte du SVE 512 bits, est encore un prototype à accès restreint). Les recherches à mener consistent à étudier comment adapter les codes à ce type de vectorisation en relation avec les compilateurs qui seront disponibles. L'outil armie [arm] permet dès maintenant de tester la validité des codes utilisant SVE (après compilation par clang [llv]) et donc de préparer l'arrivée des processeurs commerciaux supportant ce jeu d'instructions.

Une autre piste est explorée par Intel (décrite dans le brevet [US+18] et expliquée sur le site [csa]) : L'accélérateur spatial configurable (ou CSA pour « Configurale Spatial Accelerator ») utilise des cellules programmables pour implémenter des graphes de tâches représentant des boucles de calcul ou des portions de code. Le principe de base est de plaquer le graphe des opérations contenues dans la boucle, tel que l'a déterminé le compilateur, sur une grille d'unités fonctionnelles. Les arcs du graphe sont les liens entre les unités fonctionnelles (voir figure 9.4). Le passage d'une itération à l'autre se fait en re-parcourant le graphe avec les valeurs de l'itération suivante. La boucle suivante sera traitée en reprogrammant les liens correspondant au nouveau graphe.

Une telle architecture a le potentiel d'atteindre un grand rapport $GFlops/W$. Néanmoins, à ce jour, ce n'est qu'une description d'intention. Il faudra attendre de disposer d'une suite logicielle adaptée et d'hypothétiques prototypes pour déterminer son réel intérêt. Si une telle architecture devait voir le jour, ce serait une rupture par rapport au modèle actuel. En particulier, il sera nécessaire d'évaluer son impact sur les codes et voir si les modèles de programmation actuels se prêtent correctement à l'utilisation de ce type de matériel.

Ces nouveaux développements (SVE et CSA) poussent à utiliser les instructions d'OpenMP 5.0, version qui a été rendue publique à la conférence Supercomputing de novembre 2018. Si l'intérêt de ces extensions est immédiat pour SVE, dans le cas du CSA, elles serviront à identifier les boucles qui seront les plus intéressantes à migrer sur cette nouvelle architecture. Les codes devront être adaptés pour prendre en

3. **N**on **V**olatile **M**emory

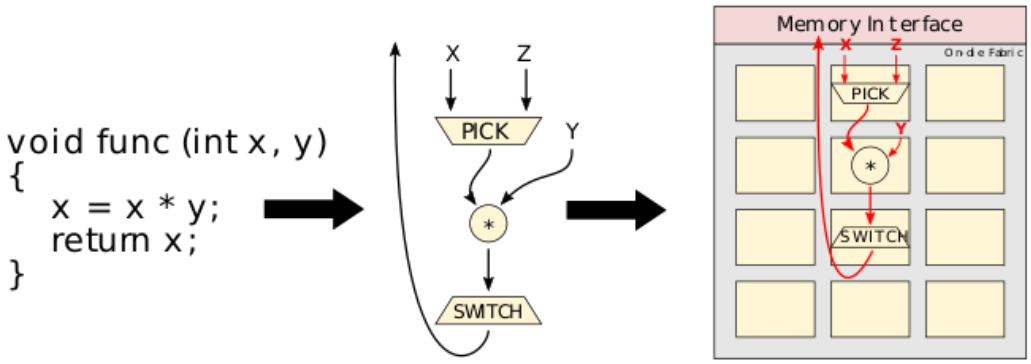


FIGURE 9.4 – Schéma décrivant le fonctionnement du CSA. Le code de gauche est traduit par le compilateur selon le graphe du milieu. A l'exécution, les arcs du graphe de compilation sont plaqués sur les switchs programmables de la figure 9.5. ©Wikichip tiré de l'explication du brevet d'Intel.

compte ces nouvelles directives, dès que les compilateurs disposeront d'un support effectif de cette nouvelle version du standard.

9.1.3 Autres pistes

Au-delà des deux nouveautés en approche, la construction de machines exaflopiques va provoquer une situation de rupture par rapport aux conceptions de machines actuelles. La communauté HPC doit se préparer à un bouleversement des architectures conduisant à des machines fortement hétérogènes. Les hétérogénéités seront à bas niveau comme dans le cas du CSA ou des GPUs, mais aussi à l'échelle plus globale de la machine qui sera composée de plusieurs partitions, chacune étant adaptée à une ou plusieurs classes de programmes. On pourra par exemple trouver une partition « classique » reposant sur des CPUs puissants (mais plus « énergivores ») aux côtés d'une partition accélérée de type « booster » [Eic+13] telle que proposée par le Jülich Supercomputing Center (JSC).

Des modifications des supercalculateurs tels que nous les connaissons aujourd'hui vont devenir aussi incontournables. Il s'agit par exemple de dissocier complètement la composante calcul de celle des I/O avec par exemple l'introduction des « proxy I/O » [pro]. Cette évolution est assez inévitable pour au moins trois raisons :

1. supprimer les disques des parties calcul : les disques rotatifs, appelés à être remplacés par des SSD, sont des fragilités qui deviendront de plus en plus prégnantes à mesure que les ordinateurs seront plus gros. Supprimer ceux-ci contribuera à la fiabilité de l'ensemble ;

2. augmenter les débits I/O : l'utilisation du réseau rapide pour écrire les données permet d'atteindre des débits très importants, bien plus que ceux accessibles sur des disques locaux. Dans le cas des proxy I/O qui sont expérimentés sur Tera 1000, le nœud de calcul écrit, via le réseau rapide, dans la mémoire d'une machine auxiliaire spécialement configurée pour prendre en charge les écritures de gros volumes de données dans un temps minimum. Cette spécialisation des fonctions est appelée à se développer car elle permettra une transition douce vers le modèle hétérogène évoqué plus haut ;

3. étudier comment aller plus loin que les systèmes de fichiers parallèles classiques (Lustre, GPFS) et aller vers des stockages objets permettant de manipuler des milliards de fichiers représentant des millions de petaoctets (on estime un volume de plusieurs Zettaoctets – soit 10^{21} octets – stockés en 2025).

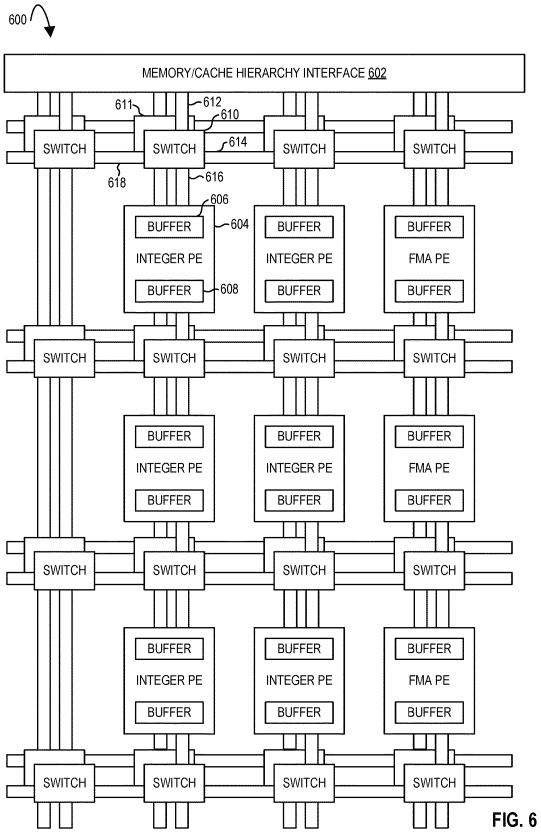


FIGURE 9.5 – Schéma décrivant le contenu du CSA. Les switchs sont programmables et permettent de relier les unités fonctionnelles entre elles en décrivant un graphe acyclique. Illustration extraite du brevet d'Intel.

9.2 Quel futur ?

Si les technologies et méthodes de programmation pour arriver à l'exaflop sont finalement bien comprises, de nouvelles technologies émergentes risquent de changer radicalement la façon d'appréhender le calcul haute performance. Il s'agit de techniques matérielles et logicielles poussées par l'intelligence artificielle (IA) d'une part et le calcul quantique d'autre part.

Le calcul quantique est une technologie en plein développement tant sur le plan matériel que logiciel. Diverses voies matérielles sont explorées et il est impossible aujourd'hui de dire quelles solutions survivront et surtout quelles seront celles qui pourront être industrialisées et miniaturisées pour être utilisées en dehors des laboratoires. Pour l'aspect logiciel, les recherches en cours portent sur les algorithmes qui seront compatibles avec cette technologie et sur les langages qui seront utilisés pour programmer au mieux les machines qui verront le jour. En 2019, il semble difficile d'imaginer un ordinateur quantique généraliste, notamment par l'absence de mémoires quantiques permettant de lire et d'écrire des informations au sein de la partie calculatoire quantique car la lecture d'une information quantique détruit son aspect quantique. Il est plus facile d'imaginer un superordinateur classique auquel on adjoindra **un** accélérateur quantique pour décupler ses performances sur **quelques** algorithmes. A ce jour, les contraintes physiques de refroidissement à très basse température et d'isolation des perturbations environnementales sont telles que ce ne seront pas des technologies de grande diffusion dans un avenir proche.

Les technologies de l'intelligence artificielle font des progrès importants et commencent à se rapprocher du monde de la simulation numérique, surtout via l'usage des GPUs. On parle de HPC augmenté où les algorithmes classiques peuvent être soit assistés par des techniques issues de l'IA (par exemple utilisation du meilleur pré-conditionnement du gradient conjugué selon le type de matrice produite par le calcul en cours) soit complètement remplacés (bibliothèque auxiliaire déterminant des propriétés de matériaux

remplacée par un réseau de neurones). Il sera important de pouvoir placer sa **confiance** dans les résultats de l'IA en étant capable d'expliquer voire démontrer le fonctionnement des outils de l'IA.

IA, HPC augmenté par l'IA et calcul quantique sont donc des champs de recherches immenses pour les années à venir.

Au delà de l'exaflop, les puissances de calcul seront telles que de nouveaux usages du HPC seront rendus possibles voire banals. Le HPC pourra faire naturellement partie de la boucle de décision (≪ HPC in the loop ≫) et permettra par exemple, le développement du jumeau numérique pour **tous** les objets manufacturés d'importance. Toute modification sur l'objet réel pourra être reportée sur le jumeau numérique et, via la simulation, on pourra anticiper l'impact de la modification sur le fonctionnement du jumeau réel. Mettre en place les jumeaux numériques réclame de maîtriser de nombreux sujets tels que les très grandes bases de données (à cause du grand nombre d'objets à suivre) ou la simulation numérique de haute performance (pour modéliser le comportement des objets). Cette évolution des usages ne sera possible que si le HPC sait devenir un outil de confiance, facile à mettre en œuvre pour une réelle fluidité d'utilisation. Pour de telles puissances de traitement, des auteurs tels que Liao et al. [Lia+18] parlent déjà de machines atteignant le Zettaflop – 10^{21} flops – dans environ 100 MW. La lecture de leur papier prolonge assez naturellement certaines des réflexions exposées dans ce manuscrit telles que :

- l'hétérogénéité des architectures pour pouvoir traiter tout type de code, sonnant ainsi la fin des machines généralistes telles que nous les connaissons ;
- la prééminence du modèle MPI+X qui évoluera vers plus de performance portable. On notera l'espoir de voir émerger enfin des langages spécifiques par domaine (ou DSL [4]) qui permettront de développer rapidement des applications pour une meilleure productivité des développeurs ;
- la poursuite des recherches sur les technologies réseau à l'instar de ce qui a été entrepris pour le BXI ;
- la poursuite des évolutions des systèmes de stockage accélérés par la plus grande diffusion des mémoires non volatiles.

Au final, les sujets d'études ne manqueront pas, qu'ils concernent les matériels, les logiciels de bas niveau, les modèles de programmations, les algorithmes ou les chaînes de simulation. Dans un premier temps, les sujets les plus importants à surveiller seront la consommation d'énergie des futurs systèmes et l'impact des nouveaux types de mémoires sur les usages de l'informatique et donc sur les codes futurs. L'émergence de nouvelles architectures ou de nouveaux types d'accélérateurs sera à suivre aussi. Enfin il ne faudra pas négliger la conception des systèmes I/O et de stockage qui peuvent rapidement devenir ≪ les maillons faibles ≫ de l'ensemble, surtout sous la pression croissante du HPDA. Tout cela ne pourra pas se faire sans assurer une veille technologique constante accompagnée du développement de mini-applications permettant de tester les nouveautés matérielles et logicielles.

– ♦ ∞ ♦ ∞ ♦ ∞ ♦ ∞ ♦ ∞ ♦ ∞ ♦ –

4. DSL = **D**omain **S**pecific **L**anguage

Bibliographie

[Agu+07] D. Aguilera, T. Carrard, **G. Colin de Verdière**, J-P Nominé et V. Tabourin. « Parallel Software and Hardware for capability visualization of HPC results ». In : *Astronum* (2007).

[Amd67] Gene Amdahl. « Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities ». In : *AFIPS Conference Proceedings* (1967), p. 483-485.

[Ami+13] Mickaël Amiet, Patrick Carribault, Elisabeth Charon, **Guillaume Colin de Verdière**, Philippe Deniel, Gilles Grospellier, Guénolé Harel, François Jollet, Jacques-Charles Lafoucrière, Jacques-Bernard Lekien, Stéphane Mathieu, Marc Pérache, Jean-Christophe Weill et Gilles Wiber. « Contemporary High Performance Computing : From Petascale toward Exascale ». In : sous la dir. de Jeffrey S. Vetter. CRC Press, 2013. Chap. 4 "Tera 100".

[arm] armie. url : `https://developer.arm.com/products/software-development-tools/hpc/arm-instruction-emulator/`.

[BDG13] F. Bodin, R. Dolbeau et **G. Colin de Verdière**. « One OpenCL to Rule Them All ». In : *6th International Workshop on Multi-/Many-Core Computing Systems* (sept. 2013).

[Bec+95] Donald J Becker, Thomas Sterling, Daniel Savarese, John E Dorband, Udaya A Ranawak et Charles V Packer. « BEOWULF : A parallel workstation for scientific computation ». In : International Conference on Parallel Processing. T. 95. 1995.

[Ber+12] L. Bergé, **G. Colin de Verdière**, S. Mauger et S. Skupin. « Simulations sur processeurs graphiques d'impulsions laser nanoseconds ». In : *revue Chocs, Avancées 2012* (oct. 2012), p. 6.

[BG11] Luc Bergé et **G. Colin de Verdière**. « Des processeurs graphiques pour simuler la lumière ». In : *La Recherche* (nov. 2011).

[Bre+03] O. Bressand, **G. Colin de Verdière**, J-C Lafoucrière, J-P Nominé et I. Surin. « Exploitation des résultats de calcul : du stockage à la visualisation des données ». In : *revue Chocs 28* (oct. 2003), p. 15-24.

[CM15] B. Clark et R. Madachy. « Software Cost Estimation Metrics Manual for Defense Systems ». In : *Software Metrics Inc.* (2015).

[Col04] Phillip Colella. « Defining software requirements for scientific computing ». In : *Slide of 2004 presentation included in David Patterson's 2005 talk* (2004). url : `https://www.lanl.gov/conferences/salishan/salishan2005/davidpatterson.pdf`.

[csa] csa. url : `https://en.wikichip.org/wiki/intel/configurable_spatial_accelerator`.

[CTS14] H. Carter Edwards, Christian R. Trott et Daniel Sunderland. « Kokkos : Enabling manycore performance portability through polymorphic memory access patterns ». In : *Journal of Parallel and Distributed Computing* 74.12 (2014). Domain-Specific Languages and High-Level Frameworks for High-Performance Computing, p. 3202-3216. ISSN : 0743-7315. DOI : https://doi.org/10.1016/j.jpdc.2014.07.003. URL : http://www.sciencedirect.com/science/article/pii/S0743731514001257.

[cud] CUDA. URL : https://developer.nvidia.com/cuda-zone.

[Dav17] John L. Hennessy et David A. Paterson. *Computer Architecture : A Quantitative Approach*. T. 6th edition. 2017. ISBN : 0128119055.

[DBB07] Romain Dolbeau, Stéphane Bihan et Fran cois Bodin. « HMPP : A Hybrid Multicore Parallel Programming Environment ». In : *Workshop on General Purpose Processing on Graphics Processing Units* (oct. 2007).

[Eic+13] N. Eicker, T. Lippert, T. Moschny et E. Suarez. « The DEEP Project - Pursuing Cluster-Computing in the Many-Core Era ». In : *2013 42nd International Conference on Parallel Processing* (Lyon, France). Oct. 2013, p. 885-892. DOI : 10.1109/ICPP.2013.105.

[epi] EPI. URL : https://ec.europa.eu/digital-single-market/en/news/european-processor-initiative-consortium-develop-europes-microprocessors-future-supercomputers.

[exa] EXANODE. URL : http://exanode.eu/.

[Fly72] M. J. Flynn. « Some Computer Organizations and Their Effectiveness ». In : *IEEE Transactions on Computers* C-21.9 (sept. 1972), p. 948-960. ISSN : 0018-9340. DOI : 10.1109/TC.1972.5009071.

[G C09] **G. Colin de Verdière**. « Going forward with GPU ». In : *High Performance Computing on Vector Systems* (2009).

[G C11] **G. Colin de Verdière**. « Introduction to GPGPU, a hardware and software background ». In : *Comptes rendus de mécanique, Académie des sciences* 339 (fév. 2011).

[G C13] **G. Colin de Verdière**. « Contributions au Strategic Research Agenda de l'ETP4HPC ». In : *ETP4HPC SRA1* (avr. 2013). URL : http://www.etp4hpc.eu/pujades/files/ETP4HPC_book_singlePage.pdf.

[G C15] **G. Colin de Verdière**. « Computing Element Evolution towards Exascale and its Impact on Legacy Simulation Codes ». In : *Eur. Phys. J. A 51 : 163* (2015). DOI : 10.1140/epja/i2015-15163-3.

[G C88] **G. Colin de Verdière**. « Simulation numérique de diagnostics X ». In : *Rapport des activités Laser, CEA, Centre d'études de Limeil-Valenton.* (1988).

[Gou71] H. Gouraud. « Continuous Shading of Curved Surfaces ». In : *IEEE Transactions on Computers* C-20.6 (juin 1971), p. 623-629. ISSN : 0018-9340. DOI : 10.1109/T-C.1971.223313.

[gra] GRAPH500. URL : https://graph500.org/.

[gre] GREEN500. URL : https://www.top500.org/green500/.

[GS] **G. Colin de Verdière** et J. Sewall. « From correct to 'correct and efficient' : a Hydro2D case study with Godunov's scheme ». In : *High Performance Parallelism Pearls Volume One : Multicore and Many-core Programming Approaches*. Sous la dir. de James Reinders et Jim Jeffreys. Morgan Kauffmann. Chap. 2.

[Gui10] **Guillaume Colin de Verdière**. « HMPP port ». In : *Prace Workshop* (mar. 2010). URL : http://www.prace-ri.eu/IMG/pdf/04_teslahmpp_gcdv.pdf.

[Gui97] **Guillaume Colin de Verdière**. « Visualisation des résultats sur calculateur parallèle ». In : *revue Chocs 16* (avr. 1997), p. 65-70.

[Han+95] Charles Hansen, **G. Colin de Verdière**, Michael Krogh, James Painter et Roy Troutman. « Binary-Swap Volumetric Rendering on the T3D ». In : *Cray Users Group Conference, Denver Co.* (Mar. 1995), p. 61-69.

[hpc] hpcg. url : http://www.hpcg-benchmark.org/.

[hpl] hpl. url : http://www.netlib.org/benchmark/hpl/.

[ibm] ibmp8. url : https://www.ibm.com/developerworks/community/wikis/form/anonymous/api/wiki/61ad9cf2-c6a3-4d2c-b779-61ff0266d32a/page/f1abe75a-a2b2-43dd-9d75-7dae28f5bc5f/attachment/3d574a4b-b414-42c8-85b0-f941115d569f/media/2014-06%20Power%208%20Servers%20June.pdf.

[Lav+12] Pierre-François Lavallée, **G. Colin de Verdière**, Philippe Wautelet, Dimitri Lecas et Jean-Michel Dupays. « Porting and optimizing HYDRO to new platforms and programming paradigms – lessons learnt ». In : (2012). url : http://www.prace-project.eu/IMG/pdf/porting_and_optimizing_hydro_to_new_platforms.pdf.

[Lia+18] Xiang-ke Liao, Kai Lu, Can-qun Yang, Jin-wen Li, Yuan Yuan, Ming-che Lai, Li-bo Huang, Ping-jing Lu, Jian-bin Fang, Jing Ren et Jie Shen. « Moving from exascale to zettascale computing : challenges and techniques ». In : *Frontiers of Information Technology & Electronic Engineering* 19.10 (oct. 2018), p. 1236-1244. issn : 2095-9230. doi : 10.1631/FITEE.1800494. url : https://doi.org/10.1631/FITEE.1800494.

[llv] llvm. url : https://llvm.org/.

[Loi18] Julien Loiseau. « Le choix des architectures hybrides, une stratégie réaliste pour atteindre l'échelle exaflopique. » 2018REIMS006. Thèse de doct. 2018. url : http://www.theses.fr/2018REIMS006/document.

[LRW91] Monica D. Lam, Edward E. Rothberg et Michael E. Wolf. In : *The cache performance and optimizations of blocked algorithms*. Proceedings of the fourth international conference on Architectural support for programming languages and operating systems (New York, NY, USA). ACM, 1991, p. 63-74. doi : 10.1145/106972.106981.

[maq] maqao. url : http://www.maqao.org/.

[Mau+13] Sarah Mauger, **G. Colin de Verdière**, Luc Bergé et Stefan Skupin. « GPU accelerated fully space and time resolved numerical simulations of self-focusing laser beams in SBS-active media ». In : *Journal of Computational Physics* 235 (fév. 2013), p. 606-625.

[Mau11] Sarah Mauger. « Couplage entre auto-focalisation et diffusion Brillouin stimulée pour une impulsion laser nanoseconde dans la silice ». 2011PA112146. Thèse de doct. 2011. url : http://www.theses.fr/2011PA112146/document.

[mb2] mb2020. url : http://montblanc-project.eu/.

[MG09] S. Marchesin et **G. Colin de Verdière**. « Accurate Volume Rendering For Adaptive Mesh Refinement Data ». In : *IEEE Visualization* (2009).

[Mol+94] S. Molnar, M. Cox, D. Ellsworth et H. Fuchs. « A sorting classification of parallel rendering ». In : *IEEE Computer Graphics and Applications* 14.4 (juil. 1994), p. 23-32. issn : 0272-1716. doi : 10.1109/38.291528.

[mor] morton. url : https://en.wikipedia.org/wiki/Z-order_curve.

[mpi] mpi. url : https://www.mpi-forum.org.

[Noa13] Gabriel Noaje. « Un environnement parallèle de développement haut niveau pour les accélérateurs graphiques : mise en œuvre à l'aide d'OPENMP ». 2013REIMS028. Thèse de doct. 2013. URL : http://www.theses.fr/2013REIMS028/document.

[opea] OPENACC. URL : https://www.openacc.org/.

[opeb] OPENCL. URL : https://www.khronos.org/opencl/.

[opec] OPENGL. URL : https://www.khronos.org/opengl/.

[oped] OPENMP. URL : https://www.openmp.org/.

[Ote+18] L. Oteski, **G. Colin de Verdière**, S. Contassot-Vivier, S. Vialle et J. Ryan. « Towards a Unified CPU–GPU code hybridization : A GPU Based Optimization Strategy Efficient on Other Modern Architectures ». In : *Advances in Parallel Computing series* 32 (mar. 2018). Sous la dir. de Sanzio Bassini, Marco Danelutto, Patrizio Dazzi, Gerhard R. Joubert et Frans Peters.

[Pai+95] James Painter, Patrick Mc Cormick, Michael Krogh, Charles Hansen et **Guillaume Colin de Verdière**. « ACLMPL : Portable and Efficient Message Passing for MMPs ». In : *First European T3D Workshop, Lausanne* (nov. 1995), p. 4-10.

[PD84] T. Porter et T. Duff. « Compositing Digital Images ». In : *Computer Graphics (proceedings of SIGGRAPH 1984)* 18.3 (juil. 1984), p. 253-259.

[Pho73] Bui Tuong Phong. « Illumination of Computer-Generated Images ». In : *Department of Computer Science, University of Utah* UTEC-CSs-73-129 (juil. 1973).

[pov] POVRAY. URL : http://www.povray.org/.

[pro] PROXYIO. URL : http://www.nfsv4bat.org/Documents/ConnectAThon/2014/Ganesha_9P_IO_Proxy_Cthon.pdf.

[pvm] PVM. URL : https://www.csm.ornl.gov/pvm/pvm_home.html.

[Rad+18] Milan Radulovic, Kazi Asifuzzaman, Darko Zivanovic, Nikola Rajovic, **Guillaume Colin de Verdière**, Dirk Pleiter, Manolis Marazakis, Nikolaos Kallimanis, Paul Carpenter, Petar Radojkovic et Eduard Ayguadé. In : *Mainstream vs. Emerging HPC : Metrics, Trade-offs and Lessons Learned*. Proceedings of SBAC-PAD (Lyon, France). 2018.

[Ram] Ramses. URL : http://irfu.cea.fr/Phocea/Vie_des_labos/Ast/ast_sstechnique.php?id_ast=904.

[Rap] Rapidmind. URL : https://en.wikipedia.org/wiki/RapidMind.

[SDK05] R. Strzodka, M. Doggett et A. Kolb. « Scientific Computation for Simulations on Programmable Graphics Hardware ». In : *Simulation Modelling Practice and Theory* 13 :8 (2005), p. 667-681.

[sve] SVE. URL : https://developer.arm.com/docs/dui0965/latest/sve-overview/introducing-sve.

[syc] SYCL. URL : https://www.khronos.org/sycl/.

[tbb] TBB. URL : https://www.threadingbuildingblocks.org/.

[top] TOP500. URL : https://www.top500.org.

[US+18] Fleming Jr. Kermin E. (Hudson MA US), US) Glossop Kent D. (Merrimack NH, Steely Jr. Simon C. (Hudson NH US), Tang Jinjie (Acton MA) US) et Gara Alan G. (Palo Alto CA US). « PROCESSORS, METHODS, AND SYSTEMS WITH A CONFIGURABLE SPATIAL ACCELERATOR ». 20180189231. Juil. 2018. URL : http://www.freepatentsonline.com/y2018/0189231.html.

[Val90] Leslie G. Valiant. « A Bridging Model for Parallel Computation ». In : *Commun. ACM* 33.8 (août 1990), p. 103-111. ISSN : 0001-0782. DOI : 10.1145/79173.79181. URL : http://doi.acm.org/10.1145/79173.79181.

[vtk] VTK. URL : https://www.vtk.org/.

[yaf] YAFARAY. URL : http://www.yafaray.org/.

Index

Titre français : **A la recherche de la haute performance pour les codes de calcul et la visualisation scientifique**

Cette thèse vise à démontrer que l'algorithmique et la programmation, dans un contexte de calcul haute performance (HPC), ne peuvent être envisagées sans tenir compte de l'architecture matérielle des supercalculateurs car cette dernière est régulièrement remise en cause. Après avoir rappelé quelques définitions relatives aux codes et au parallélisme, nous montrons que l'analyse des différentes générations de supercalculateurs, présents au CEA lors de ces 30 dernières années, permet de dégager des points de vigilances et des recommandations de bonnes pratiques en direction des développeurs de code. En se reposant sur plusieurs expériences, nous montrons comment viser une performance adaptée aux supercalculateurs et comment essayer d'atteindre la performance portable voire la performance extrême dans le monde du massivement parallèle, incluant ou non l'usage de GPU. Nous expliquons que les logiciels et matériels dédiés au dépouillement graphique des résultats de calcul suivent les mêmes principes de parallélisme que pour les grands codes scientifiques, impliquant de devoir maîtriser une vue globale de la chaîne de simulation. Enfin, nous montrons quelles sont les tendances et contraintes qui vont s'imposer à la conception des futurs supercalculateurs de classe exaflopique, impactant de fait le développement des prochaines générations de codes de calcul.

Mots clés : HPC, Simulation numérique, codes de calcul, visualisation scientifique, GPU, parallélisme

English title : **Searching for the highest performance for simulation codes and scientific visualization**

This thesis aims to demonstrate that algorithms and coding, in a high performance computing (HPC) context, can not be envisioned without taking into account the hardware at the core of supercomputers since those machines evolve dramatically over time. After setting a few definitions relating to scientific codes and parallelism, we show that the analysis of the different generations of supercomputer used at CEA over the past 30 years allows to exhibit a number of attention points and best practices toward code developers. Based on some experiments, we show how to aim at code performance suited to the usage of supercomputers, how to try to get portable performance and possibly extreme performance in the world of massive parallelism, potentially using GPUs. We explain that graphical post-processing software and hardware follow the same parallelism principles as large scientific codes, requiring to master a global view of the simulation chain. Last, we describe tendencies and constraints that will be forced on the new generations of exaflopic class supercomputers. These evolutions will, yet again, impact the development of the next generations of scientific codes.

Key works : HPC, numerical simulation, scientific code, scientific visualization, GPU, parallelism

Discipline : Informatique

Spécialité : Calcul Haute Performance

Université de Reims Champagne-Ardenne
CReSTIC – EA3804
UFR Sciences Exactes et Naturelles,
Moulin de la Housse, 51867 Reims