Table des matières

Chapitre 1

Introduction générale

Sommaire

La simulation numérique par le calcul haute performance est devenue, grâce à la constante évolution des ordinateurs, un atout crucial et un enjeu majeur pour l'industrie. En effet, la modélisation numérique permet à des acteurs industriels comme EDF de simuler des phénomènes physiques très complexes afin d'étayer des décisions d'exploitation (barrages, centrales). On peut, par exemple, simuler le fonctionnement complet d'une centrale nucléaire : de sa construction à la production d'électricité et même la gestion de scénarios d'accidents. Cependant, force est de constater que le résultat d'un logiciel de simulation numérique subit plusieurs approximations effectuées aux diverses étapes du processus de simulation (la modélisation mathématique du problème physique, la discrétisation du modèle mathématique et la résolution numérique en arithmétique des ordinateurs). Au vu de l'importance des décisions prises sur la foi de calculs, les exigences de sûreté des ouvrages de production imposent alors de s'assurer d'une part de la **reproductibilité** et d'autre part de la **qualité** et **fiabilité** des résultats des simulations.

Cette thèse s'inscrit dans le cadre de la vérification numérique des codes de simulation. La validation numérique d'un logiciel de simulation consiste à étudier sa fiabilité. Nous appelons ici *fiabilité d'un code de calcul*, sa capacité à produire des résultats justes malgré les erreurs d'arrondi dues à l'arithmétique flottante IEEE 754. La validation numérique est d'autant plus importante à l'heure actuelle où on est en constante recherche de meilleures puissances de calcul dans les codes pour simuler des phénomènes de plus en plus complexes et à plus large échelle sur des architectures massivement parallèles.

Plan du chapitre : Notre démarche dans ce chapitre introductif sera dans un premier temps d'expliquer le principe général de la simulation numérique en général : ses principales étapes, les moyens et outils utilisés (section 1.1). Nous nous intéresserons, dans un second temps, aux codes de calculs industriels, notamment ceux d'EDF (section 1.2). Nous évoquerons également l'important apport du HPC [4] dans les simulations (section 1.3). Enfin, nous nous poserons la question de la fiabilité des résultats de ces codes et l'influence de l'arithmétique de l'ordinateur ; nous introduirons la thèse en présentant de façon détaillée le sujet, son contexte, les motivations et les principaux enjeux de nos travaux (section 1.4).

1.1 Simuler pour décider

Pour le commun des mortels, les mots *laboratoire* et *recherche*, font généralement penser aux expérimentations, aux maquettes, aux tubes à essai et aux blouses blanches. Cependant, il existe bien des cas où il est tout simplement impossible, pour des raisons diverses (complexité, coût, inaccessibilité à l'échelle humaine, dangerosité), de réaliser certaines expériences. Dans ces cas, on fait appel à l'ordinateur à travers la simulation numérique qui fournit un outil pratique et très efficace afin de comprendre, de contrôler et de prévoir le fonctionnement des systèmes physiques. On parle alors d'expérience *in silico*. Dès lors, l'ordinateur est devenu un fantastique outil d'investigation et la simulation un outil indispensable, voire incontournable, dans plusieurs domaines de recherche et de développement notamment dans l'industrie. Son champ d'application est extrêmement vaste : on peut citer la mécanique des fluides, la science des matériaux, l'astrophysique, la physique nucléaire, l'aéronautique, la climatologie, la mécanique quantique, la biologie, la chimie, les sciences humaines, les mathématiques financières etc.

4. High Performance Computing, Calcul Haute Performance en français

Si on devait définir la simulation numérique, on pourrait dire que c'est le procédé selon lequel on exécute un (des) programme(s) sur un (des) ordinateur(s) en vue de représenter un phénomène physique [CEA, 2007]. Elle fait gagner du temps (et beaucoup d'argent) aux constructeurs d'automobiles et d'avions. Elle permet aux météorologues de prévoir le temps qu'il fera dans plusieurs jours [Ghidaglia et Rittaud, 2004]. On pourrait presque résumer la simulation numérique à une adaptation aux moyens numériques des modèles mathématiques. En effet, le point de départ du processus de simulation est l'observation d'un phénomène physique. Le modèle est une traduction des observations en équations mathématiques. La modélisation d'un phénomène consiste alors à prendre en compte les principes fondamentaux (conservation de la masse, de l'énergie, etc) et à déterminer les paramètres essentiels (positions, vitesses, températures...) qui permettent une description simple et réaliste de chaque élément intervenant et de son évolution dans le temps. Le modèle est complet [5] lorsque toutes les équations (traduisant les lois physiques qui régissent tous les éléments du système) sont écrites.

De la modélisation mathématique, on passe ensuite à la simulation grâce à la programmation en langages informatiques. Des méthodes numériques spécifiques sont utilisées pour résoudre les équations issues des modèles. La mise en œuvre de ces méthodes est elle aussi, une étape incontournable du processus de simulation. Bien souvent, les modèles sont posés sous forme d'une équation aux dérivées partielles (EDP) ou d'un système d'EDPs et on dispose très rarement de solutions analytiques pour ces équations. Il est alors indispensable de se tourner vers des méthodes d'approximation. Le choix de ces méthodes doit se faire en adéquation avec la modélisation physique du problème. Deux approches sont principalement mises à contributions : les méthodes déterministes et les méthodes probabilistes (ou statistiques). Dans la première catégorie, on résout les équations après avoir discrétisé les variables. Les méthodes déterministes les plus connus sont les volumes finis, les différences finies et les éléments finis [Ern et Guermond, 2004]. Les méthodes probabilistes, encore appelées "Monte-Carlo", définissent les techniques permettant d'évaluer une quantité déterministe à l'aide de l'utilisation de tirages aléatoires [Kalos et Whitlock, 2008]. Les méthodes numériques ont fait et font l'objet de nombreuses études, la littérature sur le sujet est d'ailleurs très vaste. On peut cependant en trouver un large aperçu dans ces ouvrages [Allaire, 2005, Dautray *et al.*, 1988, Lucquin et Pironneau, 1996].

La simulation numérique est finalement matérialisée par le logiciel de calcul. Cependant, en amont et en aval du processus, de nombreuses opérations complexes sont faites pour préparer l'expérimentation et ensuite pour exploiter au mieux les résultats. La CAO (conception assistée par ordinateur) permet de représenter les scènes avec des formes géométriques qui sont ensuite discrétisées par maillage. Les résultats des calculs sont sauvegardés et traités pour constituer une base qui puisse servir de références. Enfin, le processus se concrétise lorsque l'on visualise le phénomène étudié sur un écran. La dernière étape est donc la confrontation avec les observations de départ. Cette confrontation permet d'améliorer les modèles physiques, leurs paramètres et les logiciels de calcul. On pourrait presque dire que la simulation numérique est une autre approche du réel [CEA, 2007]. Il va sans dire, qu'elle ne peut supplanter les expérimentations et les observations, mais l'un de ses intérêts essentiels est de prédire les situations inédites. Simuler permet donc de mieux comprendre et d'anticiper. Pour les acteurs industriels, "simuler" permet principalement de décider. A titre d'exemple, l'investissement pour la construction d'un système hydro-informatique (un logiciel de calcul pour les simulations hydrauliques) est de l'ordre de 500000 à 1 million d'euros par an, mais les bénéfices pouvant

5. Parfois, on ne peut avoir toutes les équations, on peut quand même modéliser le système en approximant la physique

provenir d'un tel ouvrage sont de grandeurs bien plus élevées. Ainsi une simulation numérique de l'impact d'un ouvrage d'art en situation de crue a évité récemment une dépense de 12 millions d'euros. Une autre permettant de prédire de manière convaincante l'évolution d'une tache thermique a permis de réduire de coûteuses campagnes de mesures [Hervouet, 2007].

1.2 La modélisation numérique chez EDF

Les enjeux de la modélisation numérique sont considérables pour l'industrie, notamment pour EDF, le premier fournisseur mondial d'électricité. En ce sens, la division Recherche et Développement (R&D) du Groupe a pour principale mission d'aider les départements métiers (ingénierie) à relever les défis énergétiques de demain [EDF R&D, 2012], notamment dans le cadre de la transition énergétique. Pour ce faire, EDF R&D conçoit de nombreux logiciels de simulation numérique adaptés aux besoins des ingénieurs du Groupe. Ces logiciels permettent aux départements métiers de prédire des phénomènes extrêmement complexes influençant la durée de vie des centrales nucléaires, de simuler des dizaines de milliers de scénarios pour optimiser la production et gérer les risques sur les marchés de l'énergie, etc.

La suite logicielle Telemac-Mascaret [6] , par exemple, est dédiée aux écoulements à surface libre [Hervouet, 2007]. Le logiciel Telemac résout les équations de Saint-Venant et de Navier-Stokes grâce à la méthode des éléments finis. Il est utilisé pour les études liées à l'environnement aquatique, aussi bien pour les études d'impact et le dimensionnement des ouvrages que le calcul des marées et la simulation des crues. Il s'agit là d'un intérêt stratégique pour le Groupe EDF car ses principaux outils de production (barrages, centrales thermiques et nucléaires) sont généralement situés en bord de mer ou de rivière. Le chapitre 6 présente de façon plus détaillée la suite Telemac-Mascaret ainsi que ses principaux domaines d'application.

On peut également citer le Code_Aster [7] (un des plus vieux d'EDF R&D dédié à la mécanique des structures et à la thermo-dynamique) ainsi que le Code_Saturne [8] (dédié à la mécanique des fluides) [Archambeau *et al.*, 2004] ou les codes Sim Diasca, SYRTHES et Code_Carmel3D. Un bref récapitulatif des principaux codes développés à la R&D d'EDF est présenté dans le *tableau* 1.1. Ajoutons que ces codes peuvent être couplés. Le couplage permet d'associer un ou plusieurs codes pour résoudre un problème plus complexe. La sortie du code A peut alors être utilisé comme paramètre d'entrée du code B. Dans d'autres cas, le code A est appelé dans le code B. Il n'est pas inutile de préciser que le couplage doit se faire tout en respectant les contraintes physiques. A titre d'exemple, le Code_Saturne peut être couplé avec d'autres codes mais également avec lui même. Il peut être également couplé avec le code de thermique SYRTHES (voir *figure* 1.1) ou le code de mécanique Code_Aster.

Outre ces logiciels de simulation scientifique, EDF R&D développe aussi des *chaînes de calcul de cœur* dédiées à la simulation du fonctionnement des cœurs de centrale nucléaire. Ces chaînes de calcul permettent, grâce au calcul du flux neutronique angulaire [Marguet, 2011], de déterminer les paramètres pour une exploitation optimale des centrales. Elles sont également utilisées pour répondre aux exigences de l'Autorité de Sûreté Nucléaire (ASN). La chaîne actuelle en exploitation par les ingénieurs de la division production nucléaire est COCCINELLE [Hypolite *et al.*, 2012]. Une autre chaîne plus récente, COCAGNE [Guillo *et al.*, 2010b] a été développée. Cette dernière repose sur des modèles physiques plus aboutis et permet de faire des

6. Voir http://www.openmascaret.org
7. Voir http://www.code-aster.org
8. Voir http://code-saturne.org

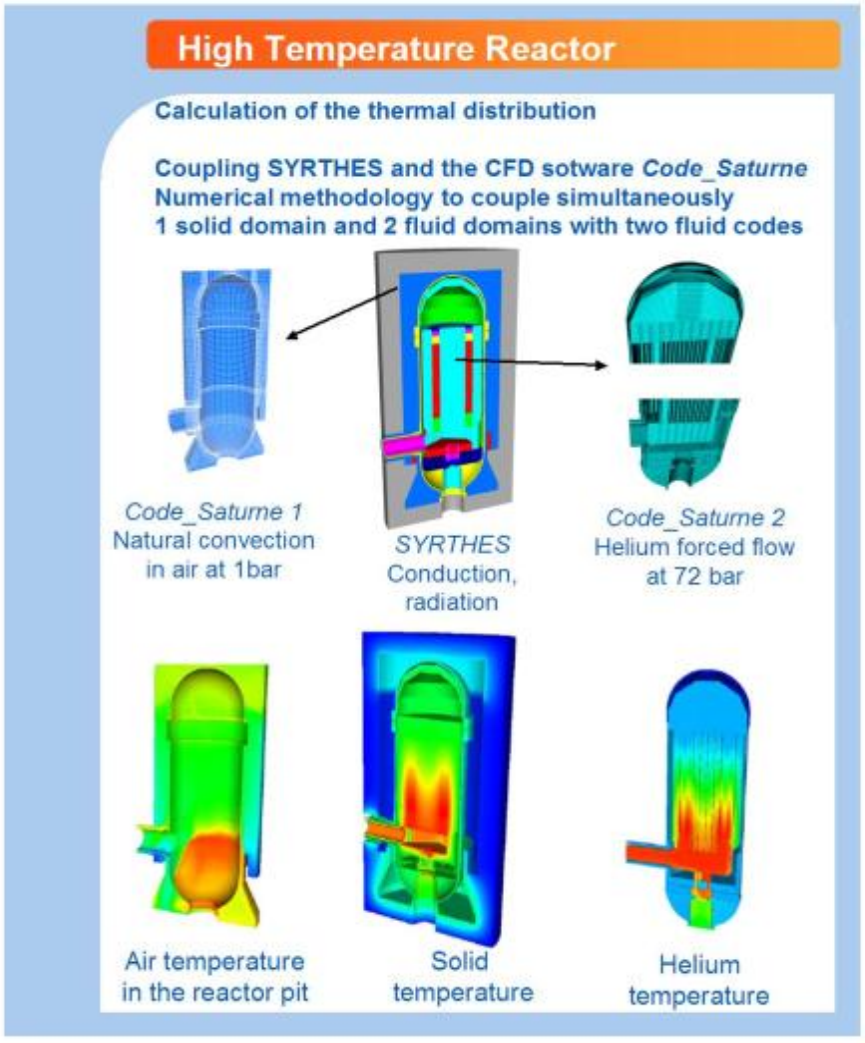


Figure 1.1 – Couplage Code_Saturne et SYRTHES pour calculer la distribution thermique dans un réacteur. Source : http://innovation.edf.com/recherche-et-communaute-scientifique/logiciels

Tableau 1.1 – Présentation des principaux codes Open Source d'EDF R&D, Source : [EDF R&D, 2013]

Nom	Objectifs et champ d'applications
CloudCompare	Logiciel d'édition et de traitement de données 3D (nuages de points ou maillages) développé depuis 2004 [Girardeau-Montaut, 2006]. Les données 3D sont recueillies sur les ouvrages EDF afin de simuler des opérations de maintenance ou de suivre les déformations géométriques.
Code_Aster	Logiciel de simulation numérique en calcul des structures (d'analyse et de modélisation multiphysique non linéaire, la fatigue, etc). Ce code est utilisé pour les études de comportement des matériaux utilisés pour les ouvrages (actuels et futurs) d'EDF.
Code_Saturne	Code généraliste de Mécanique des Fluides Numériques (CFD). Il permet de modéliser les écoulements incompressibles ou dilatables, avec ou sans turbulence ou transfert de chaleur. Le code résout les équations de Navier-Stokes avec la méthode des volumes finis.
Code_Carmel3D	Code de calcul de champ électromagnétique par éléments finis permettant d'effectuer des simulations numériques de contrôles non-destructifs (CND) par courants de Foucault dans les tubes de générateurs de vapeur [Moreau *et al.*, 2009].
Code_TYMPAN	Code de propagation acoustique dans des scènes 3D complexes. Il est utilisé pour l'évaluation et la prévision de l'impact sonore des sites industriels dans le cadre d'études d'ingénierie en acoustique environnementale.
SYRTHES	Code généraliste de thermique pour la résolution numérique de la conduction et du rayonnement en milieu transparent [Rupp et Peniguel, 1999]. La résolution de la conduction est basée sur une approche éléments finis.
Sim_Diasca	Moteur de simulation discrète intensément concurrent (parallèle et réparti). L'objectif est de traiter des problèmes de grande taille (plusieurs millions d'instances de modèles en interaction) tout en garantissant que les simulations respectent des propriétés essentielles (la causalité, la reproductibilité totale et une certaine forme d'ergodicité).

simulations plus complexes et plus proches de la réalité. COCAGNE a été principalement commanditée pour valider les plans de chargement de l'EPR Flamanville [9].

Dans le souci d'intégrer tout le processus de simulation (pré-traitement, CAO, maillages, visualisation, calcul, post-traitement) dans un seul et unique système logiciel et surtout d'augmenter la productivité des études, EDF co-développe avec le CEA et une vingtaine de partenaires industriels, la plateforme SALOME [10]. Cette plateforme open source permet la modélisation détaillée de l'ensemble des phénomènes physiques (mécanique, thermohydraulique, neutronique) et leurs interactions dans un environnement commun. On peut par exemple y lancer les principaux codes que nous avons mentionné plus tôt [11] (*tableau* 1.1). Cette plateforme facilite ainsi le couplage de codes (comme nous l'évoquions au §3 section 1.2) et l'utilisation de modules externes comme Homard, un module dédié à l'adaptation de maillage [Nicolas et Fouquet, 2013].

Il faut donc souligner l'importance actuelle de la simulation numérique pour le monde industriel. Le progrès technique, avec l'amélioration des capacités des ordinateurs et les avancées dans les méthodes numériques ont alors permis à la simulation de prendre une nouvelle dimension.

1.3 Vers des applications de plus en plus performantes

L'évolution continue des architectures matérielles offre des puissances de calcul inimaginables il y a seulement 20 ans. Les supercalculateurs sont actuellement capables d'exécuter des millions de milliards d'opérations par seconde (pétaflopique) voire mille millions de milliards d'opérations par seconde (exaflopique) [Dongarra *et al.*, 2011]. J.J.Dongarra et A.J. van der Steen illustrent parfaitement cette évolution dans [Dongarra et van der Steen, 2012] où ils dressent un bilan complet de l'évolution des systèmes HPC au cours des dernières décennies. Par exemple, il est intéressant de noter qu'un téléphone Iphone 4 est capable de réaliser 1.02 gigaFlops soit l'équivalent d'une machine du Top500 [12] [Meuer *et al.*, 2012] en 1995 [Dongarra, 2013].

En France, la machine Pangea de Total [13] est la plus puissante. Sa performance crête théorique est estimée à $2,296$ pétaFlops et elle a été classée 11^e au classement Top500 de juin 2013. Cette machine succède au palmarès à Curie [14] dont la performance crête théorique est de $1,667$ pétaFlops (classée 15^e au classement Top500 de juin 2013). EDF R&D est représenté dans ce classement avec ses machines Zumbrota (classée 38^e, performance crête théorique de $838,9$ téraFlops) et Ivanoe (classée 205^e, performance crête théorique de 191 téraFlops, voir annexe A.3 pour sa description détaillée). La première place du classement est actuellement occupée par une machine chinoise Tianhe-2 dont la performance crête théorique est estimée à $54,902$ pétaFlops. Elle succède à une machine américaine Titan qui a une performance crête théorique

9. L'EPR est une centrale de troisième génération. Sa construction a débuté en 2007 et se poursuit. Voir http://energie.edf.com/nucleaire/carte-des-centrales-nucleaires/epr-flamanville-3/presentation-48324.html

10. Voir http://www.salome-platform.org/

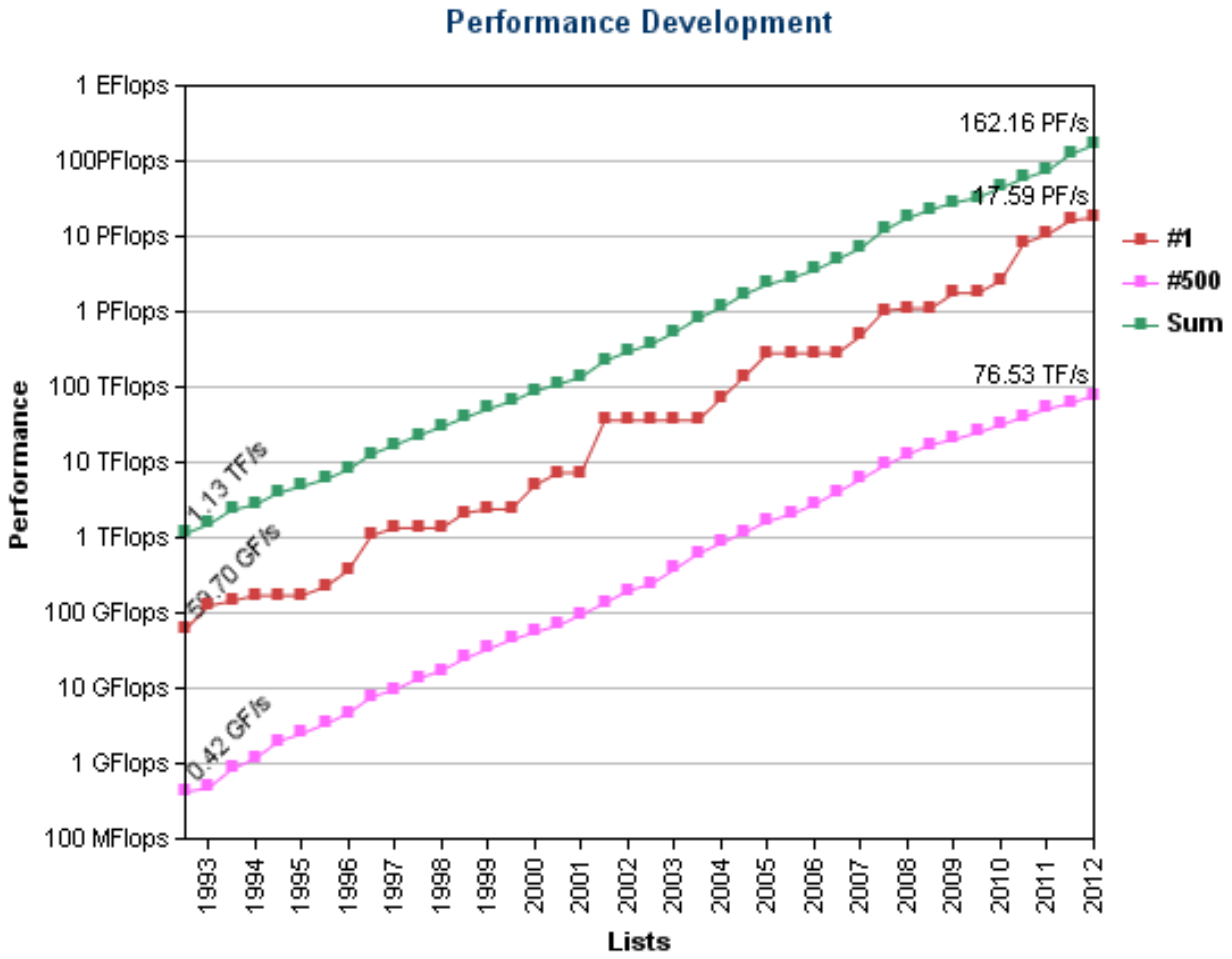
11. sous réserve d'avoir une distribution contenant les modules des codes

12. Le Top500 est le classement des cinq cents supercalculateurs les plus puissants au monde. Il est réalisé avec le test de performance LINPACK. Voir http://www.top500.org

13. http://www.top500.org/system/178071

14. La machine Curie a été financée par le GENCI (Grand Équipement National de Calcul Intensif) et installée au TGCC (Très Grand Centre de Calcul) du CEA à Bruyères-Le-Chatel(91). Elle est mise à disposition des chercheurs français et européens depuis mars 2012. La mise en place de la machine rentre dans le cadre de l'infrastructure européen de calcul intensif PRACE (Partnership for Advanced Computing in Europe).

Figure 1.2 – Croissance exponentielle de la performance des supercalculateurs depuis 1990. On présente ici l'évolution des machines classées 1^{er}, 500^{e} et la performance totale combinée des supercalculateurs de la liste Top500, Source : http://www.top500.org/



de $27,112$ pétaFlops (classée 2^e). Dès lors, cette évolution permanente a apporté une nouvelle dimension à la simulation numérique.

Cette nouvelle dimension apportée par le HPC, est pleinement illustrée dans le cahier spécial du magazine *La Recherche*, édition de novembre 2012 [lar, 2012]. Par exemple, grâce aux supercalculateurs, les autorités pourraient être alertées plus rapidement des risques de tsunami. Le Cenalt [15] (Centre d'alerte aux tsunamis) peut prévenir l'arrivée imminente de vagues générées par un séisme sous-marin. Le temps de calcul ne permettait pas auparavant une utilisation en temps réel des simulations. La parallélisation du code vient renforcer l'estimation du risque et fournit une carte nettement plus détaillée de l'impact attendu des vagues sur les cotes. Quand on se rappelle des dégâts des tsunamis de 2004 au large de l'île indonésienne de Sumatra [16] et celui de 2011 de la côte Pacifique du Tohoku au Japon [17], on ne peut qu'être ravi de ces récentes avancées.

Les supercalculateurs interviennent également dans la recherche en chimie du vivant. La modélisation des molécules du vivant permet de mieux comprendre le déclenchement des pathologies telles que la maladie d'Alzheimer. Ces simulations permettent alors de mieux orienter la recherche vers de bons traitements thérapeutiques. Une étude concernant la maladie d'Alzheimer a d'ailleurs été réalisée sur la machine Curie de Genci en décembre 2011. Au cours de ces simulations 80 000 cœurs de calculs du supercalculateur Curie ont été mis à contribution.

Il est alors impossible, aujourd'hui, de dissocier simulation numérique et calcul haute performance. L'augmentation de la complexité des modèles induit une augmentation importante du nombre de calculs et donc, des contraintes sur les temps d'exécution [Kirschenmann, 2012]. Cet apport considérable du HPC se retrouve également dans les simulations numériques à EDF, notamment dans l'évolution de la taille des cas traités au cours des deux dernières décennies.

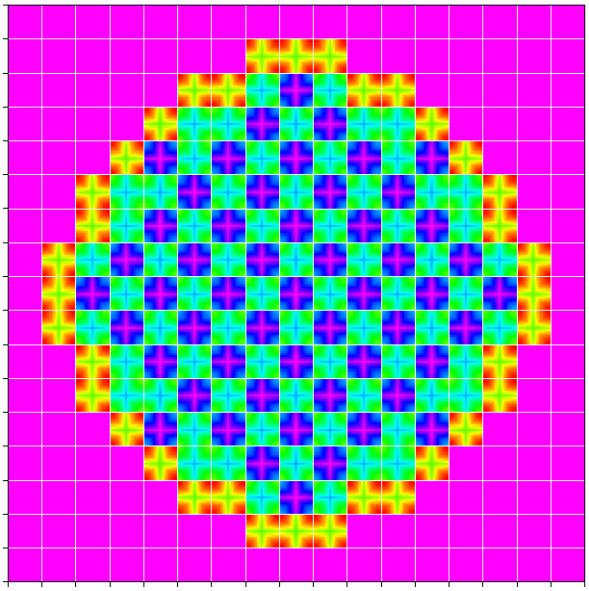
En dynamique moléculaire par exemple, les ingénieurs-chercheurs ne pouvaient simuler que quelques centaines d'atomes sur une centaine de pas de temps dans les années 1960. Progressivement, ils sont arrivés à traiter des cas de 10^4 atomes sur 10^4 pas de temps (1995) et aujourd'hui ils tournent des cas de millions d'atomes sur des millions de pas de temps. Rappelons que la dynamique moléculaire intervient dans le cadre de la simulation des matériaux. Celle-ci peut se faire à différentes échelles (macroscopique, mésoscopique et atomique). La dynamique moléculaire (simulation à l'échelle atomique) consiste à déterminer l'évolution des atomes du système étudié en résolvant les équations de mouvement de tous les atomes. Elle permet de calculer les propriétés des liquides, gaz solides et molécules [Souffez et Domain, 1997]. En termes de temps de calcul (pour un pas de temps et par atome), on est passé de $170\mu s$ sur une machine IBM RS6000 [18] en 1997 à $5\mu s$ sur la frontale du supercalculateur Ivanoe (2013).

La neutronique, cœur de métier par excellence du Groupe EDF, n'est pas en reste. Par exemple, la chaîne de calcul COCAGNE peut traiter un problème comportant jusqu'à mille milliard d'inconnues (10^{12} ddls [19], voir *figure* 1.3). Étant beaucoup plus aboutie et plus complète que la précédente chaîne, ses résultats peuvent servir de référence pour valider les résultats de COCCINELLE, cependant la validation nécessite de traiter beaucoup plus d'inconnues [Guillo *et al.*, 2010a]. Un problème 3D COCCINELLE comportant 1 million d'inconnues sera alors validé par un calcul 3D COCAGNE sur un problème dit « crayon par crayon », correspondant à une discrétisation plus fine et comportant 100 millions d'inconnues. De nombreux travaux sont alors effectués afin de réduire les temps d'exécution de ces codes en profitant des performances

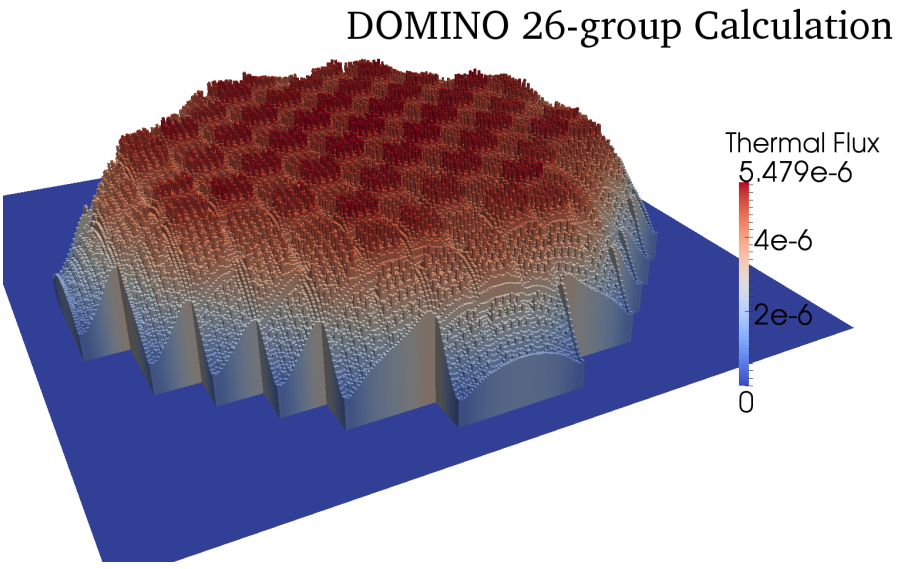
15. http://www.info-tsunami.fr/
16. Voir http://en.wikipedia.org/wiki/2004_Indian_Ocean_earthquake_and_tsunami
17. Voir http://en.wikipedia.org/wiki/2011_Tohoku_earthquake_and_tsunami
18. A titre d'information, la machine IBM RS6000 coûtait 30000 F.F. en 1997
19. ddl = dégré de liberté, 1 ddl = 1 inconnue.

Figure 1.3 – La *figure* 1.3a représente une vue 2D d'un modèle de réacteur de type REP 900 MW. On distingue 3 types d'assemblages caractérisés par 3 niveaux d'enrichissement du combustible (bleu, vert et jaune). Ce modèle est utilisé pour effectuer des calculs cœur et la nappe de flux issue d'un calcul cœur 3D crayon par crayon effectué sur ce modèle avec COCAGNE est donnée par la *figure* 1.3b [Courau *et al.*, 2013].

(a) Réacteur de type REP 900 MW



(b) Nappe de flux issue d'un calcul cœur 3D



offertes par les nouvelles machines. Ces travaux concernent principalement les solveurs neutroniques utilisés dans les chaînes de calcul. Aujourd'hui, on est en mesure de traiter un problème 3D COCAGNE de 10^9 ddls en 11 heures et un autre de 10^{11} ddls en 8 heures. Notons qu'en 2009, ce même type de calcul nécessitait 10000 heures pour traiter un cas de 10^{10} ddls et 100000 heures pour un cas de 10^{12} ddls [Barrault *et al.*, 2011, Kirschenmann *et al.*, 2011, Courau *et al.*, 2013].

L'évolution du Code_Saturne, ces dernières années, illustre également l'impact considérable du calcul haute performance. En 2003, on pouvait traiter des cas de $5 \cdot 10^5$ cellules sur un des vecteurs (processeur vectoriel) de la machine Fujistu VPP 5000 et le calcul durait 2 mois. Aujourd'hui, pour la même durée de calcul avec 2000 cœurs sur un cluster Idataplex IBM, on traite des cas de $2 \cdot 10^8$ cellules. Les temps que nous venons de mentionner concernent des cas traités en production. Pour des cas d'étude d'EDF R&D, il est possible de résoudre des cas de $3 \cdot 10^9$ cellules. Il est intéressant de noter que ce genre de calcul nécessite 5 Tb pour le stockage et 3 Tb de mémoire pendant l'exécution. Malgré toutes ces avancées, des études sont encore menées afin d'améliorer les performances du code [Fournier *et al.*, 2013, Fournier *et al.*, 2012, Fournier *et al.*, 2011].

Bien souvent, dans le but d'optimiser leurs performances, ces codes font appel à des bibliothèques scientifiques externes. Il s'agit de bibliothèques de communication (MPI) ou de calcul (BLAS). Rappelons que les codes de simulations étant conçus pour résoudre des gros systèmes mathématiques, une importante partie des temps d'exécution est due au calcul. L'utilisation de bibliothèques optimisées dédiées au calcul scientifique est le moyen le plus simple pour maximiser les performances d'un code. A titre d'exemple, l'utilisation de la bibliothèque MKL d'Intel, qui est réputée être l'une des meilleures, permet d'améliorer considérablement les performances d'un code.

La simulation numérique, grâce au calcul haute performance, est devenue de *facto*, un outil essentiel de la recherche scientifique, technologique et industrielle. Elle permet de remplacer les expériences qui ne peuvent être menées en laboratoire quand elles sont dangereuses (accidents), de longue durée (climatologie), inaccessibles (astrophysique) ou interdites (essais nucléaires). La simulation améliore également la productivité en procurant un gain de temps important. Au vu de son importance, s'assurer de la qualité des résultats des codes de simulation devient plus qu'obligatoire, et primordial pour les codes industriels. Les travaux de la thèse que nous présentons dans la prochaine section interviennent dans ce cadre là.

1.4 Présentation de la thèse

1.4.1 Motivations

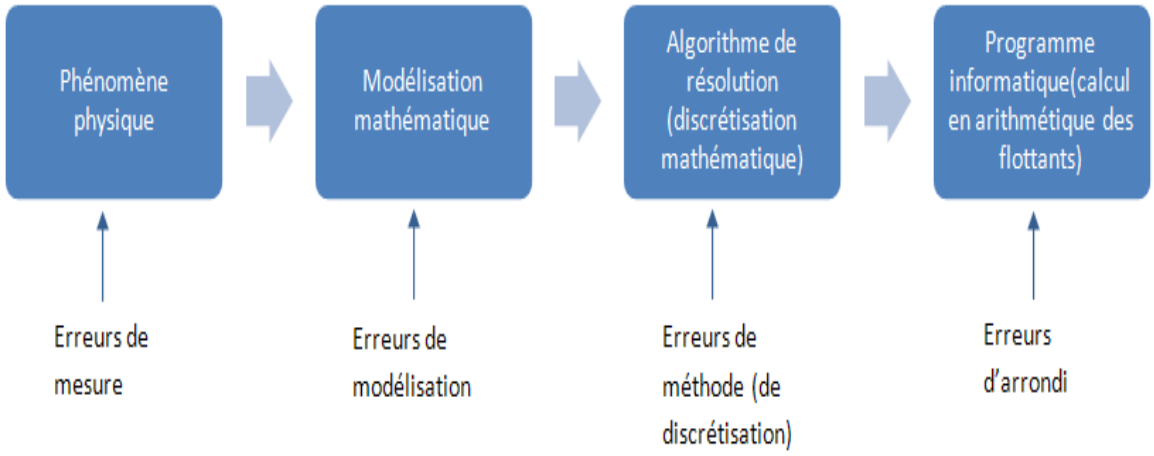
Nous commençons d'abord par rappeler quelques points essentiels de la simulation numérique que nous avons évoqués dans les sections précédentes. Nous insistons particulièrement sur ces points car ils constituent véritablement la motivation fondamentale de notre travail.

1. Le processus de simulation peut être décomposé en quatre grandes étapes : l'observation des phénomènes physiques, la modélisation des lois physiques en équations mathématiques, la discrétisation des équations et pour terminer la résolution numérique des équations grâce aux codes exécutés sur ordinateur. Les équations mathématiques obtenues (souvent des EDPs) sont discrétisées puis résolues (approchées) par des méthodes numériques spécifiques. Rappelons que discrétiser est l'opération consistant à remplacer des relations portant sur des fonctions continues, dérivables, etc, par un nombre fini de

relations algébriques portant sur les valeurs prises par ces fonctions en un nombre fini de points de leur ensemble de définition.

2. Les codes sont exécutés sur des ordinateurs avec une arithmétique propre à l'ordinateur : arithmétique dite flottante qui ne permet pas de représenter exactement tous les nombres réels. Ajoutons aussi que ces programmes sont généralement exécutés dans des environnement HPC où des milliards d'opérations sont effectuées chaque seconde dans un ordre imprédictible.

Figure 1.4 – Les différentes étapes de la simulation numérique



En d'autres termes, il en ressort que plusieurs étapes séparent l'étude du phénomène physique de l'obtention du code de simulation : chaque étape peut être éventuellement source d'erreurs (cf. *figure* 1.4). Les résultats du logiciel de simulation numérique subissent donc plusieurs approximations. Dans le cadre de notre travail, nous nous intéressons aux erreurs dues à la résolution numérique en arithmétique flottante. Il est important de noter, qu'on ne peut effectuer de calcul numérique flottant sans risque de produire des résultats erronés. Pour illustrer nos propos, nous présentons ci-dessous deux problèmes récents rencontrés au cours d'études menées à EDF R&D.

Problème 1. Une étude de quantité de matières dans le cadre d'une réaction d'oxydo-réduction nécessite l'extraction très précise des racines d'un polynôme de degré 3. Une seule de ces racines est admissible, c'est à dire réelle et aboutissant à des quantités de matière et des pressions partielles positives. L'algorithme de Cardan [20], considéré comme stable numériquement, est utilisé pour le calcul des racines. Il a été remarqué au cours de cette étude, qu'avec les types standards (double et long double), on ne pouvait obtenir qu'une seule racine. En augmentant la précision des calculs, on obtient le bon nombre de racines réelles (3), ainsi qu'une amélioration de la précision de la seule racine trouvée auparavant.

20. voir http://fr.wikipedia.org/wiki/Méthode_de_Cardan

Problème 2. Une autre étude nécessitait le calcul d'intégrales de types équation 1.1. Ces intégrales sont calculées avec Maxima [21], logiciel libre de calcul formel. Comme dans l'exemple précédent, il a été remarqué que dès lors qu'on raffinait les mailles, les résultats divergeaient avec les types conventionnels.

$$I = \int_{x_0}^{x_2} w^f(x)\, w_J^c(x)\, dx, \tag{1.1}$$

Ces exemples illustrent, à juste titre, à quel point il est difficile de calculer juste. Le calcul sur les nombres flottants peut facilement engendrer des pertes de précision liées à l'erreur d'arrondi effectuée à chaque opération. Pire, ces pertes sont encore plus importantes dès qu'il s'agit d'exécutions sur des architectures parallèles où l'on ne peut contrôler les ordres des opérations et ceci malgré les différents algorithmes tolérants aux pannes [Bader, 2007, chap. 13]. De fait, il est quasi impossible d'avoir des résultats reproductibles dès lors qu'on travaille sur des architectures parallèles. ***Comment évaluer efficacement, dans un contexte industriel, la qualité des résultats d'un code parallèle de simulation numérique sans pour autant détruire les performances de ce dernier ?*** Telle est la question fondamentale que nous nous posons dans le cadre de cette thèse. Toutefois, ***il ne suffit pas simplement d'évaluer la qualité numérique d'un code, il faut pouvoir l'améliorer dans le cas où elle est jugée insuffisante***.

1.4.2 Contexte et objectifs

Nous définissons par *validation numérique* d'un code, le processus visant à étudier sa fiabilité en termes de qualité numérique. Concrètement, cela consiste à considérer les effets de la propagation d'erreurs d'arrondi sur la qualité des résultats d'un calcul. D'une manière générale, la fiabilité d'un code peut être validée empiriquement par comparaison avec des valeurs de référence (solution analytique, benchmarks, comparaison avec un autre code, etc.). Cependant, dans le cadre de ces travaux, nous ne nous occupons que des problèmes numériques.

L'analyse et la compréhension des effets de la propagation des erreurs d'arrondi permet d'estimer plus précisément la qualité numérique des résultats d'un calcul effectué avec l'arithmétique flottante. Plusieurs méthodes et outils de validation ont été développés pour étudier la propagation d'erreurs d'arrondi. Diverses études ont démontré que la bibliothèque CADNA [Jézéquel *et al.*, 2010, Lamotte *et al.*, 2010] est un outil adapté pour l'étude de la qualité numérique des codes industriels [Scott *et al.*, 2007, Moulinec *et al.*, 2011a]. Elle est assez simple à implémenter sur un code séquentiel écrit dans un seul langage (Fortran90, C/C++). Le principal avantage de CADNA se trouve dans sa capacité de localiser les pertes de précision, d'étudier la précision des résultats intermédiaires et finaux et de proposer un véritable diagnostic à la fin des exécutions. Une étude pertinente de ce diagnostic peut alors aider le numéricien à proposer des améliorations algorithmiques. Insistons également sur un autre point qui fait de l'outil CADNA un formidable outil de travail pour l'industrie : la longueur du cycle de vie des codes de calcul industriel et le fait qu'ils sont pour la plupart conçus par des scientifiques (mécaniciens, neutroniciens, etc) et non par des experts en informatique scientifique. Ce dernier constat justifie l'utilisation du Fortran dans de nombreux codes. Le Code_Aster a par exemple fêté ses 20 ans d'existence en 2009 alors que la première version de la chaîne de calcul COCCINELLE a été mise en place en 1983. Durant toute cette période, le code doit être maintenu, corrigé et amélioré, d'où la nécessité d'un outil facile d'utilisation. Ajoutons aussi que la composante coût financier est un critère non négligeable pour l'industriel.

21. Maxima est un logiciel libre de calcul formel. Il est disponible sous Linux, Mac OS X et Windows. Maxima est sous licence GNU GPL depuis 1998. http://maxima.sourceforge.net/

Mais, l'outil de validation CADNA peut-il être utilisé dans tous les codes ? Répondre positivement à cette question serait une fabulation. En effet, bien souvent et principalement dans le but d'améliorer leurs performances, les codes de calcul scientifique tels que ceux développés à EDF R&D (Code_Aster, Telemac) font appel à des bibliothèques externes d'échange de données (MPI, BLACS) et/ou de calcul scientifique(BLAS, LAPACK) [Denis et Montan, 2012]. La bibliothèque actuelle CADNA, bien qu'étant un outil efficace pour la validation numérique, ne peut donc être utilisée simplement sur des grands codes industriels. Afin d'étudier le comportement numérique d'un code, il faut que celui-ci soit entièrement instrumenté. Il importe donc d'implémenter des extensions compatibles avec l'outil de validation numérique CADNA.

L'implémentation de ces diverses extensions pose un problème de performance, la complexité algorithmique et la taille des logiciels de calcul numérique impliquant d'importants temps d'exécution. Il est donc important de chercher à minimiser l'impact de CADNA sur les performances de ces bibliothèques. A titre d'exemple, l'implémentation directe (naïve) de CADNA dans les BLAS ruine les performances en raison des changements de mode d'arrondi intrinsèques à la méthode et de la nécessité d'éliminer les optimisations. Le but de la thèse est donc de lever ce verrou technologique et scientifique en réécrivant les routines des bibliothèques BLAS, MPI, BLACS et LAPACK utilisant CADNA.

Nous nous proposons, dans le cadre de la thèse, de travailler principalement sur les bibliothèques de communications (MPI et BLACS) et les routines de la bibliothèque de calcul BLAS. Nous travaillerons à étendre les fonctionnalités de ces bibliothèques de façon à les rendre compatibles avec l'arithmétique stochastique discrète, arithmétique implémentée dans l'outil CADNA. Nous nous intéresserons particulièrement à une implémentation efficace de la fonction DGEMM des BLAS intégrant CADNA. Dans un second temps, les bibliothèques développées seront validées et testées dans certaines fonctionnalités des codes industriels d'EDF R&D, en l'occurrence Telemac-2D. Enfin, nous étudierons la qualité numérique de Telemac-2D. Nous ferons une étude comparative des différents algorithmes compensés et montrerons dans quelle mesure ces derniers peuvent améliorer la précision et la qualité des résultats issus d'une exécution d'un code industriel. Nous porterons également une attention particulière à l'impact de ces algorithmes sur les performances des codes.

1.4.3 Environnement

Cette thèse s'inscrit dans le cadre d'une collaboration entre le LIP6 [22] (UPMC [23]) et le département SINETICS [24] d'EDF R&D visant à contrôler la qualité numérique de codes de simulation industriels parallèles. L'aspect novateur de ce travail est de se concentrer sur la qualité numérique d'un code industriel avec une contrainte liée à la performance du code.

1.5 Plan de lecture

Nous tâcherons, au fil des pages de ce manuscrit, de vous présenter nos travaux ainsi que nos principaux résultats. Le chapitre 2 passe en revue les principaux outils et méthodes de validation numérique. L'arithmétique IEEE 754, son implémentation sur les architectures matérielles et ses conséquences pour la simulation numérique y sont présentés. Le processus de validation numérique pour les acteurs industriels y est également décrit. Ensuite, nous présentons l'outil CADNA et la méthode CESTAC qui sont au cœur de ce travail au chapitre 3.

22. Laboratoire de Paris 6 http://www.lip6.fr/
23. Université Pierre Marie Curie http://www.upmc.fr/
24. SImulation en NEutronique, Technologies de l'Information et Calcul Scientifique

Comme nous l'avons souligné plus tôt, ce travail a pour objectif d'implémenter des extensions à l'outil CADNA. Le chapitre 4 est consacré à CADNA_MPI : une extension de CADNA pour le standard de communication MPI. Le chapitre 5 retrace nos activités autour de l'implémentation de l'arithmétique stochastique dans les bibliothèques de calcul scientifique. Ce chapitre s'intéresse particulièrement aux routines BLAS et à sa fonction DGEMM. Les chapitres 6 et 7 présentent la validation numérique d'un code industriel avec l'outil CADNA. Dans un premier temps, CADNA est utilisé pour étudier la qualité numérique du code Telemac-2D. A la suite de cette étude, les algorithmes compensés sont introduits. Nous montrons que ces types d'algorithmes peuvent être mis à contribution pour améliorer la qualité des résultats produits. Un retour d'expérience de l'utilisation de CADNA est fait dans le chapitre 7. Quelques idées et pistes d'amélioration de l'outil sont également exposées dans ce chapitre. Enfin au chapitre 8, nous terminons par un bilan concret et un retour d'expérience du travail effectué au cours ses trois dernières années enrichissantes passées à EDF R&D.

Chapitre 2

La validation numérique : principes et méthodes

Sommaire

Nous avons montré dans le chapitre 1 que le résultat d'un code de simulation numérique subit plusieurs approximations notamment celles dues à la résolution numérique en arithmétique flottante IEEE 754. L'utilisation de cette arithmétique peut influencer fortement la fiabilité des résultats en raison des erreurs d'arrondi qu'elle y introduit. De fait, l'étude de la qualité numérique est cruciale pour les codes industriels tels que ceux développés à EDF R&D.

Dans ce chapitre, nous évoquons les principales notions qui nous semblent être indispensables pour la conception de logiciels de simulation numérique plus sûrs ou plus fiables. Il existe dans la littérature plusieurs ouvrages (et thèses) qui sont consacrés à ces questions. A titre d'exemple, citons le livre [Muller *et al.*, 2010] qui présente de façon approfondie toutes les notions liées aux nombres à virgule flottante et à leur implémentation sur les architectures contemporaines. Cet ouvrage est d'ailleurs considéré à l'heure actuelle (et à juste titre) comme la référence sur ces sujets parce qu'il traite de l'implémentation de l'arithmétique flottante sur les architectures actuelles et des principaux problèmes qui en découlent. Toutefois, les exemples traités dans la littérature se font généralement d'un point de vue académique sans confrontation avec des problèmes issus de l'industrie. Nous nous proposons, dans notre cas, d'apporter une vision industrielle. Notre objectif est de présenter un aperçu succinct de toutes les notions qu'un développeur de logiciel de simulation numérique (code de calcul) doit maîtriser en ce qui concerne l'arithmétique flottante.

Tout au long du chapitre, nous recenserons les ouvrages que nous considérons indispensables à l'approfondissement des notions présentées. Outre l'ouvrage de référence [Muller *et al.*, 2010], nous conseillons également les livres [Higham, 2002] pour les aspects liés à la stabilité et à la précision des algorithmes d'algèbre linéaire et [Kulisch et Ebrary, 2008] pour des aspects plus théoriques sur les erreurs d'arrondi. Le lecteur intéressé y trouvera largement son compte.

Plan du chapitre : Nous commençons d'abord par rappeler les principes de l'arithmétique des ordinateurs puis nous nous intéressons aux erreurs d'arrondi et à ces conséquences sur les résultats des simulations (section 2.1). Nous cherchons également à identifier le(s) véritable(s) responsable(s) de ces erreurs. Dans la seconde partie (section 2.2), nous présentons les principaux outils qui permettent de limiter les erreurs d'arrondi, à leurs plus faibles expressions, sur les résultats des simulations numériques. Enfin, en section 2.3, nous nous intéressons à la validation numérique dans l'industrie.

2.1 Parce que l'ordinateur n'est pas obligé d'être juste [25]

Ainsi, l'ordinateur, ce magnifique outil dont on ne cesse de vanter les mérites n'est pas parfait. Il lui arrive de ne pas calculer juste. Il est d'ailleurs à l'origine de plusieurs tragédies qui ont entraînées des pertes de vie humaines (cas de l'anti-missile Patriot qui a raté l'intersection d'un Scud pendant la guerre d'Irak en 1991) [26]. Parfois, on gagnerait plus à ne pas avoir une

25. *Parce que l'ordinateur n'est pas obligé d'être juste...*. Nous avons choisi de commencer cette partie par ces mots en référence à un célèbre roman d'Ahmadou Kourouma intitulé *Allah n'est pas obligé*. Cette œuvre raconte le périple d'un orphelin Birahima devenu enfant-soldat et de son oncle à travers des pays dévastés par la guerre (Libéria, Guinée, Sierra Leone) et dénonce la cruauté des conditions de vie des enfants-soldats. Le titre a été choisi ainsi car il veut dire qu'« Allah n'est pas obligé d'être juste dans toutes ces choses ici-bas » et ce malgré le nombre astronomique de croyants dans ces pays là.

26. Une liste de catastrophes dus à l'arithmétique flottante est disponible à `http://dutita0.twi.tudelft.nl/users/vuik/wi211/disasters.html`

confiance aveugle en l'ordinateur car même si notre programme est bien écrit on peut tout de même observer des comportements bizarres. A titre d'exemple, considérons le polynôme ci-dessous proposé dans [Rump, 1988] :

$$f(x,y) = 333.75b^6 + a^2(11a^2b^2 - b^6 - 121b^4 - 2) + 5.5b^8 + \frac{a}{2b} \tag{2.1}$$

Tableau 2.1 – Évaluation du polynôme de l'équation 2.1 pour a = 77617.0 et b = 33096.0 sur une machine IBM 370 et sur un poste fixe HP Z600 avec l'extrait de code 2.1 compilé avec GCC 4.6

	IBM 370	Z600
simple précision	1.172603	-9.87501233229925472e+29
double précision	1.1726039400531	-1.18059162071741130e+21
précision étendue	1.172603940053178	6.95334866191693095e-310

Lorsqu'on évalue ce polynôme pour a = 77617.0 et b = 33096.0 sur une machine IBM 370, on obtient les résultats de la première colonne *tableau* 2.1. On pourrait ainsi croire que la simple précision est suffisante. En revanche, cette même évaluation sur notre machine de travail (HP Z600, voir annexe A) donne des résultats complètement contradictoires (*tableau* 2.1 colonne 2). En réalité, le résultat exact est de $-0.827396059946821\ldots$ [Rump, 1988].

Source 2.1 – Évaluation du polynôme de Rump en C

```
1  #include<stdio.h>
2  int main(void)
3  {
4   double a = 77617.0 ;
5   double b = 33096.0 ;
6   double b2,b4,b6,b8,a2,firstexpr,f ;
7   b2 = b*b;
8   b4 = b2*b2 ;
9   b6 = b2*b4 ;
10  b8 = b4*b4 ;
11  a2 = a*a ;
12  firstexpr = 11*a2*b2 - b6 - 121*b4 - 2 ;
13  f = 333.75*b6 + a2*firstexpr + 5.5*b8 + (a/(2.0*b)) ;
14  printf("Resultat DP %1.17e  \n",f)
15  return 0 ;
16 }
```

En soi, ces problèmes ne constituent pas une véritable surprise. Ils ont pour principale origine le passage d'un ensemble de nombres réels $\mathbb{R}$ (ensemble continu) vers l'ensemble des nombres flottants $\mathbb{F}$ (ensemble fini). Dans cette première partie, nous nous posons la même question que se posait David Goldberg : *What every computer scientist should know about floating-point arithmetic* [Goldberg, 1991]. Nous commençons d'abord par présenter l'arithmétique IEEE 754 et ses conséquences sur les résultats des calculs. Nous nous intéresserons ensuite à l'implémentation matérielle de la norme. Nous finirons par une liste non exhaustive des algorithmes critiques.

2.1.1 Du réel au flottant

L'arithmétique des nombres réels sur ordinateur est dite arithmétique à virgule flottante[27]. Un nombre x est représenté en virgule flottante en base $\beta \geq 2$ par un triplet (s, m, e) tel que :

$$x = (-1)^s \cdot m \cdot \beta^e \tag{2.2}$$

où

- $s \in \{0, 1\}$ est son signe (0 pour le positif, 1 pour le négatif) ;
- m est sa mantisse réelle en base β représentée sous la forme $d_0.d_1d_2...d_{p-1}$ et p la précision telle que $0 \leq d_i < \beta$ et donc $0 \leq m < \beta$;
- e son exposant, un entier compris entre e_{min} et e_{max}.

Pendant longtemps, la représentation avait un principe unique, mais il était différent suivant le compilateur, le langage ou l'architecture de la machine. Le résultat d'une opération arithmétique différait alors suivant l'ordinateur ou le langage utilisé. Pour harmoniser les résultats, la norme IEEE 754 a été introduite en 1985 [ANSI/IEEE, 1985], puis révisée en 2008 [IEEE Computer Society, 2008]. Cette norme fixe la représentation des nombres et le comportement des opérations élémentaires pour l'arithmétique à virgule flottante. Cette norme définit également les formats des données, les valeurs spéciales, les modes d'arrondi, la précision des opérations de base et les règles de conversion. Elle a pour objectifs de permettre la conception de programmes portables, de rendre les programmes déterministes d'une machine à une autre, de conserver des propriétés mathématiques et de gérer correctement les arrondis et les conversions.

La norme définit cinq formats de base pour la représentation des flottants :

- en représentation binaire ($\beta = 2$) binary32, binary64, binary128 sur respectivement 32, 64 et 128 bits ;
- en représentation décimale ($\beta = 10$) decimal64, decimal128.

A ces formats s'ajoutent le binary16 (précision moitié) et decimal32 [IEEE Computer Society, 2008]. Le *tableau* 2.2 récapitule la spécification des trois principaux formats binaires. Tous les formats (binaire et décimal) sont présentés dans le document officiel de la norme [IEEE Computer Society, 2008, p.13].

Tableau 2.2 – Les trois principaux formats IEEE 754

Format	Appellation	Répartition des bits signe + mantisse + exposant	e_{min}	e_{max}
binary32	simple précision	$1 + 23 + 8$	-126	127
binary64	double précision	$1 + 52 + 11$	-1022	1023
binary128	quadruple précision	$1 + 112 + 15$	-16382	16383

L'équation 2.2 entraîne plusieurs représentations possibles pour un nombre flottant. Afin d'éviter cela, la représentation *normalisée* a été définie. La mantisse normalisée m du nombre flottant x est représentée par $n+1$ bits : $m = 1.\underbrace{x_1x_2x_3...x_{n-1}x_n}_{f}$ où les x_i sont des bits et f (n bits) la partie fractionnaire de m. On a alors : $m = 1 + f$ et $1 \leq m < 2$. Seule la partie fractionnaire f est stockée physiquement. On parle alors de 1 *implicite*. La norme introduit également les nombres *sous-normaux* (dénormalisés) pour représenter les nombres compris entre 0 et $1 \times 2^{e_{min}}$.

27. Il existe aussi l'arithmétique à virgule fixe, elle est utilisée notamment dans les systèmes embarqués.

En effet, on autorise près de 0 des nombres de la forme $x = (-1)^s \times 0.f \times 2^{e_{min}}$. Dans ce cas de figure, la mantisse ne respecte pas la règle du 1 *implicite* et $e = e_{min}$.

L'exposant e est un entier signé de k bits. La norme IEEE 754 a choisi d'utiliser une représentation biaisée stockée avant la mantisse. Cette représentation permet de faire des comparaisons entre flottants dans l'ordre lexicographique (en laissant de coté le signe s) et de représenter le nombre 0 avec $e = f = 0$. L'exposant stocké physiquement est l'exposant biaisé eb tel que $eb = e + b$ où b est le biais. Les exposants non biaisés $e_{min} - 1$ et $e_{max} + 1$ (respectivement 0 et $2^k - 1$ en biaisé) sont réservés pour zéro, les dénormalisés et les valeurs spéciales. Les valeurs spéciales sont les infinis et NaN(Not A Number). Quelques exemples de représentations sont présentés dans le *tableau* 2.3.

Tableau 2.3 – Quelques exemples de représentation en simple précision (32 bits).

x	s	eb	m
$+0$	0	00000000	00000000000000000000000
$+\infty$	0	11111111	00000000000000000000000
2	0	10000000	00000000000000000000000
6.5	0	10000001	10100000000000000000000
2^{-149}	0	00000000	00000000000000000000001

Notons également, que la norme décrit cinq exceptions :
- *INVALID operation :* en cas d'opération d'arithmétique illicite (comme $\sqrt{-1}$ ou ∞/∞), le résultat est NaN (Not A Number) ;
- DIVIDE by ZERO : en cas de calcul de $x/0$, le résultat est $\pm\infty$;
- OVERFLOW : lorsque le résultat d'une opération est trop grand en valeur absolue en regard de la borne supérieure de la plage des exposants représentables ;
- UNDERFLOW : lorsque le résultat d'une opération est trop petit en valeur absolue en regard de la borne inférieure de la plage des exposants représentables ;
- INEXACT : lorsque le résultat d'une opération ne peut être représenté exactement, et doit donc être arrondi.

Un mécanisme de 5 drapeaux (flag) permet d'informer le système sur le comportement des opérations arithmétiques.

2.1.2 Notion d'arrondi

Il est impossible de représenter exactement tous les nombres avec la norme IEEE 754. Si x et y sont deux nombres représentables en machine, alors le résultat d'une opération entre x et y n'est, en général, pas représentable en machine. On utilise alors une valeur approchée (un arrondi) c'est à dire renvoyer vers un des nombres représentables voisins. L'utilisation de l'arrondi de calcul introduit alors une incertitude dont la propagation peut être source de résultats erronés. Le standard IEEE 754 spécifie quatre modes d'arrondi :
- Arrondi vers $+\infty$: $RU(x)$
- Arrondi vers $-\infty$: $RD(x)$
- Arrondi vers 0 : $RZ(x)$
- Arrondi au plus près : $RN(x)$ le nombre est arrondi à la valeur la plus proche. C'est le mode d'arrondi par défaut.

Les arrondis vers $\pm\infty$ ou 0 sont dits *arrondis dirigés* car ils sont dirigés dans une direction donnée. Une conséquence des différents modes d'arrondis est qu'une fois le mode choisi, le résultat

d'une opération est parfaitement spécifié. On parle alors d'*arrondi correct*. La norme IEEE 754 impose l'*arrondi correct* pour les 4 opérations de base (addition, soustraction, multiplication, division) ainsi que pour la racine carrée. Il est difficile d'obtenir l'arrondi correct pour les autres fonctions mathématiques ($\cos, \sin, \log, \exp \ldots$)[28]. Pour ces fonctions, la norme n'impose rien. Dans ces cas, si y est le résultat exact recherché, on retourne soit $RD(y)$ soit $RU(y)$. On parle de résultat *fidèle* (*faithful result ou faithful rounding*). Il est intéressant de lire à ce sujet, le chapitre 3 de [de Dinechin, 2007] qui est consacré à l'arrondi correct dans la bibliothèque mathématique.

2.1.3 La qualité numérique des calculs

La qualité d'un résultat est généralement confondue avec la précision de celui-ci. Cependant, il convient de distinguer la précision d'un résultat (*accuracy* en anglais) calculé de la précision de calcul (*precision* en anglais) [Higham, 2002]. Pour cela, il est nécessaire de définir l'erreur absolue et l'erreur relative :

Definition 2.1. Soit $\widehat{x}$ une approximation d'un réel x non nul, l'erreur absolue commise pendant l'approximation de x est la quantité $E_a(\widehat{x}) = |\widehat{x} - x|$ et l'erreur relative $E_r(\widehat{x}) = \frac{|\widehat{x}-x|}{|x|}$.

La précision d'un résultat fait référence à l'erreur relative ou absolue commise pendant l'approximation d'une quantité alors que la précision du calcul désigne l'erreur relative commise pendant une opération arithmétique élémentaire ($+, -, \times, \div$). En arithmétique flottante, la précision de chaque opération est majorée par l'*unité d'arrondi* notée **u** ou eps (*machine epsilon*). De fait, la précision de calcul (encore appelée *précision de travail*) est confondue avec l'unité d'arrondi.

Le problème majeur du calcul flottant est la propagation des erreurs d'arrondi. A chaque arrondi, on perd *a priori* un peu de qualité numérique, on parle d'*erreur d'arrondi*. Même si une opération isolée retourne le meilleur résultat possible, une suite de calculs peut conduire à d'importantes erreurs du fait du cumul des erreurs d'arrondi. Un résultat final d'un calcul comprenant plusieurs opérations peut être éloigné de la valeur exacte, voire de signe contraire. Les principales sources d'erreurs d'arrondi sont l'élimination catastrophique, l'absorption et l'accumulation des erreurs.

L'élimination catastrophique (couramment appelée *cancellation*) est le principal problème du calcul flottant. Elle arrive au cours de l'évaluation d'une expression impliquant une soustraction, ou l'addition de deux nombres de signes opposés lorsque les deux nombres sont très proches en valeur absolue. Dans ce cas, la plupart des bits de poids forts des opérandes vont s'annuler entre eux. L'ensemble des bits du résultat final peut être sans rapport avec la valeur que l'on souhaite calculer. En d'autres termes, la valeur du résultat étant petite devant les opérandes, si ceux-ci sont eux-même des résultats de calculs avec des erreurs d'arrondi, l'erreur relative commise sur le résultat final est encore plus grande.

On constate souvent le phénomène d'absorption lors de l'addition de deux nombres ayant des ordres de grandeurs très différents. Le résultat de l'addition peut être assez proche du nombre très grand, on perdra donc la trace du plus petit des deux nombres. En fait, l'absorption apparaît lors d'une addition dont un paramètre est de l'ordre d'un ulp (*Unit in the Last Place*) ou du demi-ulp suivant le mode d'arrondi de l'autre paramètre ou plus petit. Il existe plusieurs définitions de l'ulp dans la littérature mais nous rappelons ci-dessous la définition initiale donnée par William Kahan[29] :

28. http://perso.ens-lyon.fr/jean-michel.muller/Intro-to-TMD.htm
29. Père de l'arithmétique flottante et de la norme IEEE 754. Voir http://www.eecs.berkeley.edu/~wkahan/

Definition 2.2. $ulp(x)$ est la distance entre deux nombres flottants proches de x, même si x est l'un d'eux.

En d'autres termes, l'ulp est la distance entre deux nombres flottants au voisinage de x. La relation entre l'ulp et l'unité d'arrondi **u** est expliquée dans [Muller *et al.*, 2010, section 2.6]. On considère que l'erreur relative commise par l'arrondi au plus près $RN(x)$ est majorée par $1/2ulp$. Pour une représentation flottante en base β avec une précision p, **u** est définie comme ci-dessous :

$$\mathbf{u} = \begin{cases} \frac{1}{2}ulp(1) = \frac{1}{2}\beta^{1-p} & \text{pour le mode d'arrondi au plus près,} \\ ulp(1) = \beta^{1-p} & \text{pour les modes d'arrondis dirigés.} \end{cases}$$

Remarque 2.1.1. La combinaison de ces deux phénomènes (élimination catastrophique et absorption) peut tourner à la catastrophe. Par exemple en simple précision, si $a = 1$ et $b = 2^{-30}$ alors $a + b = 1 \Rightarrow (a + b) - a = 0$.

La propagation des erreurs est également un problème important. Une erreur commise sur un résultat res, peut entraîner une plus grande erreur quand le dernier résultat res est utilisé dans un nouveau calcul. Lorsque les erreurs du premier ordre sont très significatives, elles peuvent engendrer des erreurs de second ordre énormes.

Théoriquement, ces problèmes entraînent la perte des propriétés algébriques (par exemple la perte de l'associativité) et la perte de la notion d'ensemble continu (les réels sont approchés par un ensemble discret) et ont un impact important sur les résultats des algorithmes numériques. On peut également noter une perte de précision dans les calculs en arithmétique flottante (l'ordinateur ne fournit pas la valeur exacte mais une valeur approchée) et des problèmes de branchements conditionnels (le déroulement d'un programme dépend souvent de tests : *Si* $x < y$ *alors* . . . . *Sinon*. . . , la réponse de ces tests peut varier en fonction de la propagation des erreurs d'arrondi).

Nous avons exposé dans cette section les grandes lignes et définit le vocabulaire de l'arithmétique à virgule flottante et de la norme IEEE 754. Les chapitre 2 et 3 de l'ouvrage de Jean-Michel Muller [Muller *et al.*, 2010] présentent ces notions en profondeur. On peut aussi consulter plusieurs pages web dédiées à ce sujet :

- http://www.validlab.com/ ;
- http://www.eecs.berkeley.edu/~wkahan/ ;
- http://perso.ens-lyon.fr/jean-michel.muller/ ;
- http://www.vinc17.org/research/index.fr.html.

2.1.4 Ce que nous devrions tous savoir

Certains évoquent à tort ou à raison le côté obscur de la simulation numérique. En effet, les problèmes numériques ne proviennent pas toujours des programmateurs mais aussi de l'architecture ou du langage de programmation utilisé. Nous nous proposons ici de faire un petit tour d'horizon des concepts (notions) que tout concepteur de logiciel de simulation numérique devrait absolument savoir.

2.1.4.1 Implémentation de l'arithmétique flottante

Nous avons montré dans la section précédente que l'arithmétique à virgule flottante repose sur la norme IEEE 754 qui exige un arrondi correct pour les quatre opérations élémentaires

$(+, -, \times, /)$ et la racine carrée. Dans ces conditions, *qui est responsable de l'implémentation et du respect de la norme sur nos machines ?* En effet, les résultats de l'exécution d'un code dépendent de l'environnement de travail (OS), de l'architecture matérielle sur laquelle est exécuté le code (le processeur), du langage de programmation (C, C++, Fortran), du compilateur utilisé (Intel, GNU GCC, etc) et du développeur qui a écrit le code.

Le processeur : il est le responsable de l'implémentation matérielle des opérations de base de l'arithmétique flottante grâce aux FPU (floating point unit) qui possèdent des registres internes dédiés aux flottants. Il signale les exceptions et s'occupe de l'écriture des résultats en mémoire. De manière générale, les architectures actuelles disposent d'implémentations matérielles pour l'addition, la soustraction, la multiplication, la division et racine carrée pour la simple et la double précision (cf *tableau* 2.2) [Muller *et al.*, 2010, section 3.5]. Sur ces architectures modernes, les opérations en arithmétique flottante sont confiées aux instructions SIMD (Single Instruction Multiple Data). Ce type d'instructions permet de réaliser en un cycle une même opération sur tous les éléments d'un vecteur. Il n'est pas superflu d'ajouter que suivant les architectures matérielles, la taille des registres internes des FPU est différente (64 bits, 96 bits, 128 bits). Cette différence de taille peut engendrer des différences dans les résultats en fonction des ordinateurs. A ces instructions, on ajoutera les FMA (Fused Multiply-and-Add) disponibles sur presque toutes les architectures modernes (ARM, Power, PowerPC, IA64, GPU, processeurs Intel et AMD). Le FMA permet de faire en une seule opération une addition et une multiplication $a \times b + c$ avec un seul arrondi. Plus rapide et plus précis, il permet une meilleure implémentation de la racine carrée et de la division. Cependant, son utilisation dépend du compilateur. Par exemple, par défaut GCC désactive l'utilisation du FMA sauf pour l'addition et la multiplication [30] [De Dinechin, 2013]. Notons que l'utilisation des FMA a été standardisée dans la norme IEEE 754-2008.

Le système d'exploitation (OS) : l'OS s'occupe du traitement des exceptions signalées. C'est également lui qui est en charge du calcul des fonctions ou opérations non gérées par le processeur (par exemple les fonctions mathématiques $\sin, \cos, \log, etc$, la racine carrée sur les architectures récentes). La gestion des nombres flottants incombe aussi à l'OS (précision, choix du mode d'arrondi par défaut, nombres dénormalisés). Dans les récents FPU, le mode d'arrondi peut être géré directement par les instructions assembleurs.

Les Langages de programmation : ils respectent une sémantique d'interprétation bien précise expliquée dans leurs documents de référence. Chaque langage possède sa propre sémantique. L'ordre d'évaluation d'une expression et la précision des résultats intermédiaires dépend du langage utilisé. Par exemple, les documents de référence du Fortran stipulent qu'il respecte uniquement les propriétés mathématiques. En revanche, le langage C impose un sens pour les évaluations et les résultats intermédiaires sont calculés avec la plus grande précision disponible sur l'architecture si les performances ne sont pas impactées. C'est la principale raison pour laquelle, il est conseillé d'utiliser des parenthèses pour obliger les langages à respecter l'ordre d'évaluation que l'on souhaite.

Le compilateur : chaque compilateur offre des centaines d'options de compilation et par défaut ces options servent prioritairement à l'optimisation en termes de vitesse d'exécution. Généralement, ces optimisations se font au dépend de la qualité numérique.

Finalement, "peu" de responsabilités reposent directement sur les développeurs. En effet, plusieurs choix ayant un impact considérable sur la qualité numérique des résultats sont faits à

30. GCC propose l'option *-mno-fused-madd* pour s'assurer de la non utilisation des FMA par le processeur.

l'insu des développeurs. Nous voulons dans cette partie convaincre de l'importance des langages et des compilateurs dans la reproductibilité des résultats de calcul. Il n'existe pas de solution miracle pour avoir des résultats reproductibles, mais le développeur gagnerait beaucoup à porter attention à la documentation du langage qu'il utilise ainsi que des options de compilation auxquelles il fait appel. Nous avons présenté ici un survol de la compatibilité entre la norme IEEE 754 et son implémentation dans les ordinateurs. Ces aspects de la norme sont traités en détail dans les chapitre 3 et 7 de [Muller *et al.*, 2010]. Ce chapitre traite en particulier des spécificités des langages C, C++, Fortran et Java à propos duquel W. Kahan a écrit un article intitulé *How Java's floating-point hurts everyone everywhere* [Kahan *et al.*, 1998].

2.1.4.2 Les algorithmes critiques

Au delà des problèmes liés au respect de la norme, certains algorithmes de base utilisés dans les simulations numériques sont reconnus très instables numériquement. On citera en particulier la sommation, le produit scalaire et l'évaluation polynômiale. La qualité numérique des résultats de ces trois problèmes dépend directement du nombre de conditionnement des paramètres en entrée. En effet, le nombre de cancellations est proportionnel au nombre de conditionnement (introduit en section 2.2.1). Rump a consacré une grande partie de ses travaux à l'obtention d'algorithmes de sommation qui minimisent l'effet des erreurs d'arrondi [Ogita *et al.*, 2005, Rump, 2009]. Un nombre important d'ouvrages a été consacré à ce sujet dans la littérature [Malcolm, 1971, Demmel et Hida, 2004, Zhu et Hayes, 2010, Demmel et Nguyen, 2013]. La sommation est d'autant plus importante puisqu'à partir d'un algorithme de sommation précis, on obtient un algorithme de produit scalaire d'une meilleure précision, un meilleur algorithme pour le produit de matrices et les différents algorithmes d'algèbre linéaire faisant appel au produit scalaire. En ce qui concerne les évaluations polynomiales, on peut se référer à la thèse [Louvet, 2007]. Ces travaux sont pour la plus part basés sur le concept d'*algorithmes compensés* que nous introduisons dans la section 2.2 et que nous mettons en pratique au chapitre 6. Le livre [Higham, 2002] est la référence à lire en ce qui concerne la stabilité numérique d'algorithmes d'algèbre linéaire.

2.2 L'ordinateur peut être source d'erreur... et alors ?

En se basant sur les notions exposées dans la section 2.1, il est évident qu'il est impossible d'éviter la propagation des erreurs d'arrondi dans les calculs flottants. Il faudrait sinon se passer de l'ordinateur et travailler uniquement sur des systèmes complètement fiables. Encore faudrait-il que de tels systèmes existent et si oui qu'ils soient capables de faire de grandes simulations complexes comme c'est le cas aujourd'hui [Muller *et al.*, 2010, section 7.6]. Dans ce contexte, le mieux que l'on puisse faire est de borner l'erreur finale afin d'estimer la qualité numérique de notre résultat (section 2.2.1). L'analyse et la compréhension des effets de la propagation des erreurs d'arrondi permettent d'estimer plus précisément la qualité numérique des résultats d'un calcul à arithmétique flottante. Une autre solution envisageable pour limiter les conséquences de la propagation des erreurs d'arrondi est la mise en œuvre des techniques qui permettent d'améliorer la précision des résultats calculés (section 2.2.2). Outre l'analyse et l'amélioration de la précision, on peut aussi envisager de détecter les erreurs afin d'y proposer des solutions (section 2.2.3).

2.2.1 Analyser l'erreur

Dans le cadre de l'analyse d'erreur, on peut citer l'analyse directe et l'analyse inverse [Wilkinson, 1963]. L'idée est d'analyser les erreurs commises afin de borner les résultats. Trois notions sont indispensables pour cela : l'erreur inverse, l'erreur directe et le nombre de conditionnement.

L'analyse directe et l'analyse inverse

En général une fonction f est évaluée en un point grâce à un algorithme numérique. Dans les faits, cet algorithme ne calcule qu'une approximation du résultat (section 1.4.1). Ainsi au lieu de calculer $y = f(x)$, il calcule $\widehat{y} = \widehat{f}(x)$.

L'analyse d'erreur directe a pour objectif de majorer la distance séparant le résultat calculé $\widehat{y}$ du résultat exact y. Cette distance est appelée erreur directe. Elle peut être majorée de façon relative ou absolue selon qu'on considère l'erreur absolue ou relative. L'analyse inverse est basée sur le principe de Wilkinson qui considère le résultat calculé $\widehat{y}$ juste et cherche à identifier une valeur Δ_x telle que $\widehat{y} = f(x + \Delta_x)$. Δ_x constitue alors l'erreur inverse commise lors de la résolution du problème. On retiendra principalement que l'analyse d'erreur directe apporte des éléments de réponses à la question *quelle est la précision du résultat calculé par l'algorithme numérique considéré ?* alors que l'analyse d'erreur inverse tente de répondre à la question *pour quel jeu de données le problème a t'il effectivement été résolu ?* [Louvet, 2007, p. 15]. Ajoutons qu'un algorithme est reconnu inverse stable s'il fournit la solution exacte du problème pour un jeu de données entachées de petites perturbations. C'est le cas en arithmétique flottante quand l'erreur inverse est de l'ordre de l'unité d'arrondi $\mathbf{u}$.

Le nombre de conditionnement permet de faire le lien entre l'erreur directe et l'erreur inverse grâce à la relation ci-dessous définie dans [Higham, 2002, p 9] :

$$\text{erreur directe} \leq \text{nombre de conditionnement} \times \text{erreur inverse} \tag{2.3}$$

Le nombre de conditionnement permet de quantifier l'effet d'une perturbation des paramètres d'entrée d'un problème sur le résultat du problème. Plus il est grand, plus une petite perturbation des données induit une erreur importante sur la solution. Pour un nombre de conditionnement grand, on parle alors d'un problème mal conditionné [Louvet, 2007, section 2.2]. En considérant les mêmes notations que celles utilisées au paragraphe précédent et si f est dérivable, le nombre de conditionnement (relatif) $c(x)$ est définie par :

$$c(x) = \left| \frac{xf'(x)}{f(x)} \right|$$

Dans le cas particulier de l'arithmétique flottante, si on utilise un algorithme inverse stable, l'équation 2.3 est équivalente à :

$$\text{précision} \leq \text{nombre de conditionnement} \times \mathbf{u} \tag{2.4}$$

Il importe alors de remarquer que pour un nombre de conditionnement supérieur à $\mathbf{u}^{-1}$, la précision du résultat ne peut être inférieure à 1. De ce constat, on déduit qu'un problème est bien conditionné à la précision de travail $\mathbf{u}$ si son nombre de conditionnement est très petit devant $\mathbf{u}^{-1}$. A titre d'exemple, le nombre de conditionnement d'une somme de nombre flottant $\sum p_i$ est :

$$cond(\sum p_i) = \frac{\sum |p_i|}{|\sum p_i|}$$

En double précision IEEE 754, la somme est dit mal conditionnée pour un nombre de conditionnement supérieur à 2^{53}. Dans le pire des cas, aucun digit du résultat calculé n'est correct [Ogita *et al.*, 2005]. Les principes de l'analyse inverse sont utilisés dans la bibliothèque LAPACK [Anderson *et al.*, 1993] pour borner l'erreur commise sur les résultats de calcul. Les ouvrages [Higham, 2002, Langlois, 2001a, Wilkinson, 1994, Rump, 2005] permettent de se faire une meilleure idée de l'analyse de l'erreur. L'arithmétique d'intervalles et l'approche probabiliste de la méthode CESTAC font aussi appel à des notions d'analyse d'erreur.

L'arithmétique d'intervalles

Dans l'arithmétique d'intervalles, les nombres sont représentés par des intervalles (un nombre réel x est représenté par un intervalle $[\underline{x}, \overline{x}]$ englobant x). L'arithmétique d'intervalles permet de transposer les opérateurs de l'arithmétique flottante sur les intervalles grâce à la propriété suivante :

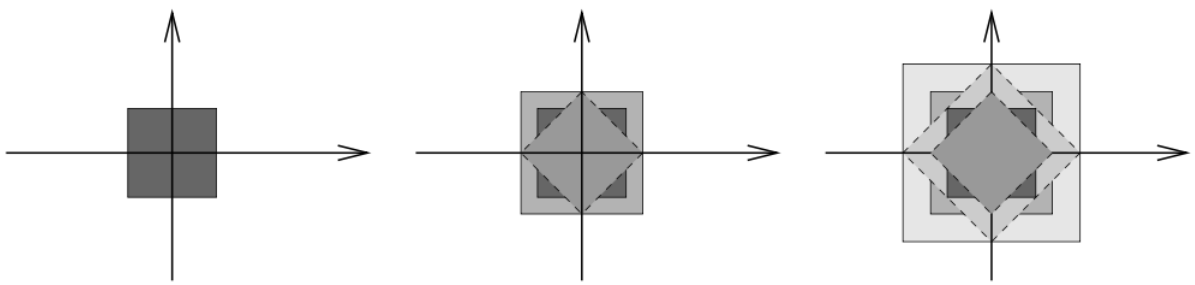
Soit $*$ un opérateur arithmétique défini sur $\mathbb{R} \times \mathbb{R}$, et soient I_x et I_y deux intervalles de $\mathbb{I}_\mathbb{R}$ l'ensemble des intervalles définis sur les réels :

$$\forall I_x, I_y \in \mathbb{I}_\mathbb{R}, x \in I_x \wedge y \in I_y \Rightarrow x * y \in I_x * I_y$$

Les principales relations arithmétiques sont définies de la façon suivante :
- $-[a,b] = [-b,-a]$;
- $[a,b]^{-1} = [\frac{1}{b}, \frac{1}{a}]$ si $0 \notin [a,b]$;
- $|[a,b]| = [max(0,a,-b), max(-a,b)]$;
- $[a,b] + [c,d] = [a+c, b+d]$;
- $[a,b] - [c,d] = [a-d, b-c]$;
- $[a,b] \times [c,d] = [min(a \times c, b \times c, a \times d, b \times d), max(a \times c, b \times c, a \times d, b \times d)]$;
- $[a,b]/[c,d] = [a,b] \times [c,d]^{-1}$.

Les livres [Moore *et al.*, 2009, Rump, 2005] constituent des références sur le sujet. On peut également consulter la page web de Nathalie Revol `http://perso.ens-lyon.fr/nathalie.revol/` ou la page `http://www.cs.utep.edu/interval-comp/main.html` qui sont dédiées à l'arithmétique d'intervalle. Cette arithmétique a été implémentée dans plusieurs bibliothèques

Figure 2.1 – Effet d'enveloppement en arithmétique d'intervalles : après deux rotations successives de $\pi/4$ du petit carré central, on ne retrouve pas le petit carré central mais le grand carré [Revol, 2004].



et logiciels dont C-XSC [31] qui définit une classe C++ [Hofschuster et Krämer, 2004] et une extension de Matlab appelée INTLAB [32] [Rump, 1999].

Le principal inconvénient de l'arithmétique d'intervalles est la surestimation des intervalles et donc une surestimation des erreurs. En effet, les calculs effectués dans l'arithmétique d'intervalles provoquent un élargissement mécanique des intervalles qui s'explique principalement par deux raisons :

- le phénomène d'enveloppement ou *wrapping effect* : nous illustrons ce phénomène dans la *figure* 2.1.
- la décorrélation (ou dépendance) des données : ce phénomène est une conséquence directe de la façon de calculer dans cette arithmétique. A titre d'exemple, considérons un intervalle $I = [0, 20]$, si on calcule l'expression $I - I$, on aura $I - I = [0 - 20, 20 - 0] = [-20, 20]$. Or, en développant un minimum cette expression, on aurait eu : $I - I = I \times ([1, 1] - [1, 1]) = I \times [0, 0] = [0, 0]$. Le résultat dépend directement de l'expression utilisée.

En résumé, retenons que l'arithmétique d'intervalles a pour principal avantage de garantir une borne pour les résultats. En revanche, elle nécessite la réécriture les codes pour chaque problème et elle est très difficile à utiliser sur des codes industriels déjà écrits excepté sur de petites sections du code. De surcroît, elle nécessite parfois une modification des algorithmes.

L'approche probabiliste

L'approche probabiliste se base sur la méthode CESTAC (Contrôle et Estimation STochastique des Arrondis de Calculs) [Vignes et La Porte, 1974]. L'idée de cette méthode est de propager aléatoirement les erreurs d'arrondi en faisant chaque opération arithmétique n fois avec un mode d'arrondi choisi aléatoirement. En pratique, les opérations sont effectuées 2 ou 3 fois. A partir des n échantillons du résultat, on détermine le résultat final ainsi que sa précision en termes de nombre de chiffres significatifs exacts.

Grâce à cette méthode, on définit l'*arithmétique stochastique discrète (DSA)* [Chesneaux et Vignes, 1992, Vignes, 2004] qui est implémentée dans la bibliothèque CADNA [Chesneaux, 1988, Jézéquel *et al.*, 2010, Lamotte *et al.*, 2010]. Simple à utiliser dans un code séquentiel (F90, C et C++), CADNA n'est pas très intrusive, ce qui est un atout non négligeable en particulier pour une utilisation industrielle [Scott *et al.*, 2007, Moulinec *et al.*, 2011a]. Le principal avantage de CADNA est sa capacité à localiser les pertes de précision et les différentes instabilités numériques au cours d'une exécution. CADNA permet d'étudier la précision des résultats intermédiaires et finaux et de proposer un véritable diagnostic à la fin des exécutions. Une étude pertinente de ce diagnostic peut alors aider l'ingénieur à proposer des améliorations algorithmiques. En revanche, il présente pour inconvénient de fournir un résultat en probabilité. Dans le chapitre 3, nous revenons en détail sur l'implémentation de la DSA et le fonctionnement de l'outil CADNA.

2.2.2 Éviter l'erreur

Une autre manière de concevoir les problèmes liés à la propagation des erreurs d'arrondi est de considérer que les précisions de travail (simple ou double précision) ne sont pas suffisantes et essayer d'améliorer la qualité des résultats en augmentant la précision de travail.

31. www.xsc.de
32. www.ti3.tu-harburg.de/rump/intlab

Ainsi, lorsque la double précision IEEE 754 n'est pas suffisante pour garantir une précision donnée, diverses solutions existent pour augmenter la précision des calculs. On peut par exemple utiliser une arithmétique exacte permettant de ne jamais perdre en précision. Ce type d'arithmétique utilise une représentation rationnelle des nombres. C'est ce qu'implémente la bibliothèque GMP [33] (GNU Multiple Precision arithmetic library) [Granlund, 1996]. Toutefois, il est difficile d'utiliser ce type d'arithmétique dans le calcul scientifique. En effet, son utilisation se révèle très coûteuse en termes de temps de calcul et le résultat d'un calcul rationnel n'est pas forcément rationnel. Il importe aussi de noter que, dans certains cas, l'augmentation de la précision ne fait que repousser le problème de l'arrondi plus loin, cette augmentation ne résout pas totalement le problème.

Il est également possible de se servir de la quadruple précision introduite avec la révision de la norme IEEE 754. Finalement, la solution à laquelle on fait souvent appel est l'arithmétique multiprécision qui permet à l'utilisateur de choisir une précision de calcul arbitrairement élevée. Les exemples de bibliothèques de calcul flottant multiprécision les plus connues sont MP de Brent [34] [Brent, 1978], ARPREC [35] [Bailey *et al.*, 2002] de Bailey et MPFR [36] [Fousse *et al.*, 2007] développée en France par l'INRIA. La bibliothèque MPFR, basée sur GNU MP, propose les quatre modes d'arrondis pour les opérations élémentaires, ainsi que les principales fonctions élémentaires avec arrondi correct. On peut également citer la bibliothèque MPFI (Multiple Precision Floating-point Interval) [37] [Revol et Rouillier, 2005] qui permet de combiner l'arithmétique d'intervalles et l'arithmétique multiprécision.

Des travaux ont aussi été réalisés pour obtenir des arithmétiques qui simulent deux fois (double double) ou quatre fois (quad-double) la précision de travail. On peut citer à ce propos les travaux de Bailey dans le cadre de la bibliothèque QD [Bailey *et al.*, 2010]. La bibliothèque XBLAS (Extended and Mixed Precision BLAS) [Li *et al.*, 2002] implémente l'arithmétique double double.

Nous ne pouvons citer les outils pour améliorer la précision des résultats de calcul sans évoquer les algorithmes compensés. Ces méthodes consistent à estimer l'erreur commise et à l'ajouter au résultat calculé [Malcolm, 1971, Demmel et Hida, 2004, Rump, 2009, Zhu et Hayes, 2010] et présente l'avantage de n'utiliser que la précision de travail offerte par l'architecture. En effet, on considère que l'erreur d'arrondi commise en calculant l'addition de $fl(a+b)$ est elle-même un nombre flottant δ tel que :

$$x + \delta = a + b, \quad \text{avec} \quad x = fl(a+b) \quad \text{et} \quad \delta \quad \text{un nombre flottant} \tag{2.5}$$

A partir de ce résultat, on obtient des algorithmes de sommation plus sûrs. Une telle transformation est appelée *error-free transformation* [Ogita *et al.*, 2005]. Citons également la méthode CENA, développée par Langlois. Le principe de la méthode CENA est de calculer une linéarisation de l'erreur d'arrondi et de l'ajouter au résultat calculé [Langlois, 2001b].

2.2.3 Détecter l'erreur

Détecter les erreurs permet aussi de les éviter. Il existe aujourd'hui de nombreuses méthodes qui permettent d'estimer les erreurs d'arrondi dues à l'arithmétique IEEE 754. Citons

33. http://gmplib.org
34. http://wwwmaths.anu.edu.au/~brent/pub/pub043.html
35. http://crd-legacy.lbl.gov/~dhbailey/mpdist/
36. http://www.mpfr.org/
37. http://mpfi.gforge.inria.fr/

par exemple les méthodes issues du domaine de l'interprétation abstraite comme celles utilisées dans l'outil Fluctuat [Goubault *et al.*, 2008] du CEA qui permettent de calculer des surapproximations des erreurs générées.

L'interprétation abstraite permet grâce à une analyse statique (i.e. sans l'exécuter) de prouver des propriétés d'un programme qui sont valables pour toutes les exécutions possibles de celui-ci. Il est nécessaire pour cela de définir une sémantique collectrice qui regroupe toutes les sémantiques pour tous les environnements. Prouver qu'un programme est correct revient à vérifier que la sémantique collectrice de ce programme est incluse dans la spécification (l'ensemble des états valides) du programme [Ioualalen, 2013, chapitre 3].

Les travaux de Martel ont été consacrés à l'interprétation abstraite et à son application à l'amélioration de la qualité numérique des programmes. Il a présenté dans [Martel, 2006] une sémantique qui permet de définir un domaine abstrait qui calcule une approximation des erreurs d'arrondi introduites par l'arithmétique à virgule flottante durant l'évaluation d'une expression. Une autre sémantique présentée toujours par Martel permet de formaliser une abstraction qui a pour but d'optimiser automatiquement la précision numérique d'un programme [Martel, 2007]. Cette dernière sémantique montre qu'il est possible de se servir de la sémantique de propagation d'erreurs d'arrondi afin de synthétiser un nouveau programme dont l'erreur d'arrondi est plus faible [Martel, 2009].

Outre l'interprétation abstraite, des méthodes stochastiques et des techniques issues du domaine de la différentiation automatique (A.D. [38]) [Bischof *et al.*, 2008,Griewank et Walther, 2008] de code peuvent aussi être utilisées pour estimer les erreurs d'arrondi et les réduire [Braconnier et Langlois, 2002]. Le logiciel CADNA, que nous présentons dans le chapitre 3, implémente l'approche stochastique que nous avons introduit en section 2.2.1.

L'idée de la différentiation automatique est de considérer un programme comme une fonction mathématique définie par composition de fonctions élémentaires $(+, -, \times, \div)$ et d'en calculer sa dérivée. Cette méthode s'appuie sur la règle de la dérivation en chaîne : si f et g sont deux fonctions dérivables alors $f \circ g$ est différentiable telle que $(f \circ g)' = g'(f' \circ g)$ [Chapoutot *et al.*, 2009, section 2.5]. La méthode permet, par le biais des valeurs différentielles, une analyse de dépendance entre les variables présentes dans le programme et ainsi d'en déduire une approximation des erreurs. Deux implémentations sont possibles pour la méthode de différentiation automatique : par surcharge d'opérateurs (basée sur le concept d'objet des langages de programmation) et par transformation de code source. On retrouve la première implémentation dans la bibliothèque ADOL-C [Walther et Griewank, 2012] dédiée au langage C++. Ce type d'implémentation a pour avantage d'être facilement utilisable grâce aux objets mais dégrade les performances. La transformation de code source est utilisée dans les bibliothèques ADIC [39] [Bischof *et al.*, 1997], TAPENADE [40] [Hascoët et Araya-Polo, 2006] pour les langages C/C++ et le Fortran. La transformation de code s'appuie sur des concepts issus de la compilation. L'idée est de modifier le programme initial en y ajoutant des instructions permettant le calcul des dérivées.

2.3 La validation numérique en industrie

La validation numérique en industrie est une partie importante de la conception des outils de simulation numérique. Elle rentre dans de cadre du processus V&V (vérification et valida-

38. Automatic Differentiation, voir http://www.autodiff.org/
39. http://www.mcs.anl.gov/research/projects/adic/
40. http://www-sop.inria.fr/tropics/tapenade.html

tion). La vérification analyse les comportements du logiciel par rapport aux spécifications préétablies. La validation a pour objectif de confronter les résultats des simulations à des modèles ou à des mesures. En fait, par abus de langage, les termes *vérification* et *validation* sont souvent utilisés comme synonymes. Toutefois, il existe une petite nuance. La vérification numérique consiste à vérifier la qualité numérique. La vérification permet la validation de la qualité numérique.

Les outils et méthodes que nous avons cités dans les sections précédentes peuvent tous servir dans ce processus mais à des étapes différentes. En ce qui concerne les erreurs d'arrondi, nous résumons le processus de vérification numérique en cinq étapes indépendantes. On considère ici que la modélisation mathématique a déjà été faite.

1. Analyse des algorithmes : la première étape dans la conception d'un code de calcul est le choix des algorithmes à implémenter. Les algorithmes doivent être choisis en fonction des problèmes physiques traités. Il est nécessaire de faire une analyse d'erreur (directe, inverse) de chaque algorithme afin d'étudier sa stabilité. Les principaux algorithmes d'algèbre linéaire ont déjà été l'objet de nombreuses études [Higham, 2002]. Il suffira alors de consulter la littérature à sujet. Il est aussi possible pour les problèmes d'algèbre linéaire d'implanter directement dans le code des routines qui permettent d'estimer l'erreur directe ou inverse [Baboulin *et al.*, 2012].
2. Implémentation des algorithmes : au cours de cette phase, il est nécessaire de faire attention aux spécificités du langage de programmation choisi et de l'architecture matérielle cible.
3. Vérification : l'objectif ici est de vérifier la reproductibilité et l'exactitude des résultats suivant les plateformes de test. On peut faire appel à l'arithmétique des intervalles, l'arithmétique multiprécision ou l'arithmétique stochastique.
4. Détection des erreurs : dans le cas où l'étape vérification n'a pas été jugée concluante, il faut identifier les sources d'erreurs. On peut utiliser dans ce cadre les outils cités dans la section 2.2.3.
5. Améliorer la précision des résultats : après l'identification des problèmes, il faut chercher à les éliminer (ou réduire leurs effets). La première solution serait alors d'améliorer la précision en utilisant tout outil susceptible d'augmenter la précision de calcul (arithmétique multiprécision, algorithmes compensés). Une autre solution possible est de ré-écrire les parties des codes entachées de problèmes en respectant autant que possible les spécificités du langage de programmation.

D'un point de vue industriel, le coût du processus de vérification est un paramètre très important. Par coût, nous sous-entendons temps de développement ou de mise en œuvre de la méthode de validation, coût de la licence logiciel et temps d'exécution des versions de code intégrant les outils de validation. C'est la raison pour laquelle, l'industriel privilégiera, du moins dans un premier temps, les outils les moins intrusifs (sans modification de codes sources), les moins onéreux (open source) et ceux qui demande un minimum d'effort pour la prise en main. Cette notion de coût justifie également l'utilisation de bibliothèques externes pendant la conception des logiciels de simulation. En fait, il s'agit d'une mutualisation des coûts. Toutefois, nous faisons remarquer que même si l'aspect financier est important, l'industriel est prêt à payer quand il a une garantie de la qualité du service fourni et si le service répond entièrement à ses besoins.

Pour être plus complet sur la validation numérique dans l'industrie, elle ne se fait bien souvent qu'après la détection de problèmes de reproductibilité. L'idée est de détecter puis de pro-

poser des solutions. Certains outils comme Fluctuat peuvent aussi être utilisés dans un cadre industriel [Delmas *et al.*, 2009]. Fluctuat présente pour principal inconvénient d'être propriété du CEA et ne fonctionne que sur les codes développés en C ou ADA, ceci limite son utilisation pour les codes industriels écrits en Fortran. On peut également cité l'outil Astree[41] qui a été utilisé pour valider le logiciel de commande de vol électrique primaire de l'A340 et de l'A380 [Bouissou *et al.*, 2009]. Astree a été principalement conçu pour les systèmes embarqués et ne traite que des problèmes qui surviennent à l'exécution. Enfin, il y a aussi l'outil CADNA qui est utilisé à EDF R&D. Cet outil a pour principal avantage d'être disponible en C/C++ et en Fortran, qui sont deux langages souvent utilisés pour le calcul scientifique. En outre, l'outil peut être facilement implémenté dans un code surtout si ce dernier est développé en C++. CADNA permet de localiser les problèmes numériques et propose un diagnostic à la fin des exécutions. Nous présentons cet outil dans le prochain chapitre.

Des outils ou méthodes sont ensuite nécessaires pour corriger les erreurs détectées lors de la vérification numérique. Dans ce cas, l'industriel a le plus souvent recourt aux outils d'arithmétique multiprécision et notamment MPFR. A titre d'exemple, c'est cette bibliothèque qui a été utilisée dans les deux cas présentés en section 1.4.1.

2.4 Conclusion

Nous avons exposé dans ce chapitre les principales notions de la validation numérique. En effet, les résultats des calculs sur ordinateurs peuvent être erronés du fait de l'arithmétique flottante. La norme IEEE 754-1985 et sa révision de 2008 ont permis de créer un cadre pour les calculs sur les nombres à virgule flottante. La norme exige un arrondi correct pour les quatre opérations élémentaires $(+, -, \times, /)$ et la racine carrée. Cependant, pour des raisons de performance, la norme n'est pas toujours strictement respectée (section 2.1). En fait, les résultats de l'exécution d'un code dépendent principalement de l'environnement de travail (OS), de l'architecture matérielle sur laquelle est exécuté le code (le processeur), du langage de programmation (C, C++, Fortran) et du compilateur utilisé (Intel, GNU GCC, etc). Nous montrons en section 2.1.4 que très peu de responsabilités reposent sur les développeurs. Néanmoins, ils gagneraient beaucoup à porter attention à la documentation des langages de programmation ainsi que des options de compilation auxquelles ils font appel.

En résumé, il est impossible d'éviter la propagation des erreurs d'arrondi dans les calculs flottants. Le mieux que l'on puisse faire est mettre en place des méthodes qui permettent d'estimer la qualité numérique des résultats finaux. Un état de l'art des principaux outils et méthodes de validation numérique a été proposé en section 2.2. Ces méthodes sont résumées ici en trois classes :

i. Analyse des erreurs pour ensuite estimer l'erreur totale commise ;

ii. Détection des erreurs pour proposer des solutions d'amélioration ;

iii. Diminution des erreurs en augmentant ou en simulant des précisions de travail élevées.

Dans l'industrie, la validation numérique des codes de calculs ne se fait bien souvent qu'après la détection de problèmes de reproductibilité. Nous montrons dans la section 2.3 que cette validation peut se faire en plusieurs étapes indépendantes. Cependant, dans les faits, elle ne se fait principalement qu'en deux étapes : détection des erreurs et mise en place de solutions pour améliorer la qualité numérique. Notons également que la validation numérique des codes

41. http://www.astree.ens.fr/

industriels répond à des critères de coût et de performance. En ce sens, il a été montré que l'outil CADNA basé sur l'approche stochastique d'analyse d'erreurs est adapté pour la détection des instabilités numériques dans un code industriel. Outre le fait d'être disponible en C/C++ et Fortran, cet outil a pour principal avantage de localiser les problèmes numériques et de proposer un diagnostic à la fin des exécutions. La bibliothèque CADNA ainsi que la méthode CESTAC qu'elle implémente sont présentées en détail dans le prochain chapitre.

Chapitre 3

L'arithmétique stochastique discrète et son implémentation

Sommaire

Après une présentation des méthodes et outils de validation numérique dans le chapitre 2, nous présentons ici le logiciel CADNA développé par le Laboratoire d'Informatique de Paris 6 (LIP6). Ce logiciel (bibliothèque) est au centre de tous les travaux présentés dans ce document. CADNA a pour objectif de répondre à la question suivante : *What is the computing error due to floating point arithmetic on the results produced by any program running on a computer ?* Issu des travaux de la thèse [Chesneaux, 1988], il a d'abord été développé en ADA et les premières versions du logiciel était payante. Il est aujourd'hui disponible en C/C++ [Lamotte *et al.*, 2010] et Fortran [Jézéquel *et al.*, 2010] sous une licence open source [42]. La bibliothèque CADNA est une implémentation synchrone de l'arithmétique stochastique discrète qui trouve ses fondements dans la méthode CESTAC [Vignes et La Porte, 1974].

Plan du chapitre : Après quelques rappels sur la formalisation de l'erreur d'arrondi dans les calculs (section 3.1), nous introduisons la méthode CESTAC et l'arithmétique stochastique discrète (section 3.2). Nous présentons ensuite son implémentation dans CADNA en section 3.3.

3.1 L'erreur d'arrondi dans un programme informatique

Nous présentons ici une formalisation du calcul informatique. L'idée est de modéliser l'expression d'un calcul sur ordinateur par une formule mathématique exacte et d'identifier à chaque étape l'erreur commise par rapport au résultat exact (i.e. la quantité perdue du fait de la troncature ou de l'arrondi). Nous nous sommes largement inspiré de [Chesneaux, 1988] et [Chesneaux *et al.*, 2009]. Les notions présentés ici sont issues de ces travaux. Ces notions sont démontrées dans le chapitre 2 de [Chesneaux, 1988].

3.1.1 Notion de chiffres significatifs exacts

Avant d'aborder les questions d'erreurs d'arrondi dans les opérations arithmétiques, il nous est nécessaire de formaliser, dans un premier temps, la notion de chiffres significatifs exacts. Considérons R le résultat d'un calcul (résultat informatique) et r sa valeur exacte. Le nombre $C_{R,r}$ est défini comme le nombre de chiffres communs entre R et r tel que :

$$C_{R,r} = \log_{10} \left| \frac{R+r}{2(R-r)} \right| \tag{3.1}$$

L'équation 3.1 est équivalente à

$$|R-r| = \left| \frac{R+r}{2} \right| 10^{-C_{R,r}} \tag{3.2}$$

Si $C_{R,r} = 3$, l'erreur relative entre R et r est de l'ordre de 10^{-3}, ce qui signifie que R et r auront trois chiffres décimaux en commun. Précisons que lorsqu'on parle de chiffres significatifs, on pense à la représentation naturelle des nombres (c'est à dire la représentation en base 10).

Remarque 3.1.1. Certaines fois la valeur de $C_{R,r}$ peut réserver quelques surprises. Considérons par exemple $R = 2,45999999764$ et $r = 2,46000000123$, on obtient $C_{R,r} \approx 8,8358$. La différence entre les 9 et les 0 est illusoire. Les chiffres décimaux significatifs ne deviennent véritablement différents qu'à partir du neuvième.

42. Licence Paris6

3.1.2 Erreurs d'arrondi à chaque opération arithmétique

Reprenons les notations de l'équation 2.2. Soit X la représentation IEEE 754 d'un réel x. On peut alors écrire :

$$X = \varepsilon \cdot M \cdot 2^E \quad \text{et} \quad X = x - \varepsilon \cdot 2^{E-p} \cdot \alpha \tag{3.3}$$

où $\varepsilon = (-1)^s$ dans l'équation 2.2, $2^{E-p} \cdot \alpha$ représente l'erreur absolue faite sur la mantisse finie M lors de la représentation de x, p étant le nombre de bits de cette mantisse (bit caché compris). α représente l'erreur d'arrondi normalisée :

- en arrondi au plus près, $\alpha \in [-0.5; 0.5[$;
- en arrondi vers zéro, $\alpha \in [0; 1[$;
- en arrondi vers $-\infty$ ou $+\infty$, $\alpha \in [-1; 1[$;

Soit x_1 et x_2 deux nombres réels représentés sur l'ordinateur par X_1 et X_2. On a alors

$$X_i = x_i - 2^{E_i-p} \cdot \varepsilon_i \cdot \alpha_i \quad \text{pour} \quad i = 1, 2 \tag{3.4}$$

Supposons $\oplus, \ominus, \otimes, \oslash$ les opérations arithmétiques de base sur l'ordinateur (respectivement $+, -, \times, \div$). Les erreurs dues aux opérations arithmétiques entre $X1$ et $X2$ sont présentées ci-dessous. Pour chaque opération, 2^{E_3} et ε_3 représentent respectivement l'exposant et le signe du résultat calculé. α_3 est l'erreur commise sur le résultat.

$$X_1 \oplus X_2 = x_1 + x_2 - 2^{E_1-p}\varepsilon_1\alpha_1 - 2^{E_2-p}\varepsilon_2\alpha_2 - 2^{E_3-p}\varepsilon_3\alpha_3 \tag{3.5}$$

$$X_1 \ominus X_2 = x_1 - x_2 - 2^{E_1-p}\varepsilon_1\alpha_1 + 2^{E_2-p}\varepsilon_2\alpha_2 - 2^{E_3-p}\varepsilon_3\alpha_3 \tag{3.6}$$

$$X_1 \otimes X_2 = x_1x_2 - 2^{E_1-p}\varepsilon_1\alpha_1x_2 - 2^{E_2-p}\varepsilon_2\alpha_2x_1 + 2^{E_1+E_2-2p}\varepsilon_1\varepsilon_2\alpha_1\alpha_2 - 2^{E_3-p}\varepsilon_3\alpha_3 \tag{3.7}$$

En négligeant le quatrième terme qui est du deuxième ordre en 2^{-p}, on obtient :

$$X_1 \otimes X_2 = x_1x_2 - 2^{E_1-p}\varepsilon_1\alpha_1x_2 - 2^{E_2-p}\varepsilon_2\alpha_2x_1 - 2^{E_3-p}\varepsilon_3\alpha_3 \tag{3.8}$$

Pour la division, en négligeant les termes d'ordre supérieur ou égal à 2^{-2p}, on obtient :

$$X_1 \oslash X_2 = \frac{x_1}{x_2} - 2^{E_1-p}\varepsilon_1\frac{\alpha_1}{x_2} - 2^{E_2-p}\varepsilon_2\alpha_2\frac{x_1}{x_2^2} - 2^{E_3-p}\varepsilon_3\alpha_3 \tag{3.9}$$

3.1.3 Le résultat informatique

L'objectif ici est de modéliser un résultat informatique comme la somme des quantités de même expression dont chacune représente une étape du calcul.

Theorème 3.1.2. Si R est le résultat informatique d'une procédure informatique P finie, ne faisant intervenir que les quatre opérations arithmétiques élémentaires, et si r représente le résultat obtenu en effectuant la même suite de calculs mais avec une précision infinie, alors en ne retenant que les termes du premier ordre en 2^{-p}.

$$R = r + \sum_{i=1}^{S_n} g_i(d) 2^{E_i-p}\varepsilon_i\alpha_i + O(2^{-2p}) \tag{3.10}$$

où

- g_i représente des constantes ne dépendant que des données et de l'algorithme ;
- E_i les exposants des résultats intermédiaires ;
- α_i la quantité perdue lors de la troncature ou de l'arrondi ;
- ε_i les signes des résultats intermédiaires ;
- n le nombre d'opérations pendant l'exécution.

Ce théorème est démontré dans [Chesneaux, 1988, p.8]. La sommation va de 1 à S_n car le nombre de termes dépend de n mais peut ne pas lui être égal comme montré dans la modélisation des opérations élémentaires (section 3.1.2). En effet, $S_n = 3$ pour une opération.

3.2 La méthode CESTAC

3.2.1 Approche stochastique des erreurs d'arrondi

Étudier la propagation des erreurs d'arrondi revient, en fait, à étudier le comportement des α_i . L'approche stochastique prend en compte le coté optimiste de la propagation d'erreurs. Son principe fondamental consiste à interpréter les α_i comme des quantités aléatoires.

La méthode méthode CESTAC (Controle et EStimation Stochastique des Arrondis de Calcul) ou méthode de la perturbation aléatoire est proposée en 1974 par La Porte et Vignes [Vignes et La Porte, 1974]. Cette méthode est basée sur le constat suivant : lors d'un calcul en troncature la valeur par défaut est systématiquement retenue alors que le choix de la valeur par excès pour représenter le résultat est tout aussi licite. La méthode propose donc de retenir à chaque étape du calcul avec une probabilité 0.5 l'une des deux valeurs. L'implémentation de la méthode des perturbations consiste à implémenter une arithmétique dite aléatoire que nous présentons dans la prochaine section.

Considérons maintenant la modélisation du résultat informatique présentée à l'équation 3.10. Appliquer la méthode de la perturbation aléatoire revient à remplacer l'erreur α_i par $\alpha_i - h_i$. On en déduit le théorème ci-dessous :

Theorème 3.2.1. Un résultat informatique R d'une procédure informatique P finie, ne faisant intervenir que les quatre opérations arithmétiques élémentaires et ayant subi les perturbations se modélise au premier ordre en 2^{-p} par :

$$R = r + \sum_{i=1}^{Sn} g_i(d) 2^{E_i - p} \varepsilon_i (\alpha_i - h_i) \tag{3.11}$$

où

- g_i représente des constantes ne dépendant que des données et de l'algorithme ;
- E_i les exposants des résultats intermédiaires ;
- α_i la quantité perdue lors de la troncature ou de l'arrondi ;
- h_i les perturbations aléatoires, ε_i les signes des résultats intermédiaires ;
- r le vrai résultat mathématique ;
- n le nombre d'opérations pendant l'exécution.

Une démonstration de ce théorème a été faite dans [Chesneaux, 1988, p.15].

3.2.2 Arithmétique aléatoire

Soit x un nombre réel, R^+ et R^- sont les deux flottants consécutifs qui encadrent x. On suppose qu'à chaque opération élémentaire, le résultat est arrondi vers $+\infty$ donc R^+ ou $-\infty$ donc

R^-. L'arithmétique aléatoire consiste à choisir aléatoirement R^+ ou R^- avec une probabilité de 0.5. C'est le mode d'arrondi aléatoire.

L'utilisation de ce nouveau mode d'arrondi permet d'exécuter le même programme plusieurs fois avec des propagations d'erreurs différentes (le mode d'arrondi étant choisi aléatoirement). Ainsi pour N exécutions, on devrait obtenir N résultats potentiellement différents à partir desquels on pourrait tirer des conclusions sur l'impact de la propagation des erreurs d'arrondi sur le programme en question. C'est le principe de base de la méthode CESTAC. La partie de la mantisse commune aux N résultats représente alors la partie non impactée par les erreurs et donc la partie correcte (valide) du résultat.

3.2.3 Estimation de la précision

L'exécution N fois de manière synchrone d'un programme de calcul permet d'obtenir N tirages de la variable aléatoire modélisée par l'équation 3.11 où les $(\alpha_i - h_i)$ sont des variables aléatoires indépendantes distribuées uniformément sur $[+1; -1]$. Deux conséquences majeures en découlent :

1. l'espérance mathématique de la variable R est le résultat mathématique exact r ;
2. la distribution de R est quasi-gaussienne.

Implémenter CESTAC revient donc à estimer la moyenne d'une variable aléatoire gaussienne à partir d'un échantillon. Le test de Student fournit un intervalle de confiance pour l'espérance d'une gaussienne à partir d'un échantillon de cette dernière sous une probabilité donnée.

$\forall \beta \in [0; 1], \exists \tau_\beta \in \mathbb{R}$ tel que :

$$P\left(|\overline{R} - r| \leq \frac{\tau_\beta \cdot \sigma}{\sqrt{N}}\right) = \beta \tag{3.12}$$

avec

$$\overline{R} = \frac{1}{N}\sum_{i=1}^{N} R_i \quad \text{et} \quad \sigma^2 = \frac{1}{N-1}\sum_{i=1}^{N}\left(R_i - \overline{R}\right)^2 \tag{3.13}$$

Le nombre de chiffres significatifs exacts de $\overline{R}$ (le nombre chiffres significatifs communs à $\overline{R}$ et r) sous une probabilité β, est majoré par :

$$C_{\overline{R}} = \log_{10}\left(\frac{\sqrt{N}|\overline{R}|}{\sigma\tau_\beta}\right) \tag{3.14}$$

L'implémentation de CESTAC dans un code revient donc à exécuter N fois ce dernier avec un mode d'arrondi aléatoire (vers $+\infty$ ou $-\infty$). Ainsi, on obtient à chaque fois N résultats R_i. On calcule la moyenne des R_i notée $\overline{R}$ qui sera le résultat de l'opération. Ensuite, on calcule $C_{\overline{R}}$ le nombre exact de chiffres significatifs de $\overline{R}$. Il a été montré qu'avec $N = 3$, $\tau_\beta = 4.303$ on obtient une implémentation optimum ($\beta = 0.95$) [Chesneaux, 1995, Vignes, 1993].

3.2.4 Validité de la méthode

La validité et l'efficacité de la méthode CESTAC a été montré dans [Chesneaux, 1988, chapitre 5,7]. Notons que la validité de la méthode repose sur deux hypothèses fondamentales :

1. les erreurs d'arrondi α_i sont indépendantes centrées et uniformément réparties ;

2. l'approximation au premier ordre en 2^{-p} dans la modélisation du résultat informatique R est valide.

Pour ce qui est de l'hypothèse 1, nous remarquerons que les erreurs d'arrondi, par l'utilisation de l'arithmétique aléatoire, sont devenues des variables aléatoires. Le test de Student donne un intervalle de confiance pour l'espérance mathématique de la variable R. De fait, il fournit un estimateur biaisé du résultat exact r. En pratique, cette variable aléatoire n'est pas rigoureusement centrée mais il a été montré que le biais introduit ne dépasse quelques σ (écart-type de R). Un tel biais entraîne une erreur inférieure à un chiffre décimal. C'est la raison pour laquelle, l'estimation du nombre de chiffres significatifs ($C_{\overline{R}}$) faite avec la méthode CESTAC est valide si on la considère exacte à un chiffre près.

En revanche, l'hypothèse 2 est fondamentale pour la validité de CESTAC. Quand cette approximation n'est pas vérifiée, R n'est plus centré sur r, ce qui met en défaut toute la méthode. On montre que la méthode peut être mise à défaut dans les cas de division par un résultat non significatif et de la multiplication de deux résultats non significatifs. Afin d'éviter cela, un contrôle dynamique est effectué pour les opérations arithmétiques. C'est l'auto-validation de la méthode CESTAC.

L'efficacité repose essentiellement sur le choix du nombre de passages N et la probabilité de l'intervalle de confiance β. Il a été montré que 3 passages sont largement suffisants. Pour un chiffre significatif supplémentaire dans $C_{\overline{R}}$, il faut avoir 100 fois plus d'échantillons. Augmenter N permettrait de mieux estimer l'erreur mais aurait des conséquences néfastes sur les performances des codes. Il est donc inutile de chercher à augmenter le nombre de passage.

En ce qui concerne la probabilité de l'intervalle de confiance, pour $\beta = 0.95$ et $N = 3$, la probabilité de surestimer le nombre de chiffres significatifs d'au moins un chiffre est de 0.00054 et la probabilité de sous-estimer le nombre de chiffres significatifs d'au moins un chiffre est de 0.29. En résumé, choisir $\beta = 0.95$ permet de garantir un nombre de chiffres significatifs exacts avec une forte probabilité (0.99946), même si on est pessimiste d'un chiffre.

3.2.5 Arithmétique stochastique discrète

L'implémentation synchrone (N résultats R_i sont calculés en même temps) de la méthode revient alors à remplacer chaque flottant par un triplet $X = (X_1, X_2, X_3)$ chaque opération arithmétique élémentaire $\Omega \in (+, -, \times, /)$ est définie par $X\Omega Y = (X_1 \omega Y_1, X_2 \omega Y_2, X_3 \omega Y_3)$ ou ω est l'opération en arithmétique flottante correspondante suivie d'un mode arrondi aléatoire.

Un important concept introduit par CESTAC est le zéro informatique ($@.0$) [Vignes, 1986].

Definition 3.1. Si $C_{\overline{R}} \leq 0$ ou si $\forall i, R_i = 0$ alors le résultat de l'opération est un zéro informatique ($@.0$).

A partir de ce nouveau concept de *zéro*, on redéfinit de nouvelles relations.

Definition 3.2. X est stochastiquement égale à Y si et seulement si $X - Y = @.0$.

Definition 3.3.

X est stochastiquement strictement plus grand que Y si et seulement si $\overline{X} > \overline{Y}$ et $X - Y \neq @.0$.

Definition 3.4.

X est stochastiquement strictement plus grand ou égal à Y si et seulement si $\overline{X} \geq \overline{Y}$ ou $X - Y = @.0$.

La méthode CESTAC couplée à ces nouvelles définitions constitue l'arithmétique stochastique discrète (ASD ou DSA pour Discrete Stochastic Arithmetic). Les éléments de la DSA sont les *nombres stochastiques* définis par CESTAC. Nous présentons ci-dessous quelques propriétés de l'arithmétique stochastique discrète. C'est cette arithmétique qui est implémentée dans l'outil de validation numérique CADNA que nous présentons dans la section 3.3.

- $x = 0 \Rightarrow X = @.0$;
- $X = Y \Rightarrow x = y$;
- $X > Y \Rightarrow x > y$;
- $x \geq y \Rightarrow X \geq Y$;
- La relation $>$ est transitive ;
- La relation $=$ est reflexive, symétrique mais non transitive ;
- La relation $\geq$ est reflexive, antisymétrique mais non transitive.

3.3 Le logiciel CADNA

CADNA est un logiciel de validation numérique dont l'objectif principal est de quantifier l'erreur de calcul due à l'arithmétique des flottants sur les résultats produits par un programme de simulation numérique. Il existe deux versions (C/C++ et Fortran 90) disponible en téléchargement libre à `www.lip6.fr/cadna`. Pendant l'exécution d'un programme, il permet :

- de mesurer l'effet de la propagation des erreurs d'arrondis ;
- de détecter les instabilités numériques ;
- d'estimer la précision en nombre de chiffres significatifs des opérations arithmétiques flottantes ;
- de contrôler l'enchaînement du programme.

CADNA définit de nouveaux types de données les *types stochastiques* :

- *float_st* en simple précision en C/C++, *single_st* en Fortran ;
- *double_st* pour la double précision.

Les types stochastiques sont composés de trois nombres flottants (*float* ou *double*) et d'un entier *accuracy* pour stocker le nombre de chiffres significatifs. CADNA redéfinit toutes les opérations arithmétiques de base $(+, -, \times, \div)$ ainsi que les relations d'ordre pour les types stochastiques $(\leq, <, \geq, >, ==)$. Les fonctions mathématiques de base $(\sin, \cos, \exp, \ldots)$ sont également redéfinies. L'affichage est surchargé avec la fonction *strp()* laquelle fonction permet d'afficher uniquement les chiffres significatifs d'un type stochastique. Pour implémenter le logiciel CADNA dans un code, il faut donc remplacer les flottants (*float* ou *double*) par les nouveaux types stochastiques. Il faut aussi appeler *cadna_init(numb_instability, cadna_instability, cancel_level, init_random)* et *cadna_end()* au début et à la fin du code et inclure le module CADNA (respectivement cadna.h en C/C++). Le premier paramètre $numb\_instability$ de la fonction *cadna_init()* est obligatoire et permet de signifier le nombre d'instabilités que l'on souhaite détecter. Les autres paramètres sont optionnels. Retenons simplement qu'ils permettent de signaler quel type d'instabilités CADNA doit détecter.

Initialiser le paramètre $numb\_instability$ à -1 permet d'activer l'auto-validation de CESTAC (section 3.2.4) et la détection de toutes les instabilités numériques. Par la suite, nous utiliserons deux modes de fonctionnement de CADNA :

i) mode $numb\_instability$ initialisé à -1 que nous appelons (abusivement) ***mode avec auto-validation*** ;

ii) mode $numb\_instability$ initialisé à 0 que nous appelons ***mode sans auto-validation*** ; en fait, on peut qualifier ce mode d'hybride ou de basique puisqu'il n'implémente pas com-

plètement la méthode CESTAC, certes les opérations arithmétiques sont effectuées 3 fois mais l'auto-validation de la multiplication et de la division de sont pas activés ; ce mode permet essentiellement de se faire une idée du surcoût dû à la surcharge des opérations mathématiques.

Nous présentons ci-dessous deux codes C++ de calcul de racine d'un polynôme du second degré : une version sans CADNA extrait de code 3.1 et une version avec CADNA extrait de code 3.2.

Source 3.1 – Exemple de code C++ sans CADNA : calcul des racines d'un polynôme du second degré

```
#include <stdio.h>
#include <math.h>

main()
{
  float a = 0.3;
  float b = -2.1;
  float c = 3.675;
  float d, x1,x2;

  //      CASE : A = 0
  if (a==0.)
    if (b==0.) {
      if (c==0.) printf("Every complex value is solution.\n");
      else printf("There is no solution.\n");
    }
    else {
      x1 = - c/b;
      printf("The equation is degenerated.\n");
      printf("There is one real solution %+.6e\n",x1);
    }
  else {
    //      CASE : A /= 0
    b = b/a;
    c = c/a;
    d = b*b - 4.0*c;
    printf("d = %+.6e\n",d);
    //   DISCRIMINANT = 0
    if (d==0.) {
      x1 = -b*0.5;
      printf("Discriminant is zero.\n");
      printf("The double solution is %+.6e\n",x1);
    }
    else {
      //      DISCRIMINANT > 0
      if (d>0.) {
          x1 = ( - b - sqrtf(d))*0.5;
          x2 = ( - b + sqrtf(d))*0.5;
    printf("There are two real solutions.\n");
    printf("x1 = %+.6e x2 = %+.6e\n",x1,x2);
      }
      else {
  //      DISCRIMINANT < 0
  x1 = - b*0.5;
  x2 = sqrtf(-d)*0.5;
  printf("There are two complex solutions.\n");
  printf("z1 = %+.6e  +  i * %+.6e\n",x1,x2);
```

```
  printf("z2 = %+.6e  +  i * %+.6e\n",x1, -x2);
      }
    }
  }
}
```

Source 3.2 – Exemple de code C++ utilisant CADNA : calcul des racines d'un polynôme du second degré

```
#include <stdio.h>
#include <math.h>
#include <cadna.h>

main()
{

  cadna_init(-1);

  float_st a = 0.3;
  float_st b = -2.1;
  float_st c = 3.675;
  float_st d, x1,x2;

  //      CASE: A = 0
  if (a==0)
    if (b==0.) {
      if (c==0.) printf("Every complex value is solution.\n");
      else printf("There is no solution.\n");
    }
    else {
      x1 = - c/b;
      printf("The equation is degenerated.\n");
      printf("There is one real solution %s\n",strp(x1));
    }
  else {
    //     CASE: A /= 0
    b = b/a;
    c = c/a;
    d = b*b - 4.0*c;
    printf("d = %s\n",strp(d));
    //   DISCRIMINANT = 0
    if (d==0.) {
      x1 = -b*0.5;
      printf("Discriminant is zero.\n");
      printf("The double solution is %s\n",strp(x1));
    }
    else {
      //      DISCRIMINANT > 0
      if (d>0.) {
          x1 = ( - b - sqrtf(d))*0.5;
          x2 = ( - b + sqrtf(d))*0.5;
    printf("There are two real solutions.\n");
    printf("x1 = %s x2 = %s\n",strp(x1),strp(x2));
      }
      else {
  //      DISCRIMINANT < 0
  x1 = - b*0.5;
  x2 = sqrtf(-d)*0.5;
  printf("There are two complex solutions.\n");
```

```
  printf("z1 = %s  +  i * %s\n",strp(x1),strp(x2));
  printf("z2 = %s  +  i * %s\n",strp(x1), strp(-x2));
      }
    }
  }

  cadna_end();
}
```

Ci-dessous l'exécution du code sans CADNA :

```
---------------------------------
|  Second order equation         |
|  without CADNA                 |
---------------------------------
d = -3.814697e-06
There are two complex solutions.
z1 = +3.500000e+00  +  i * +9.765625e-04
z2 = +3.500000e+00  +  i * -9.765625e-04
```

Ci-dessous l'exécution du code avec CADNA : contrairement à la première exécution, on obtient une racine double. L'arithmétique en virgule flottante standard ne peut pas détecter $d = 0$. Un mauvais branchement a été effectué et le résultat est faux. Les relations d'ordre stochastique permettent de faire un bon test, par conséquent, le bon branchement est effectué et le résultat exact est obtenu.

```
------------------------------------------------------------------
CADNA_C 1.1.2 software --- University P. et M. Curie --- LIP6
Self-validation detection: ON
Mathematical instabilities detection: ON
Branching instabilities detection: ON
Intrinsic instabilities detection: ON
Cancellation instabilities detection: ON
------------------------------------------------------------------
---------------------------------
|  Second order equation         |
|  with CADNA                    |
---------------------------------
d =  @.0
Discriminant is zero.
The double solution is  0.3499999E+001
------------------------------------------------------------------
CADNA_C 1.1.2 software --- University P. et M. Curie --- LIP6
There is  1 numerical instability
1 LOSS OF ACCURACY DUE TO CANCELLATION(S)
------------------------------------------------------------------
```

Cet exemple est tiré de la documentation officielle du logiciel CADNA disponibles sur www.lip6.fr/cadna. D'autres exemples y sont présentés et ils montrent clairement l'apport de l'arithmétique stochastique discrète pour les programmes numériques.

3.4 Les récents développements

Comme signalé en introduction de ce chapitre, la version initiale de la bibliothèque CADNA était développée en ADA et était alors payante. Depuis plusieurs travaux ont été réalisés autour de la méthode CESTAC et de la bibliothèque CADNA afin d'améliorer son implémentation, d'étendre ses fonctionnalités et de l'adapter aux différents paradigmes de programmation. Deux nouvelles versions de la bibliothèque ont alors été développées : l'une en Fortran 90 [Jézéquel *et al.*, 2010] et l'autre en C/C++ [Lamotte *et al.*, 2010]. Ces deux versions sont celles actuellement disponibles sur le site de CADNA.

Le logiciel est maintenu par l'équipe CADNA du LIP6 et des améliorations y sont régulièrement portées. Outre l'implémentation dans de nouveaux langages, des travaux sont également menés afin de rendre la bibliothèque compatible à toutes les architectures matérielles disponibles et aux codes parallèles. Une première extension avait été faite en Fortran 77 pour les routines de communication PVM [Jézéquel, 2005]. Le chapitre 4 et l'annexe B de ce document traitent de l'implémentation de CADNA dans les standards de communication MPI et BLACS. D'autres travaux sont également effectués pour profiter de la mémoire distribuée et de l'utilisation des processus linux [Jézéquel *et al.*, 2013b]. La principale difficulté pour cette implémentation réside dans la gestion des modes d'arrondi par les différents threads et la mise à jour des instabilités détectées. Ajoutons à cette dernière version, une implémentation CUDA [43] pour les GPU (Graphics Processing Unit) Nvidia [Jézéquel et Lamotte, 2010, Lamotte *et al.*, 2012, Jézéquel *et al.*, 2013a]. Signalons également qu'une nouvelle version de CADNA est en cours de développement par J-L. Lamotte. Cette nouvelle version permettra l'utilisation des instructions SIMD (SSE ou AVX). Enfin, notons que deux travaux ont été effectués pour combiner l'arithmétique stochastique discrète à d'autres types d'arithmétiques. Ces deux travaux ont donné naissance à la bibliothèque SAM qui implémente l'arithmétique stochastique en précision arbitraire [Graillat *et al.*, 2011] à l'aide de la bibliothèque MPFR et la bibliothèque "Fixed CADNA" qui combine la DSA et l'arithmétique à virgule fixe [Didier *et al.*, 2004].

L'outil CADNA a contribué à l'étude de la propagation des erreurs d'arrondi dans plusieurs codes. Un large aperçu des résultats de ces diverses études est présenté dans [Jézéquel, 2005, Lamotte, 2004, Chesneaux, 1995]. Le lecteur intéressé pourra également consulter les listes de publications disponibles sur les pages :

- `http://www-pequan.lip6.fr/~jezequel/publi_jezequel.html` ;
- `http://www-pequan.lip6.fr/~lamotte/publications/index.php` ;
- `http://www-pequan.lip6.fr/~jmc/publi_chesneaux.html`.

3.5 Conclusion

Ce chapitre a été consacré à la présentation de la méthode CESTAC et de son implémentation CADNA. Cette bibliothèque est un outil efficace pour validation numérique des codes de calculs. Dans le chapitre 2, nous avons montré que cet outil répondait aux différents critères de la validation numérique des codes industriels (coût, outil non intrusif, facile d'utilisation). Toutefois, l'outil ne peut être implémenté directement dans les principaux codes industriels, ces derniers faisant généralement appel à des bibliothèques extérieures. A titre d'exemple, la version parallèle code Telemac-2D est basée sur l'utilisation des routines MPI pour les échanges

43. CUDA (Compute Unified Device Architecture) est une technologie de GPGPU (General-Purpose Computing on Graphics Processing Units). Cette technologie a été développée par Nvidia pour programmer leurs GPU en C/C++

de données entre processus. D'autres outils de calcul scientifique (BLAS, LAPACK, etc) sont également très souvent mis à contribution. C'est dans cette optique que les travaux présentés dans les prochains chapitres (4 et 5) ont été réalisés. A l'image des récents travaux d'amélioration et d'adaptation de CADNA aux nouvelles architectures matérielles (GPU, instructions SIMD), les travaux présentés dans les prochains chapitres répondent d'une part aux besoins des utilisateurs de l'outil notamment les industriels comme EDF et contribuent d'autre part à l'établissement d'une chaîne de validation numérique complète basée sur l'arithmétique stochastique discrète.

Chapitre 4

Les types stochastiques et les bibliothèques de communication

Sommaire

Un de nos objectifs prioritaires dans le cadre de cette thèse est de proposer des implémentations d'extensions pour l'outil CADNA afin de faciliter son utilisation dans les grands codes parallèles de simulation. Bien souvent, ces derniers sont un ensemble de codes séquentiels qui communiquent par une bibliothèque d'envoi de messages (ou bibliothèques *à passage de messages*). Dans ce chapitre, nous nous concentrons sur le standard de communication : MPI. Ce travail a été présenté à SMAI2011 [Montan *et al.*, 2011].

Plan du chapitre : Ce chapitre est entièrement consacré à CADNA_MPI : une implémentation de CADNA pour le standard MPI. Après une brève présentation du standard MPI (section 4.1), les sections 4.2, 4.3 et 4.4 présentent les fonctionnalités, le développement et le mode d'utilisation de l'extension. Des résultats de tests réalisés sur CADNA_MPI sont exposés en section 4.5.

4.1 Le standard MPI

CADNA_MPI est une extension de la bibliothèque CADNA ayant pour objectif d'augmenter le périmètre d'utilisation de CADNA dans les codes parallèles en proposant une extension des types définis dans MPI [Message Passing Interface Forum, 2012] pour les types stochastiques. Nous commençons en présentant le standard de communication MPI.

> *« MPI (Message-Passing Interface) is a message-passing library interface specification. All parts of this definition are significant. MPI addresses primarily the message-passing parallel programming model, in which data is moved from the address space of one process to that of another process through cooperative operations on each process. Extensions to the "classical" message-passing model are provided in collective operations, remote-memory access operations, dynamic process creation, and parallel I/O. MPI is a specification, not an implementation ; there are multiple implementations of MPI.This specification is for a library interface ; MPI is not a language, and all MPI operations are expressed as functions, subroutines, or methods, according to the appropriate language bindings which, for C and Fortran, are part of the MPI standard. »*
>
> Message Passing Interface Forum, Septembre 2012.

Le standard MPI définit donc une norme pour les échanges de messages (données) entre deux processus. La norme est définie et maintenue par une communauté de constructeurs et utilisateurs de machines parallèles. L'objectif de cette communauté était de définir une interface unique efficace, facile d'utilisation et flexible pour les échanges de messages afin de favoriser la portabilité des codes. Comme toute norme, elle évolue au cours du temps. La première version préliminaire de la norme MPI-1 a été proposée par Dongarra et al. en Novembre 1992 mais la première norme MPI-1.1 n'a été approuvée qu'en 1994 [Message Passing Interface Forum, 1994]. Cette version a conduit à un premier document officiel en juin 1995 [Message Passing Interface Forum, 1995]. La norme a ensuite évolué progressivement avec l'ajout de nouvelles fonctionnalités (gestion dynamique de processus, copies mémoire, entrés-sorties parallèles, etc) et le développement d'extensions en C/C++. MPI-3.0 a été approuvée en 2012. C'est une mise à jour majeure de la norme MPI qui comprend une extension des opérations collectives en incluant des versions non bloquantes et une nouvelle extension pour le Fortran 2008. Plusieurs

fonctionnalités obsolètes du C++ (comme le type MPI_UB utilisé dans la construction des types dérivés) ont été enlevées.

Le standard MPI a été également écrit dans le but d'obtenir de bonnes performances aussi bien sur des machines massivement parallèles à mémoire partagée que sur des clusters d'ordinateurs hétérogènes à mémoire distribuée. Ainsi, MPI possède l'avantage par rapport aux plus vieilles bibliothèques de passage de messages (comme PVM [44]) d'être grandement portable (car la bibliothèque MPI a été implantée sur presque toutes les architectures de mémoires) et rapide (car chaque implantation a été optimisée pour le matériel sur lequel il s'exécute). Les principales implémentations de MPI sont en C/C++ et FORTRAN, mais il existe aussi des implémentations en Python, OCaml et Java. Les versions open source sont MPICH [45], aujourd'hui devenue MPICH 2 en supportant le standard MPI-2 et OpenMPI [46]. Outre ces versions, il existe des implémentations optimisées par certains constructeurs (Intel, IBM, Microsoft). Nous avons travaillé avec OpenMPI [Gabriel *et al.*, 2004] qui tend à devenir la version la plus stable et la plus utilisée.

4.2 Les fonctionnalités de l'extension

A partir du mode de fonctionnement de l'outil CADNA, nous avons évalué les besoins pour rendre compatible son utilisation dans un code faisant appel aux routines MPI [Montan, 2010]. La nouvelle bibliothèque doit suivre le même mode de fonctionnement que CADNA. Il est donc nécessaire d'implémenter :

1. des routines pour échanger des données de types stochastiques (point à point, collectives) ;
2. des routines pour effectuer des opérations de réduction sur les données de types stochastiques ;
3. une fonction d'initialisation et une autre pour signaler la fin du programme et afficher le diagnostic des instabilités numériques détectées.

Échanger de données stochastiques entre processus : Pour l'échange de données de type *single_st* ou *double_st* entre deux processus à l'aide des routines MPI, il est nécessaire de créer des types dérivés. Notons que le type dérivé MPI est l'équivalent (au sens MPI) d'une structure de donnée C ou Fortran. Dès lors qu'ils sont créés, les types dérivés fonctionnent de manière identique à un type standard MPI. On peut, de ce fait, utiliser toutes les routines de communication (point à point , collectives) avec les types dérivés. Rappelons que les communications point-à-point permettent à deux processus à l'intérieur d'un même communicateur d'échanger une donnée (scalaire,tableau). Les fonctions correspondantes sont MPI_Send, MPI_Recv et MPI_Sendrecv. Les communications collectives impliquent tous les processus d'un communicateur (un ensemble de processus pouvant communiquer ensemble). Par exemple, il est possible d'envoyer une même donnée à tous les processus (MPI_Bcast), de découper un tableau entre tous les processus (MPI_Scatter).

Deux types dérivés ont alors été créés :

44. PVM (Parallel Virtual Machine) est une bibliothèque de communication (langages C et Fortran) pour machines parallèles et réseau d'ordinateurs (locaux ou distants, éventuellement hétérogènes). Il permet à un réseau d'ordinateurs d'apparaître comme un seul ordinateur.
45. www.mcs.anl.gov/mpi
46. www.open-mpi.org/

– MPI_SINGLE_ST pour le type CADNA en simple précision ;
– MPI_DOUBLE_ST pour le type CADNA en double précision.

Effectuer des réductions sur les données stochastiques : Les routines de réduction permettent de travailler simultanément sur des données localisées sur différents processeurs. On peut par exemple faire la somme de plusieurs tableaux qui se trouvent sur les différents processus. Plusieurs opérateurs de réductions existent. Les opérateurs MPI_MAX, MPI_SUM, MPI_PROD, MPI_MIN sont ceux qui peuvent s'appliquer aux types CADNA mais ils ont été écrits pour les types standards. Pour les faire fonctionner avec les nouveaux types CADNA_MPI, nous avons créé de nouveaux opérateurs.

Les opérateurs suivants ont créé dans CADNA_MPI :
– en simple précision : MPI_CADNA_SUM_SP, MPI_CADNA_PROD_SP, MPI_CADNA_MAX_SP, MPI_CADNA_MIN_SP ;
– en double précision : MPI_CADNA_SUM_DP, MPI_CADNA_SUM_DP, MPI_CADNA_MAX_DP, MPI_CADNA_MIN_DP.

Nous avons aussi travaillé sur les opérateurs MPI_MAXLOC et MPI_MINLOC qui permettent de calculer les maximums (respectivement minimums) tout en sauvegardant leurs emplacements respectifs (rang processus). Deux structures de données spéciales sont nécessaires pour ces opérateurs : *single_st_int*, *double_st_int*. Les types dérivés associés sont MPI_SINGLE_ST_INT et MPI_DOUBLE_ST_INT et les opérateurs de réductions sont MPI_CADNA_MAXLOC_SP, MPI_CADNA_MAXLOC_DP, MPI_CADNA_MINLOC_SP, MPI_CADNA_MINLOC_DP.

Initialisation et terminaison d'un programme avec CADNA_MPI : Nous avons créé deux fonctions *cadna_mpi_init (int rang, int numb_instability)* et *cadna_mpi_end (int rang)* pour l'initialisation et la terminaison de CADNA. Ces fonctions font appels respectivement à *cadna_init(numb_instability,...)* et *cadna_end()*. Nous revenons plus en détail sur le contenu de ces fonctions dans la prochaine section.

4.3 Développement

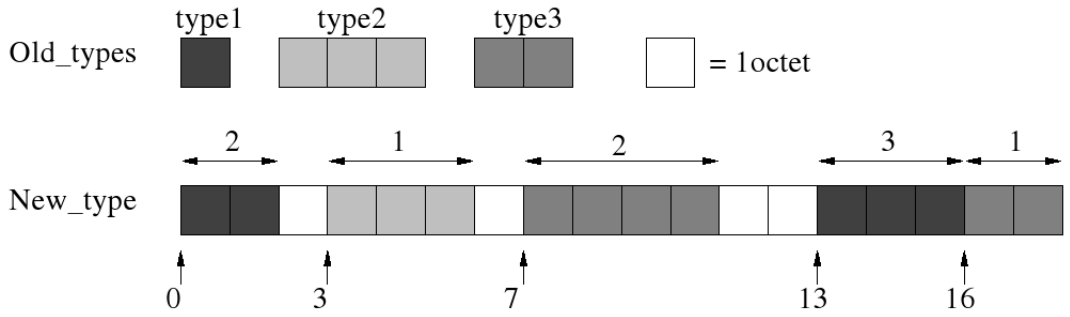
4.3.1 Un module Fortran90

Un module Fortran a été développé pour CADNA_MPI. Dans ce module on retrouve deux principales fonctions : *cadna_mpi_init(int rang, int numb_instability)* et *cadna_mpi_end(int rang)* et des fonctions nécessaires pour la création des opérateurs de réduction.

cadna_mpi_init (int rang, int numb_instability) : elle doit être obligatoirement appelée à chaque début de programme juste après l'appel des fonctions d'initialisation du standard MPI. Elle prend en paramètre le numéro du processus [47] courant et d'autres paramètres (dont certains sont optionnels) qui permettent d'initialiser CADNA en appelant la fonction *cadna_init(numb_instability,...)*. C'est également dans cette fonction que nous créons les deux types dérivés et tous les opérateurs de réduction. Les paramètres optionnels sont *cadna_instability*, *cancel_level* et *init_random* qui permettent de spécifier les types d'instabilités que l'on souhaite détecter (voir chapitre 3).

47. Le paramètre rang du processus n'est pas obligatoire. Initialement, il a été mis en place pour initialiser également MPI mais afin de rendre l'extension plus facile d'utilisation, l'initialisation de CADNA et celle de MPI ont été séparées.

Figure 4.1 – Construction d'une structure de données hétérogène avec MPI_TYPE_CREATE_STRUCT



nb_blocs=5 tab_types=(type1, type2, type3, type1, type3)

longueur_bloc=(2, 1, 2, 3, 1) deplacement=(0, 3, 7, 13, 16), en nombre d'octets

cadna_mpi_end (int rang) : cette fonction prend en entrée le numéro du processus courant ; elle aussi doit être obligatoirement appelée à la fin d'un programme utilisant CADNA_MPI. Elle fait appel à *cadna_end()* qui affiche les irrégularités détectées par CADNA pendant l'exécution du code. Cette fonction affiche à l'écran le diagnostic de la partie du code exécutée sur le processus courant.

Pour créer un type dérivé, on utilise deux fonctions MPI. Étant donné que les structures *single_st* (respectivement *double_st*) sont constituées de deux types de données différentes *real* et *integer* (respectivement *double précision* et *integer*), nos types dérivés seront hétérogènes. Nous allons utiliser d'abord la fonction **MPI_TYPE_CREATE_STRUCT**(*nb_elements, longueur_bloc, deplacement, tableau_types,new_type,code*) (voir figure 4.1).

- **nb_elements** (< *in* >) est le nombre de bloc de la structure, on appelle « bloc » un ensemble d'éléments du même type.
- **longueur_bloc** (< *in* >) est un tableau de dimension *nb_elements* contenant le nombre d'éléments de chaque bloc de la structure.
- **deplacement** (< *in* >) est un tableau de dimension *nb_elements* contenant l'adresse en octets de chaque début de bloc par rapport à celle du premier bloc. Il doit être déclaré comme : *integer(kind=MPI_ADDRESS_KIND), dimension(nb_blocs) : : deplacement*. On détermine cette adresse via la fonction **MPI_GET_ADDRESS**(*element_struct, adress_element, code*).
- **tableau_types** (< *in* >) est un tableau de dimension *nb_elements* contenant le type de chaque bloc de la structure.
- **new_type** (< *out* >) est le nouveau type de données créé.
- **code** est le code erreur en Fortran.

La création d'un type dérivé est suivie de sa validation avec la fonction **MPI_TYPE_COMMIT**(*new_type,code*). Cette validation permet l'utilisation du nouveau type par tous les processus. Pour libérer la mémoire occupée par les types créés, on fait appel à la fonction **MPI_TYPE_FREE***(new_type,code)* à la fin des programmes. La création et le commit du type dérivé se font dans la fonction *cadna_mpi_init(rang)* (voir extrait de code 4.1) et la libération dans la fonction *cadna_mpi_end(rang)*.

Source 4.1 – création de MPI_SINGLE_ST dans cadna_mpi_init(rang). Ici on utilise MPI_TYPE_EXTENT pour les adresses et on suppose que le premier bloc se trouve à l'adresse 0.

```
module cadna_mpi
  !      =============
  use CADNA
  implicit none
  INCLUDE 'mpif.h'

  integer , parameter :: nb_block=2
  integer MPI_SINGLE_ST,MPI_DOUBLE_ST
  .....
  !

  subroutine cadna_mpi_init(rank,numb_instability, cadna_instability, ↩
      cancel_level, init_random)
    implicit none
    integer, optional :: cadna_instability
    integer, optional :: cancel_level
    integer, optional :: init_random
    integer :: numb_instability
    integer,DIMENSION (0:nb_block-1) ::block_lengths,typelist,displacements
    integer  ierr,extent,rank

    ! first init CADNA
    print*,'  '
    print*,'CADNA INIT FOR PROC ',rank
    call cadna_init(numb_instability,cadna_instability, cancel_level, ↩
        init_random)

    !**** cadna_mpi_type/MPI_SINGLE_ST
    !  NAME
    !    MPI_SINGLE_ST
    !  COPYRIGHT
    !    2010 by EDF R&D
    !  DESCRIPTION
    !    The stochastic type in simple precision for MPI
    !******

    ! The first block of 3 real x,y,z
    displacements(0) = 0
    typelist(0) = MPI_REAL
    block_lengths(0) = 3

    ! second block of 1 integer acc
    call MPI_TYPE_EXTENT(MPI_REAL,extent,ierr)
    displacements(1) = 3 * extent
    typelist(1) = MPI_INTEGER
    block_lengths(1) = 1

    ! now define structured type and commit it
    call MPI_TYPE_CREATE_STRUCT(nb_block,block_lengths,displacements,\
        typelist,MPI_SINGLE_ST)
    call MPI_TYPE_COMMIT(MPI_SINGLE_ST,ierr)
    ..........
    ..........
  end subroutine cadna_mpi_init
```

La création d'un opérateur de réduction suit le même principe que la création des types dérivés. On utilise la fonction MPI_OP_CREATE*(function, commute, operateur, code)* pour la création (voir extrait de code 4.2).

- **function***(<in>)* est une fonction à écrire. Elle définit clairement l'opération à faire. Elle est définie comme toute les fonctions en Fortran : *function user_function(in, inout,len,type)* *in* tableau d'entrée, *inout* tableau d'entrée et aussi tableau de sortie, *len* la longueur des tableaux et *type* le type de donnée concernée. La variable *inout[i] = opérateur(inout[i],in[i])* i variant de 0 à *len-1* ; l'opérateur peut être l'addition, la multiplication, le min ou le max.
- **commute***(<in>)* est un booléen qui permet de signifier si l'opération à faire est associative. Si commute=true alors l'opération est commutative et associative ; et si commute=false alors la réduction commencera à partir des données du processus 0. L'arithmétique flottante n'étant pas associative, nous avons choisi *commute=false* dans nos codes sources.
- **operateur***(<out>)* est le nouvel opérateur de réduction créé.
- **code** est le code erreur en Fortran.

Source 4.2 – Création d'un opérateur de réduction : la fonction MPI_OP_CREATE permet de créer l'opérateur MPI_CADNA_SUM_SP avec la fonction SP_SUM.

```
module cadna_mpi
  .....
  !

  subroutine cadna_mpi_init(rank,numb_instability, cadna_instability, ↩
      cancel_level, init_random)
    ..........
    ..........
    !now define MPI reductions operators for CADNA
    ! SUM
    commute = .false.
    call MPI_OP_CREATE(SP_SUM,commute,MPI_CADNA_SUM_SP,ierr)
    ..........
    ..........

  end subroutine cadna_mpi_init
  !
  !****** cadna_mpi/MPI_CADNA REDUCTION OPERATORS
  !  NAME
  !    SP_SUM
  !
  !  FUNCTION
  !    Subroutine to ADD 2  MPI_SINGLE_ST
  !
  !*****************************************
  !
  SUBROUTINE SP_SUM(IN,INOUT,LEN,TYPE)

    INTEGER LEN,TYPE,I
    type(SINGLE_ST)IN(LEN),INOUT(LEN)
    DO I = 1,LEN
       INOUT(I) = IN(I) + INOUT(I)
    ENDDO
  END SUBROUTINE SP_SUM
```

Pour les opérateurs, il n'y a pas de fonction dédiée au commit. On utilise la fonction MPI_OP_FREE(Op) pour la libération mémoire.

4.3.2 CADNA_MPI pour C/C++

La bibliothèque CADNA_C a été écrite en C mais en utilisant plusieurs notions de base du C++. Les types stochastiques *float_st* et *double_st* y sont implémentés en tant que classe. Pour un souci de conformité et de compatibilité, on retrouve dans la version C/C++ de CADNA_MPI les mêmes fonctions et types que dans la version Fortran. Les types et les opérateurs ont des noms identiques et les mêmes rôles. On retrouve donc dans la version C++ de CADNA_MPI :

- les deux principales fonctions *cadna_mpi_init(rang,numb_instability)* et *cadna_end(rang)* ;
- les types dérivées MPI_SINGLE_ST, MPI_DOUBLE_ST, MPI_SINGLE_ST_INT et MPI_DOUBLE_ST_INT ;
- les opérateurs de réduction
  - en simple précision : MPI_CADNA_SUM_SP, MPI_CADNA_PROD_SP, MPI_CADNA_MAX_SP, MPI_CADNA_MIN_SP, MPI_CADNA_MAXLOC_SP, MPI_CADNA_MINLOC_SP ;
  - en double précision : MPI_CADNA_SUM_DP, MPI_CADNA_SUM_DP, MPI_CADNA_MAX_DP, MPI_CADNA_MIN_DP, MPI_CADNA_MAXLOC_DP, MPI_CADNA_MINLOC_DP.

4.4 Mode d'utilisation

Pour faciliter l'utilisation de CADNA_MPI, une librairie statique a été créée pour chaque version (C/C++ et Fortran 90). La librairie statique CADNA est incluse dans celle de CADNA_MPI. Pour compiler un code CADNA_MPI, il faudra juste faire appel à la librairie correspondante (*libcadnampiF.a* pour le Fortran 90 et *libcadnampiC.a* pour le C/C++). Dans un code C/C++, il faut inclure le fichier *cadna_mpi.h* qui se trouve dans le répertoire *CADNA/CADNA_C/inc* et compiler le fichier avec la librairie *libcadnampiC.a* qui se trouve dans le répertoire *CADNA/CADNA_C/lib*. Dans un code Fortran 90, il faut inclure le module CADNA_MPI qui se trouve dans le répertoire *CADNA/CADNA_C/inc* et compiler le fichier avec la librairie *libcadnampiF.a* qui se trouve dans *CADNA/CADNA_C/lib*. Il faut obligatoirement faire appel aux fonctions *cadna_mpi_init(rang,numb_instability)* au début du programme et *cadna_mpi_end(rang)* à la fin. La routine *MPI_Finalize()* vient toujours après la fonction *cadna_mpi_end(rang)*.

En résumé, un code parallèle utilisant CADNA devra obligatoirement respecter l'ordre suivant :

1. Insertion CADNA_MPI.h (module CADNA_MPI en Fortran) ;
2. Initialisation de MPI (*MPI_Init()*) ;
3. Initialisation de CADNA_MPI (*cadna_mpi_init()*) ;
4. Corps du programme ;
5. Terminaison de CADNA_MPI (*cadna_mpi_end(rang)*).
6. Terminaison MPI (*MPI_Finalize()*.

L'extrait de code 4.3 présente un exemple de programme C/C++ parallèle utilisant CADNA_MPI. On y retrouve l'ajout de *cadna_mpi.h*, l'appel de la fonction *cadna_mpi_init(rang,numb_instability)* et *cadna_mpi_end(rang)*. Dans ce programme, nous envoyons une donnée du type *float_st* du processus 0 vers le processus 1 puis nous effectuons une réduction des tableaux qui sont stockés en local *tab_local* de chaque processus dans un autre tableau *tab_reduction*. Ensuite, on rassemble tous les tableaux locaux dans un plus grand

tableau *tab_gather*. L'extrait de code 4.5 montre l'exécution de cet exemple. On retrouve 3 parties dans l'affichage à savoir l'initialisation de CADNA_MPI sur chaque processus, l'exécution du code, et l'affichage du diagnostic de chaque processus. On remarque qu'aucune instabilité n'a été détectée, ce qui est normal puisqu'il n'y a eu aucun calcul. L'extrait de code 4.4 est un exemple de Makefile.

Source 4.3 – Exemple de code utilisant CADNA_MPI

```
#include <cadna_mpi.h>
int main(int argc, char *argv[])
{
  int n_proc, rang, taille =2;
  int tag = 1000 ;
  MPI_Status status ;
  MPI_Init (&argc,&argv ) ;
  MPI_Comm_size(MPI_COMM_WORLD,&n_proc);
  MPI_Comm_rank(MPI_COMM_WORLD,&rang);

  cadna_mpi_init(rang,-1);

  float_st x = 77617;
  float_st rsp;
  double_st rdp,res, y = 33096.;

  if(rang==0){
   MPI_Send(&x,1,MPI_SINGLE_ST,1, tag , MPI_COMM_WORLD);
   printf("send sp =%s\n",strp(x));
  }
  if(rang==1){
   MPI_Recv(&rsp,1,MPI_SINGLE_ST,0 , tag , MPI_COMM_WORLD , &status);
   printf("recv sp=%s\n",strp(rsp));
  }

 float_st tab_local[taille];
 float_st tab_reduction[taille];
 float_st tab_gather[taille*n_proc];
 for(int i=0;  i<taille;i++)tab_local[i]= 100 + rang + i+1;

 cout<<"tab_local du proc "<<rang<<endl;
 for(int i=0;i<taille;i++)cout<<strp(tab_local[i])<<" | ";
 cout<<endl;

 MPI_Allreduce(tab_local,tab_recu,taille,MPI_SINGLE_ST,\
  MPI_CADNA_SUM_SP,MPI_COMM_WORLD);
 MPI_Allgather(tab_reduction,taille,MPI_SINGLE_ST,tab_gather,\
  taille*n_proc,MPI_SINGLE_ST,MPI_COMM_WORLD);

 cout<<"tab_reduction du proc "<<rang<<endl;
 for(int i=0;i<taille;i++)  cout<<strp(tab_reduction[i])<<" | ";
 cout<<endl;

 cout<<"tab_gather du proc "<<rang<<endl;
 for(int j=0;j<n_proc;j++)
  for(int i=0;i<taille;i++) cout<<strp(tab_gather[i])<<" | ";
 cout<<endl;

  cadna_mpi_end(rang);
  MPI_Finalize();
```

```
51   return 0;
52 }
```

Source 4.4 – Exemple de Makefile

```
MPICXX  =  mpic++

#CADNA_MPI
CADNAMPIDIR =/local00/home/S51270/Logiciels/CADNA_MPI
CADNAMPIFLAGS=-I$(CADNAMPIDIR)/CADNA_C/inc
CADNALIB=$(CADNAMPIDIR)/CADNA_C/lib/libcadnampiC.a
CADNALDFLAGS=-L$(CADNAMPIDIR)/CADNA_C/lib -lcadnampiC


LDFLAGS=-lm

EXEC = hello

all : $(EXEC)

hello.o : hello.cc
  $(MPICXX) -c $< $(CADNAMPIFLAGS)

hello : hello.o
  $(MPICXX) $< -o $@ $(CADNALDFLAGS) $(LDFLAGS)

clean :
  rm -fr *.o $(EXEC)
```

Source 4.5 – Sortie pour une exécution avec 2 processus

```
S51270@clau5ej5:/local00/home/S51270/CadnaMPI/EXEMPLE$ mpirun -np 2 hello

CADNA MPI INIT FOR PROC : 0

CADNA MPI INIT FOR PROC : 1
------------------------------------------------------------------
CADNA_C 1.1.2 software --- University P. et M. Curie --- LIP6
Self-validation detection: ON
Mathematical instabilities detection: ON
Branching instabilities detection: ON
Intrinsic instabilities detection: ON
Cancellation instabilities detection: ON
------------------------------------------------------------------
-----------------------------------------
|  TESTING CADNA MPI FOR C Language      |
|                                        |
-----------------------------------------
send sp = 0.7761700E+005
tab_local du proc 0
 0.1010000E+003 |  0.1019999E+003 |
tab_reduction du proc 0
 0.2049999E+003 |  0.2049999E+003 |
tab_gather du proc 0
 0.2049999E+003 |  0.2049999E+003 |  0.2049999E+003 |  0.2049999E+003 |

CADNA END FOR PROC 0
------------------------------------------------------------------
```

```
CADNA_C 1.1.2 software --- University P. et M. Curie --- LIP6
No instability detected
----------------------------------------------------------------
----------------------------------------------------------------
CADNA_C 1.1.2 software --- University P. et M. Curie --- LIP6
Self-validation detection: ON
Mathematical instabilities detection: ON
Branching instabilities detection: ON
Intrinsic instabilities detection: ON
Cancellation instabilities detection: ON
----------------------------------------------------------------
----------------------------------------
|  TESTING CADNA MPI FOR C Language      |
|                                        |
----------------------------------------
recv sp= 0.7761700E+005
tab_local du proc 1
 0.1019999E+003 |  0.1030000E+003 |
tab_reduction du proc 1
 0.2049999E+003 |  0.2049999E+003 |
tab_gather du proc 1
 0.2029999E+003 |  0.2049999E+003 |  0.2029999E+003 |  0.2049999E+003 |

CADNA END FOR PROC 1
----------------------------------------------------------------
CADNA_C 1.1.2 software --- University P. et M. Curie --- LIP6
No instability detected
----------------------------------------------------------------
S51270@clau5ej5:/local00/home/S51270/CadnaMPI/EXEMPLE$
```

Différents tests ont été effectués afin de tester la validité et la conformité de CADNA_MPI. Pour chaque routine de communication de MPI, les deux nouveaux types ont été testés. Les mêmes tests ont été effectués pour les opérateurs de réduction en simple précision comme en double précision. Ces tests ont été concluant. Dans la prochaine section, nous mettons à l'épreuve notre bibliothèque afin de déterminer les surcoûts éventuels qui découlent de son utilisation.

4.5 Tests et résultats

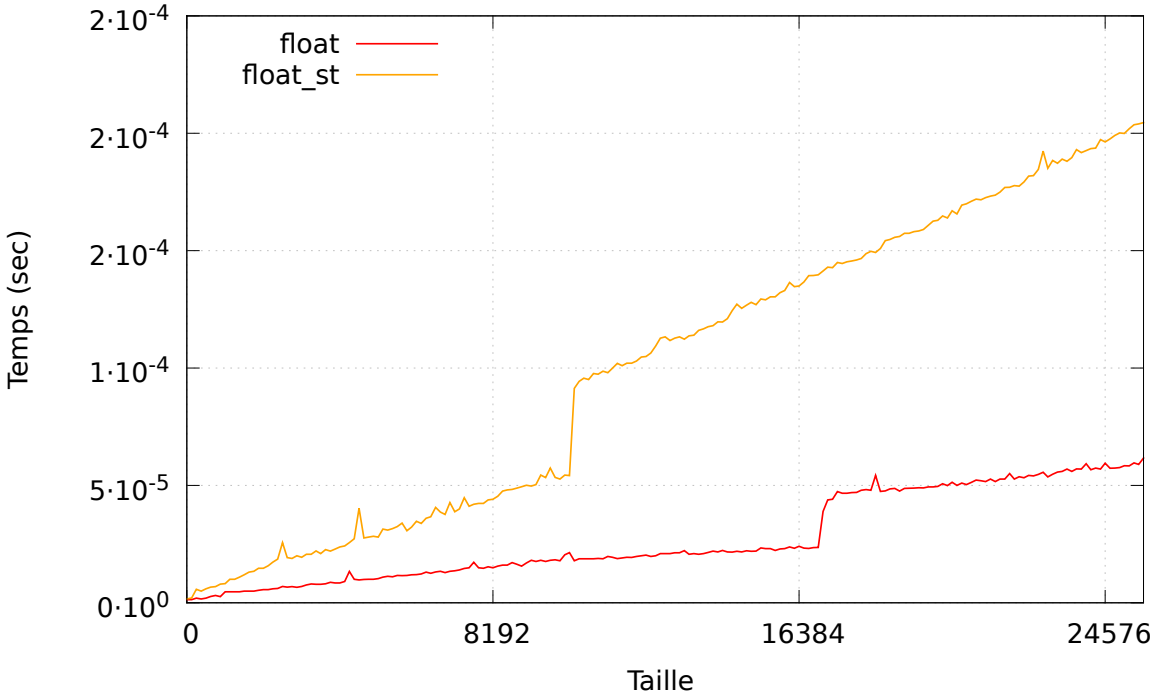
Les tests ont été principalement effectués sur un poste scientifique type EDF R&D : *HP Z600 workstation*. Cette machine, ainsi que le protocole de mesure de temps sont présentés en détail en annexe du manuscrit (annexe A, respectivement aux sections A.2 et A.1). Afin de confronter les résultats, une partie des expérimentations a aussi été réalisée sur la machine Ivanoe (présentée en section A.3). Toutes les expérimentations ont été effectuées avec la version 1.6.4 d'OpenMPI.

4.5.1 Temps de communication avec CADNA :

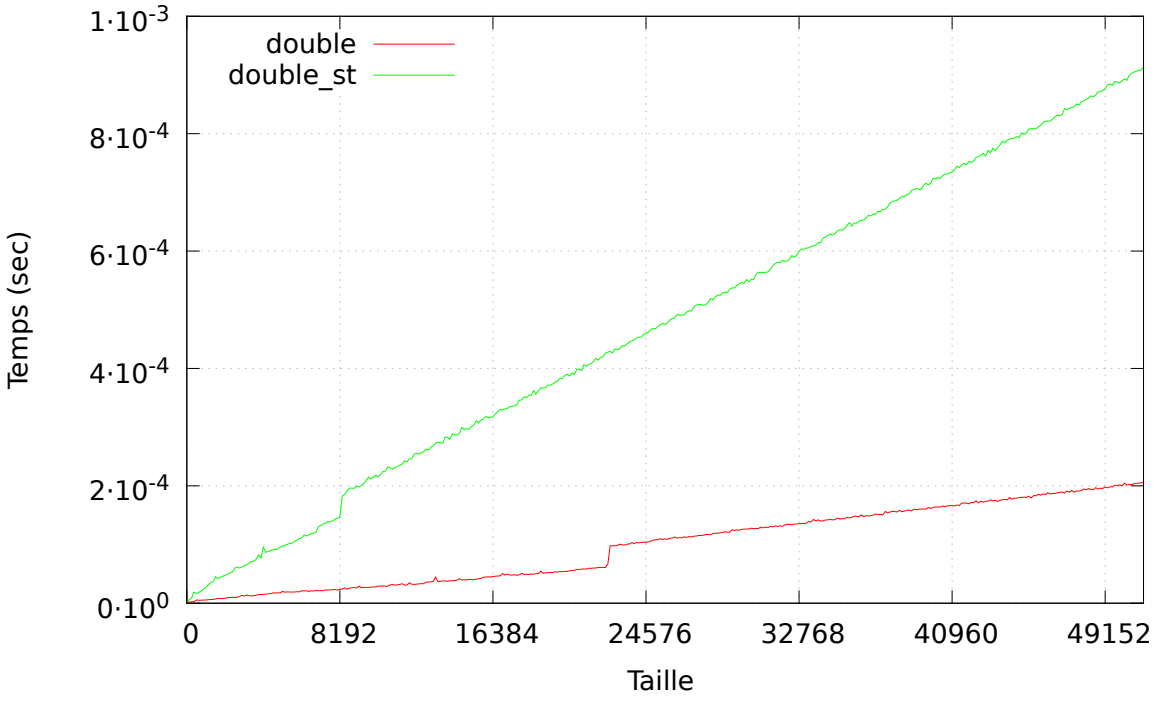
Dans un premier temps, nous voulons déterminer les temps de communication des types stochastiques. Nous avons réalisé un "ping-pong" pour estimer la durée des communications avec ces nouvelles structures. En fait, nous mesurons la durée d'un échange (envoi+réception) de tableaux de types stochastiques en fonction du nombre d'éléments envoyés. Nous avons comparé la durée des échanges d'un flottant standard (*double*, *float*) à celle des types stochastiques (*double_st*, *float_st*).

Figure 4.2 – Temps de communication sur la HP Z600

(a) simple precision



(b) double precision



Remarque 4.5.1. Un *float_st* est composé de 3 *float* et d'un entier. La taille d'un *float* est de 4 octets et celle d'un entier est de 4 octets aussi. Un *float_st* a donc une taille de $3 \times 4 + 4 = 16$ octets. L'envoi d'un *float* doit alors être environ 4 fois plus rapide que l'envoi d'un *float_st* (car taille *float_st* / taille *float* = 4).

Remarque 4.5.2. Un *double* a une taille de 8 octets, un *double_st* a donc une taille de 28 octets. L'envoi d'un *double* doit être alors environ 3.5 fois plus rapide que l'envoi d'un *double_st* (car taille *double_st* / taille *double* = 3.5).

Les mesures des deux premiers tests ont été relevées sur la HP Z600 et elles donnent les courbes des figures 4.2a et 4.2b. Nous observons dans le cas des *float* (simple précision, voir *figure* 4.2a) un facteur d'environ 4 entre les temps des *float* et ceux des *float_st*. Ce facteur est à peu près conforme avec nos hypothèses (remarque 4.5.1). En revanche, dans le cas de la double précision (voir *figure* 4.2b), on note un facteur d'environ 5 entre les temps des *double* et ceux des *double_st*.

L'incohérence des facteurs (théorique et mesuré) s'explique par le fait que les types stochastiques ne sont pas homogènes (les données ne sont pas contiguës en mémoire). Rappelons que le standard MPI offre deux solutions pour la création des structures :

- les types contigus : structures de données composées d'éléments d'un même type ;
- le types hétérogènes : structures de données pouvant contenir des types de données séparés en blocs composés d'éléments de même type.

Pour éviter ce surcoût dans les communications, une première solution est de rendre les types stochastiques homogènes et contigus. Cela implique de ne plus envoyer les entiers, qui permettent de stocker l'*accuracy*, mais de re-calculer l'*accuracy* après chaque réception. Cependant, cette solution peut s'avérer très coûteuse dès qu'il y aura un nombre important de communications, beaucoup de calculs supplémentaires seront alors nécessaires. Il importe donc de faire un compromis entre l'envoi de type contigu et le calcul de l'*accuracy*. Cela dit, quand on considère le cas du *float_st*, on remarque que le type dérivé n'est pas contigu mais qu'on obtient un rapport de 4 dans les communications. Remarquons que sa taille de 16 octets permet d'avoir des données alignées en mémoire. Une solution simple consiste alors à compléter le type *double_st* de façon à ce que sa taille soit de 32 octets pour ainsi favoriser l'alignement mémoire. Deux structures de données peuvent être alors utilisées :

1. une structure de quatre *double* : trois *double* pour le calcul et le dernier *double* pour l'*accuracy*)
2. une structure de trois *double* et deux entiers : trois *double* pour le calcul, un entier pour l'*accuracy*) et le dernier entier qui ne sert strictement à rien.

La solution 1 implique de repenser totalement la conception de CADNA et les opérations qui sont effectuées sur le champ *accuracy*. La solution 2 est alors privilégiée. C'est ce qui s'appelle utiliser un *padding*. Nous recommençons le premier test en comparant ici les *double* à un type dérivé de trois *double* et un entier (*dddi*) et un autre type dérivé de 3 *double* et 2 entiers (*dddii*). Le résultat de ce test est présenté dans la *figure* 4.3. Nous retrouvons un facteur 4 dans le cas où on utilise le *padding*. Une modification du type *double_st* a alors été faite. L'ajout de *padding* est complètement transparent pour les utilisateurs puisqu'il ne modifie pas l'arithmétique stochastique discrète.

Les résultats de la figure *figure* 4.4 sont ceux des mesures réalisées sur la machine Ivanoe. Ils confirment les résultats des tests précédents notamment l'apport du *padding*. Les résultats sont d'ailleurs meilleurs puisqu'on passe d'un rapport supérieur à 6 entre *dddi* et *double* à un rapport avoisinant 3 entre *dddii* et *double*. Le rapport est d'environ 2.5 quand on compare les

float_st aux *float*. Les facteurs sont légèrement inférieurs aux facteurs théoriques puisque les réseaux des clusters ont été conçus pour optimiser les temps de communication. Cependant, bien que les courbes des figures soient globalement linéaires, nous notons par moment des sauts (*figure* 4.2). On aurait pu imaginer que ces sauts surviennent pour une même taille de donnée (en octet) échangée mais ce n'est pas le cas. Par exemple, sur la *figure* 4.2b, on observe un saut pour un vecteur de 8192 *double_st* et un autre pour un vecteur d'environ 20000 *double*. Si le saut dans la courbe des *double* avait lieu à partir de 28672, on aurait conclu que les sauts survenaient après une taille fixe de données échangées. En fait, ces sauts sont dus à un changement de mode d'envoi des données. Mais comme l'indique la documentation officiel du standard MPI [Message Passing Interface Forum, 2012], ces changements de mode d'envoi dépendent directement des implémentations du standard :

> *« The reluctance of MPI to mandate whether standard sends are buffering or not stems from the desire to achieve portable programs. Since any system will run out of buffer resources as message sizes are increased, and some implementations may want to provide little buffering, MPI takes the position that correct (and therefore, portable) programs do not rely on system buffering in standard mode. Buffering may improve the performance of a correct program, but it doesn't affect the result of the program. »*

En d'autres termes, l'envoi de message standard peut être fait en mode *bufferisé* ou en mode *synchrone*. Le choix du mode repose entièrement sur l'implémentation MPI. Finalement, nous retenons principalement de ces tests le facteur 4 entre les temps de communication d'un *float* (respectivement *double*) et ceux d'un *float_st* (respectivement *double_st*).

Après l'évaluation des temps de communication, nous avons trouvé pertinent de tester un programme parallèle utilisant CADNA_MPI. Nous avions choisi deux cas de test :

- le produit matriciel ;
- la résolution d'un système linéaire par la méthode de Gauss sans recherche du pivot maximum.

Figure 4.3 – Temps de communication en double précision sur la HP Z600 : nous comparons une implémentation du *double_st* avec le padding (*dddii*) à une autre sans le padding(*dddi*).

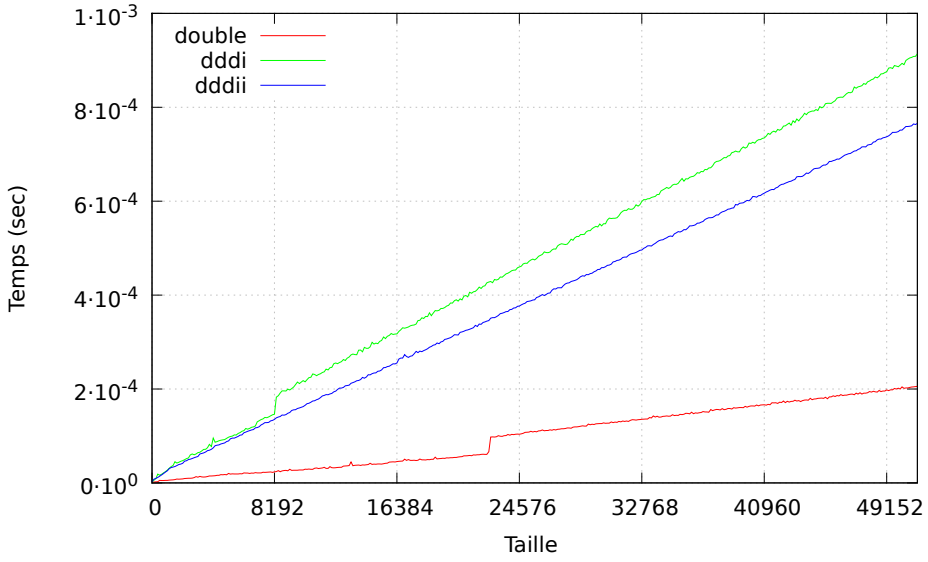
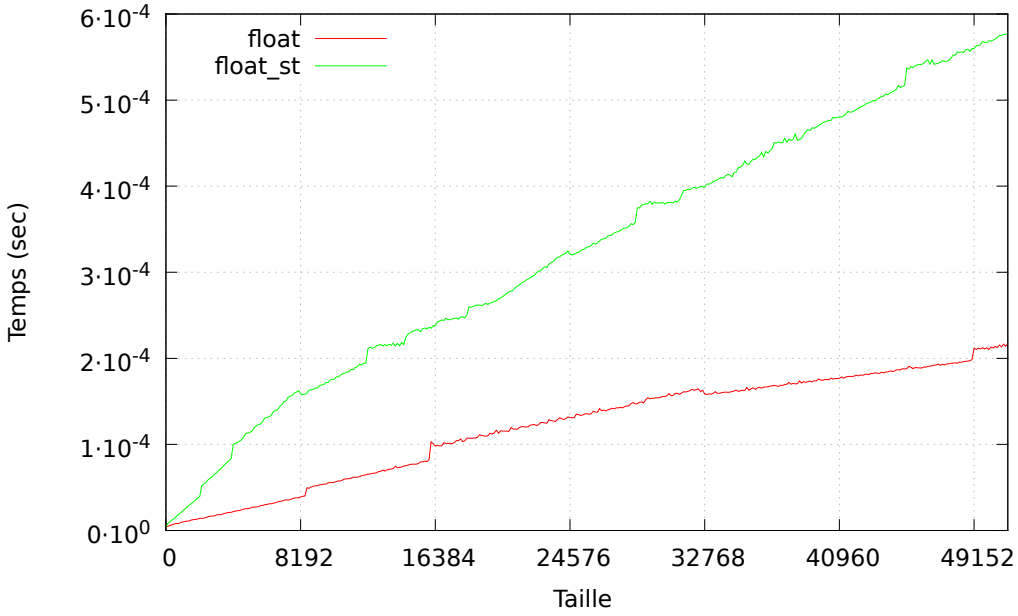


Figure 4.4 – Temps de communication sur la machine Ivanoe d'EDF R&D

(a) Simple precision



(b) Double precision

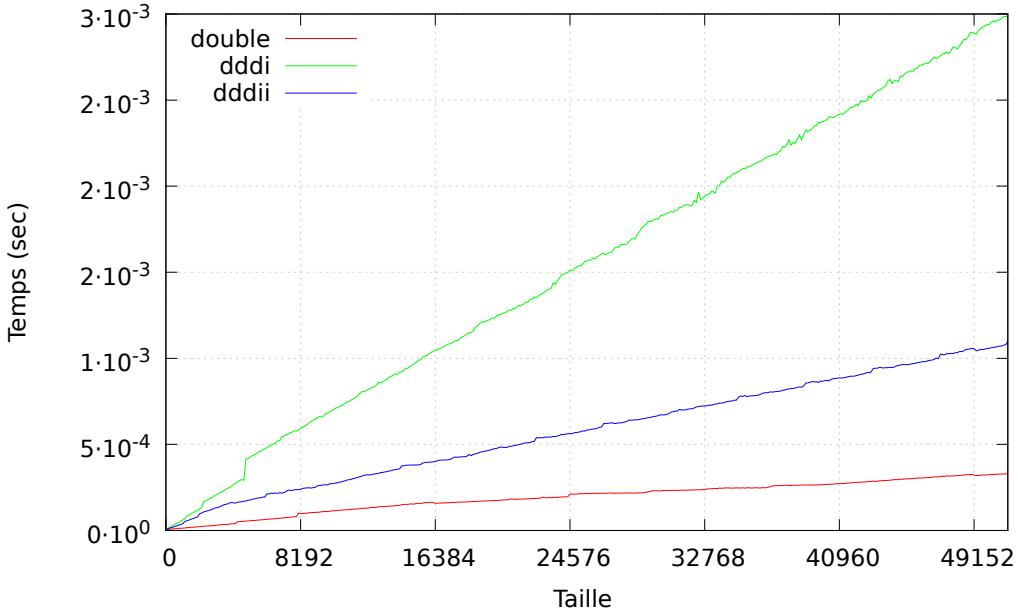


Tableau 4.1 – Produit matriciel séquentiel (double précision) avec et sans CADNA

Taille matrice carrée	256	512	1024	2048	4096
durée exécution avec CADNA (sec)	1.928	16.085	193.821	1639.183	13249.547
durée exécution sans CADNA (sec)	0.131	1.072	13.706	125.468	1352.716
rapport avec_CADNA/sans_CADNA	14,717	15,00	14,14	13,06	9,79

4.5.2 Étude d'un code de produit matriciel avec ou sans CADNA

Nous évaluons dans un premier temps le surcoût dû à CADNA en comparant des versions séquentielles du produit matriciel (voir *tableau* 4.1). Puis nous étudions le surcoût dû à CADNA_MPI, en comparant des versions parallèles reposant sur l'**algorithme** 4.1. Nous avons choisi de travailler en double précision sur un produit matriciel simple (non optimisé) de complexité $O(n^3)$. Les tests ont été effectués sur la HP Z600. La *figure* 4.5 présente les résultats pour des matrices carrées de taille 1024, 2048 et 4096. Nous présentons ensuite le rapport $TempsAvecCadnaMPI/TempsSansCadnaMPI$ en *figure* 4.6.

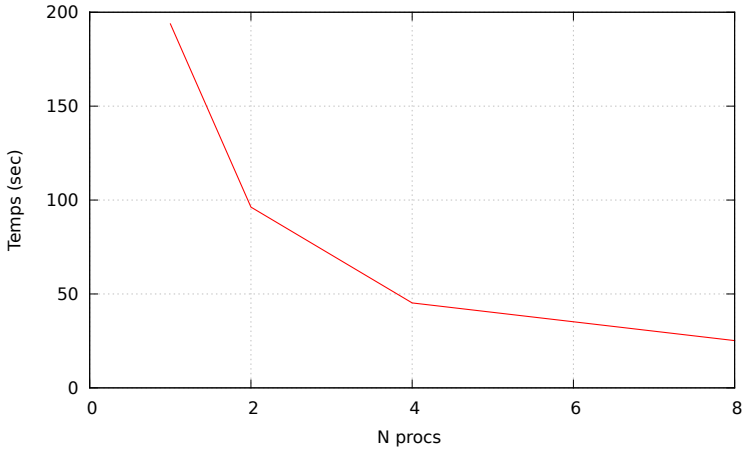
Alg. 4.1: Produit de deux matrices carrées en parallèle.

Paramètres en entrée : A et B matrices carrées ($n \times n$), np le nombre de proc
Paramètres en sortie : C matrice carrée $n \times n$
$nb = n/np$
if rang=0 **then**
  Envoyer matrice B à tous les processus
  Envoyer nb lignes de A à tous les processus
  Faire produit matriciel de $A(nb, n) \times B(n, n)$
  Réception des résultats des autres blocs
else
  Réception de matrice B
  Réception de nb lignes de A
  Faire produit matriciel de $A(nb, n) \times B(n, n)$
  Envoyer résultat à proc 0
end if

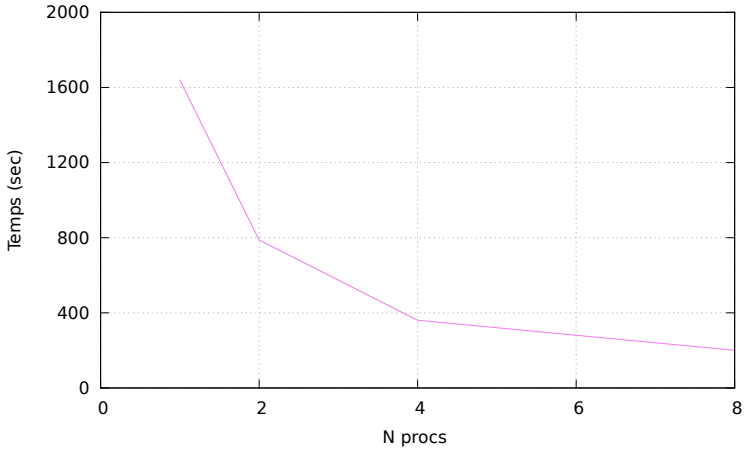
Nous avons utilisé CADNA sans l'auto-validation pour ne prendre en compte que les temps d'exécution. La *figure* 4.6 permet d'observer un rapport compris entre 10 et 15 entre les temps d'une exécution avec ou sans CADNA (CADNA_MPI) pour des matrices de taille inférieure à 4096. En fait, ce rapport n'est globalement pas influencé par le nombre de processus utilisé. Nous en déduisons que l'influence de CADNA_MPI est minime sur l'accélération d'un code (*figure* 4.5). En effet, le surcoût provient essentiellement de l'utilisation de CADNA (*tableau* 4.1), sous réserve que le code soit bien parallélisé et que les temps de communication soient négligeables devant le temps du au calcul (recouvrement des communications par le travail). Le code utilisé ici n'est pas optimisé. C'est pour cela que le rapport du cas de la matrice de 4096 est incohérent avec les autres rapports. Nous revenons sur les raisons du surcoût dans le chapitre 5 aux sections 5.3 et 5.4.

Figure 4.5 – Produit matriciel avec CADNA sur la HP Z600

(a) Matrice carré 1024



(b) Matrice carré 2048



(c) Matrice carré 4096

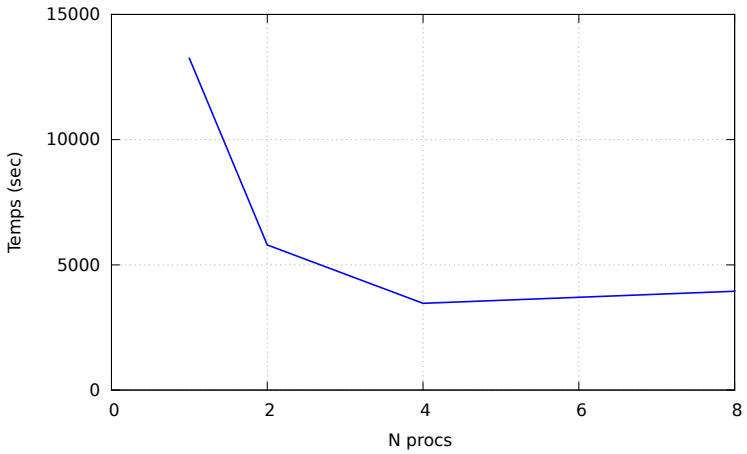
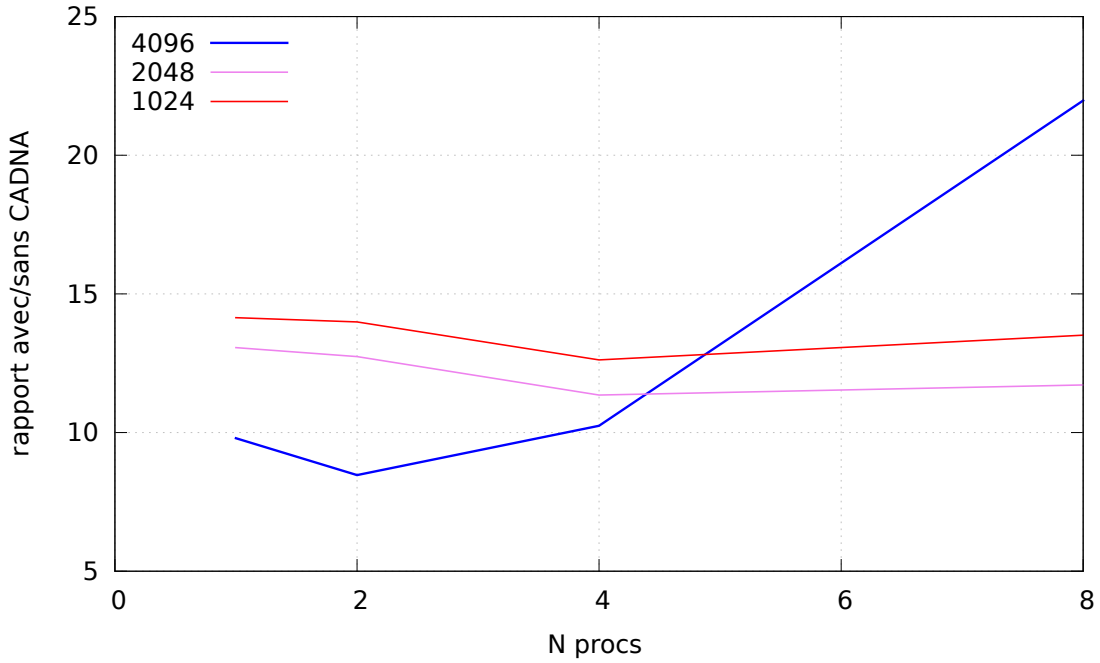


Figure 4.6 – Rapport temps d'exécution du produit matriciel AvecCadnaMPI/SansCadnaMPI sur la HP Z600



4.5.3 Élimination de Gauss sans recherche du pivot maximum

Dans un deuxième temps, nous avons travaillé sur un code de résolution de système linéaire par la méthode de Gauss sans pivotage. Nous avons choisi volontairement un algorithme reconnu instable numériquement. L'objectif ici est de montrer que CADNA_MPI détecte les instabilités commises sur chaque processus. Le mode d'auto-validation de CADNA est activé dans ce test.

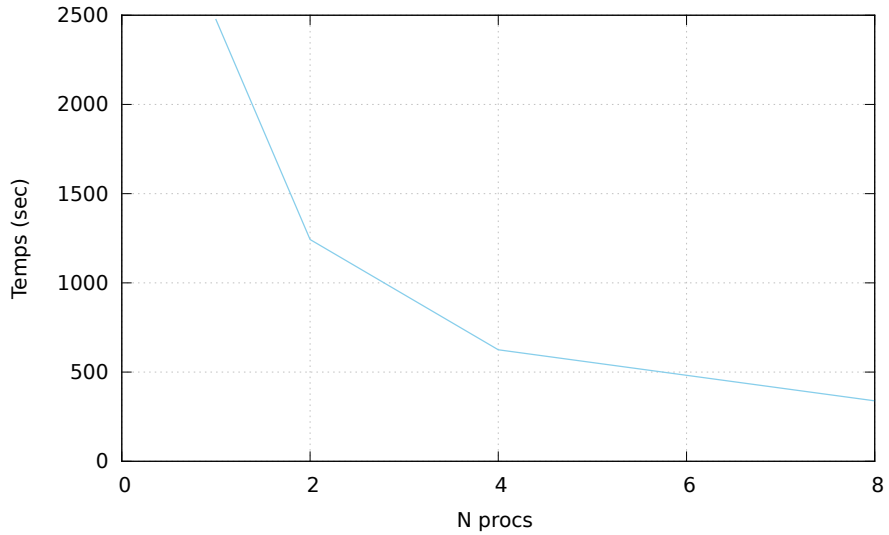
La *figure* 4.7a présente le temps d'exécution du code avec CADNA_MPI. La courbe de la *figure* 4.7b, nous présente le nombre d'instabilités numériques détectées par CADNA_MPI sur chaque processus en fonction du nombre total de processus utilisés. On peut constater que si on multiplie le nombre total de processus utilisés par le nombre d'instabilités numériques détectées par processus on retrouve le nombre d'instabilités numériques détectées lors de l'exécution séquentielle. Dans notre cas de test, le nombre important d'instabilités détectées s'explique par le fait qu'on traite des matrices aléatoires et qu'on utilise un algorithme de Gauss sans pivotage.

4.6 Conclusion

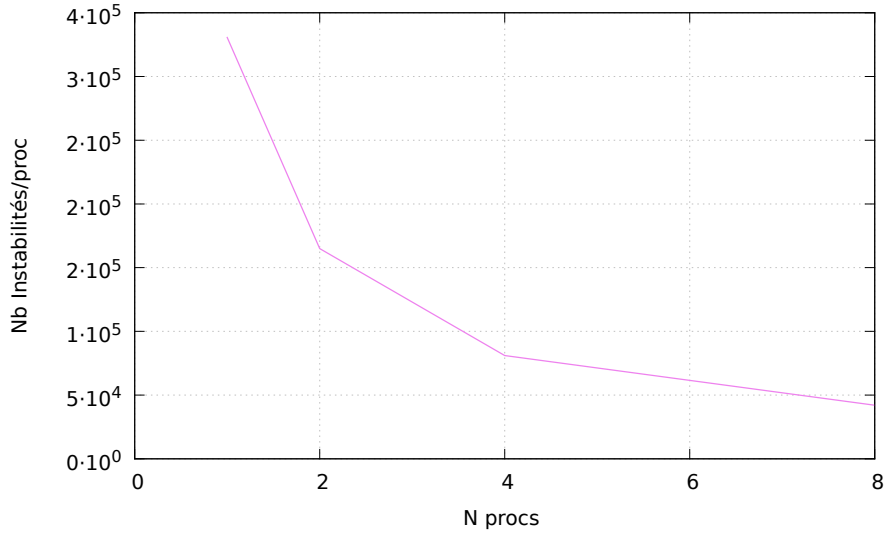
Dans ce chapitre, nous avons présenté une extension de CADNA pour le standard MPI. Nous avons montré que le temps de communication des types stochastiques était 3 à 4 fois plus long que celui des types standards. CADNA_MPI permet également d'avoir un diagnostic complet des instabilités détectées sur chaque processus. CADNA_MPI est complètement fonctionnel et a été testé sur divers cas. Il a notamment été utilisé pour valider le schéma de communication du code Telemac-2D (voir chapitre 6 d'EDF R &D [Moulinec *et al.*, 2011a]. Notons que, l'utilisation des *padding* permet d'obtenir un rapport de 4 entre la taille des types stochastiques et ceux des types conventionnels (*float*, *double*). Ce rapport permet d'avoir les données

Figure 4.7 – Résolution d'un système linéaire avec la méthode de Gauss sans pivotage, matrice A de taille 2048

(a) Temps d'exécution



(b) Nombre d'instabilités détectées par processus



stochastiques alignées en mémoire et donc de meilleures performances pour les temps de communication. L'implémentation de CADNA dans les routines BLACS utilise la même démarche que celle employée pour le standard MPI. Il s'agit d'un travail de développement informatique que nous présentons en annexe de ce manuscrit (annexe B).

Après l'implémentation de la bibliothèque CADNA dans les standards de communication, les travaux algorithmiques effectués pour une utilisation efficace de l'arithmétique stochastique discrète dans les bibliothèques de calcul scientifique sont exposés dans le prochain chapitre.

Chapitre 5

Vers une implémentation efficace de CADNA dans les routines BLAS

Sommaire

L'outil de validation numérique CADNA, que nous avons présenté au chapitre 3 est facile à utiliser et très adapté pour le contexte industriel. Il est assez simple à implémenter sur un code séquentiel écrit dans un seul langage (Fortran90, C/C++) et a pour principaux avantages sa capacité à localiser les pertes de précision et à proposer un véritable diagnostic à la fin des exécutions. Cependant, comme nous l'évoquons (chapitre 1), il ne peut être utilisé totalement dans les codes industriels puisque ces derniers, dans l'optique d'optimiser leurs performances, font généralement appel à des bibliothèques externes. Il importe alors d'implémenter des extensions à l'outil CADNA, afin de le rendre compatible avec les principales bibliothèques de calcul scientifique. Dans ce chapitre, nous nous proposons de travailler sur la problématique de l'implémentation de l'arithmétique stochastique discrète dans les principales bibliothèques d'algèbre linéaire utilisées dans le cadre du calcul scientifique. Nous nous intéressons en particulier aux routines BLAS. La méthodologie mise en place peut ensuite être étendue aux autres bibliothèques.

Dans le cadre du travail présenté dans ce chapitre, les bibliothèques scientifiques ont été comparées en effectuant des tests de performance. Tous les tests ont été réalisés sur un poste scientifique type EDF R&D : *HP Z600 workstation*. Cette machine, ainsi que le protocole de mesure de temps sont présentés en annexe du manuscrit (annexe A, respectivement aux sections A.2 et A.1). Notons que ce chapitre est une version étendue des résultats présentés dans les articles [Montan *et al.*, 2013a] et [Montan *et al.*, 2012a].

Plan du chapitre : Nous présentons ici toute notre réflexion sur l'implémentation de la DSA dans les routines BLAS. Nous commençons d'abord par présenter brièvement les principales bibliothèques de calcul scientifique (section 5.1). Nous nous intéressons particulièrement aux routines BLAS en section 5.2. Puis, dans la section 5.3, nous nous posons la question de la façon la plus simple d'intégrer l'arithmétique stochastique discrète dans ces routines. Enfin, nous nous consacrons pleinement à une implémentation efficace de la DSA dans la routine DGEMM des BLAS (section 5.4 et section 5.5). Pour améliorer les performances de cette routine DgemmCadna, nous proposons une nouvelle implémentation de la méthode CESTAC dans la section 5.6.

5.1 Sur les outils de calcul scientifique

Dans le chapitre 1, le processus de simulation numérique et ces différentes étapes ont été évoqués. La simulation numérique est finalement matérialisée par le logiciel de calcul. L'architecture des codes que nous avions étudiés au cours de cette thèse peut être, dans la plupart des cas, schématisée sous forme d'une chaîne de cinq grandes étapes successives :

1. la génération de maillage ;
2. le partitionnement ;
3. la discrétisation (temps, espace) ;
4. la résolution des équations discrétisées ;
5. le post-traitement et visualisation des résultats.

Ces différentes étapes sont toutes autant importantes puisque les résultats de l'une servent d'entrées à l'étape suivante. Toutefois, l'étape la plus coûteuse en temps correspond à la résolution des équations discrétisées. Bien souvent, cette résolution revient à traiter des problèmes

d'algèbre linéaire. Le choix de la méthode résolution est fait en fonction du type problème, des données traitées (par exemple matrice creuse, pleine) et de ses propriétés mathématiques, et enfin de la qualité de la solution ou de la robustesse de l'algorithme à mettre en place.

Figure 5.1 – Classification des bibliothèques scientifiques.

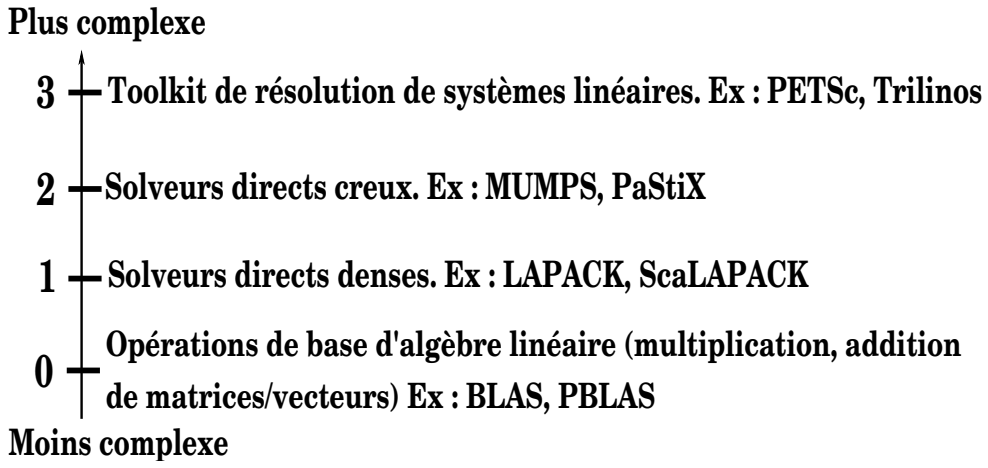


Tableau 5.1 – Liste non exhaustive des bibliothèques (gratuites) d'algèbre linéaire

Objectifs	Bibliothèques
Renumeroteurs, partitionneurs, équilibrage de charge	METIS, SCOTCH, JOSTLE etc
Produits supports	Armadillo, BLAS, BLIS, Blitz++, uBLAS, PSBLAS, MTL4, Trilinos, pOSKI etc
Solveurs directs denses	Eigen, Elemental, FLAME, Lapack, MAGMA, PLASMA, ScaLAPACK, Trilinos, ViennaCL etc
Solveurs directs creux	SuperLU, MUMPS, PaStiX, Trilinos, UMFPACK,CHOLMOD,KKTDirect etc
solveurs itératifs creux	DUNE/ISTL,GMM++, HIPS,HYPRE, ITSOL, PETSc etc
Solveurs modaux denses/creux	ARPACK, BLOPEX, FEAST, FILTLAN, PRIMME, SLEPc, TRLan, Trilinos etc
Préconditionneurs	BPKIT, MSPAI, Trilinos etc

Concevoir entièrement un code de simulation est une tache extrêmement compliquée. Bien souvent, pour gagner en productivité en en maintenance, on fait appel à des outils externes. Outre le fait d'avoir un développement plus rapide, l'utilisation de ces outils permet aussi de conjuguer efficacité, fiabilité, performance et portabilité. Il serait difficile –voire impossible– de lister tous les outils de calcul scientifique existants actuellement (voir *tableau* 5.1 pour une liste non exhaustive des bibliothèques d'algèbre linéaire gratuites). Notons tout simplement, qu'il est possible de les regrouper en 4 catégories suivant leur degré de complexité (voir *figure* 5.1). Les outils les plus complexes sont basés sur les moins sophistiqués. A ces quatre catégories de bibliothèques, nous ajoutons une dernière que nous qualifions de bibliothèques complémentaires. Il s'agit des outils dédiés à la gestion des graphes, à l'équilibrage de charge, etc. Citons

à titre d'exemple la bibliothèque de partitionnement de graphe METIS qui est utilisée dans les principaux codes d'EDF (Telemac, Code_Aster).

En somme, il existe plusieurs types d'outils qui reposent les uns sur les autres. Les librairies dédiées aux opérations de base de l'algèbre linéaire (produit scalaire, multiplication de matrices, etc) constituent la base de cette chaîne. Dans les sections à venir, nous nous intéresserons aux implémentations BLAS, qui définissent une interface standard pour les principales routines d'algèbre linéaire.

5.2 Les routines BLAS

> *« Numerical linear algebra [. . .] is fundamental to most calculations in scientific computing, and is often the computationally intense part of such calculations. Designers of computer programs involving linear algebraic operations have frequently chosen to implement certain low level operations, such as dot product or the matrix vector product, as separate subprograms. [. . .] The programming of some of these low level operations involves algorithmic and implementation subtleties that need care, and can be easily overlooked. If there is general agreement on standard names and parameter lists for some of these basic operations, then portability and effiency can also be achieved. »*
>
> Basic Linear Algebra Subprograms Technical (Blast) Forum, Août 2001.

Les Routines BLAS -*Basic Linear Algebra Subprograms*- définissent une interface de programmation standard (en anglais API : *Application Programming Interface*) pour l'algèbre linéaire. Pour une opération d'algèbre linéaire donnée, une interface standard est définie via une déclaration type de fonction [Basic Linear Algebra Subprograms Technical (Blast) Forum, 2001]. La définition standardisée permet d'avoir des implémentations différentes selon les développeurs mais toutes interchangeables. Les fonctionnalités des BLAS sont réparties en 3 niveaux ou *level* :

BLAS 1 : opérations sur les vecteurs de la forme $y = \alpha x + y$ ainsi que les opérations de produit scalaire et de norme ; complexité des routines *O(N)*.

BLAS 2 : opérations matrice-vecteur de type $y = \alpha A \times x + \beta y$; α et β étant des scalaires, x et y des vecteurs et A une matrice ; complexité des routines $O(N^2)$.

BLAS 3 : opérations matrice-matrice et notamment la très utilisée opération de multiplication de matrices xGEMM $C = \alpha A \times B + \beta C$. α et β sont des scalaires, A, B et C des matrices ; complexité des routines $O(N^3)$.

Les premiers efforts de standardisation des routines d'algèbre linéaire ont conduit à la spécification de 38 sous-programmes du niveau 1 des BLAS en 1979 [Lawson *et al.*, 1979]. Ces spécifications ont alors permis une première implémentation qui a servie au développement de LINPACK [Dongarra, 1979]. Cette première version était écrite en Fortran 77 et était principalement consacrée aux opérations sur les vecteurs. Puis, avec l'apparition des machines vectorielles et des machines à mémoires hiérarchiques, des spécifications pour les routines *level* 2 et 3 des BLAS ont été proposées [Dongarra *et al.*, 1988b, Dongarra *et al.*, 1990b] puis conçues [Dongarra *et al.*, 1988a, Dongarra *et al.*, 1990a] toujours en Fortran 77. A ces trois premières implémentations, se sont rajoutées deux autres extensions : une pour les matrices creuses [Dodson *et al.*, 1991, Duff *et al.*, 1997] et une autre implémentation en précision mixte et étendue [Li *et al.*, 2002]. L'ensemble de toutes ces parties constitue la version Netlib des BLAS. Cette version

non-optimisée est considérée comme la version de référence des routines BLAS du fait de la simplicité de ces algorithmes.

L'implémentation des routines d'algèbre linéaire a profondément évolué au cours des quatre dernières décennies [Dongarra, 2011]. Cette évolution s'est faite de pair avec celle des architectures matérielles. Initialement, les premières routines (BLAS 1) ont été développées dans le cadre du projet LINPACK (LINear algebra PACKage), ensemble de routines Fortran dédiées à la résolution de systèmes linéaires [Dongarra, 1979]. Signalons que le High Performance Linpack (HPL) benchmark, test de performance qui est [48] utilisé pour classer les ordinateurs les plus puissants du monde est issu accidentellement de cette bibliothèque [Dongarra *et al.*, 2003]. Par la suite, les routines BLAS 2 et 3 ont été implémentées. Les BLAS 3 ont alors contribué à l'avènement de la bibliothèque LAPACK (Linear Algebra PACKage) [Anderson *et al.*, 1999] au début des années 90. A l'image de LINPACK, LAPACK est également dédié à la résolution des problèmes d'algèbre linéaire (moindres carrés, calcul des valeurs propres, etc) mais avec une contrainte de performance. C'est la raison pour laquelle, le développement de LAPACK a été basé sur les routines BLAS 3. Celles-ci peuvent être optimisées pour chaque architecture matérielle, ce qui permet à la bibliothèque d'être à la fois portable et performante. L'apparition des machines à mémoire distribuée a entraîné le développement de ScaLAPACK (Scalable LAPACK) [Blackford *et al.*, 1997]. ScaLAPACK repose une approche SPMD (Single Program Multiple Data). Les routines ScaLAPACK font appel à trois bibliothèques PBLAS (Parallel BLAS), BLACS (Basic Linear Algebra Communication Subprograms) et BLAS (voir *figure* B.1). L'émergence des machines multi-cœurs (au début des années 2000) a également entraîné une évolution avec la création des bibliothèques PLASMA et MAGMA [Agullo *et al.*, 2009]. Ces dernières sont dédiées respectivement aux exécutions sur architectures multi-cœurs et sur les systèmes hybrides (multi-cœurs, GPU).

Il existe plusieurs implémentations des BLAS. Les différences entre ces versions sont d'ordre algorithmique avec d'importantes conséquences sur les performances. Dans la prochaine section, nous présentons les principales implémentations.

5.2.1 Les différentes versions

L'algèbre linéaire, particulièrement la résolution des systèmes linéaires, le calcul de valeurs propres, la méthode des moindres carrés, le produit matriciel, constitue une pierre angulaire du calcul haute performance. Elle est d'autant plus importante que les principaux codes de calcul scientifique sont basés sur des opérations d'algèbre linéaire. La grande partie de leurs temps d'exécution est consacrée à ces opérations. Au vu de son importance, plusieurs travaux ont été menés afin de les optimiser et d'exploiter au maximum les capacités des plateformes sur lesquelles ces opérations seront exécutées.

Auteurs	versions BLAS
AMD	ACML - *AMD Core Math Library* [Advanced Micro Devices, 2012]
Apple	*Accelerate Framework* [Apple Inc., 2011]
Intel	MKL - *Math Kernel Library* [Intel, 2013]
IBM	ESSL - *Engineering and Scientific Subroutine Library* [IBM, 2013]

Tableau 5.3 – Les BLAS optimisés par des constructeurs

48. Un nouveau test de performance « High Performance Conjugate Gradient (HPCG) » benchmark a été proposé [Heroux et Dongarra, 2013]

Ainsi, il existe aujourd'hui plusieurs versions optimisées, chacune ayant été optimisée soit pour une architecture donnée soit par un éditeur de logiciel (voir *tableau* 5.3). Ces dernières sont généralement payantes. Il existe cependant des solutions alternatives. On peut, par exemple, utiliser ATLAS (*Automatically Tunes Linear ALgebra Software*) [Clint Whaley *et al.*, 2001] qui génère automatiquement une version optimisée et adaptée à l'architecture sur laquelle elle sera installée. ATLAS implémente un ensemble de techniques appelé AEOS (Automated Empirical Optimization Software) [Clint Whaley *et al.*, 2001]. Les codes sources sont générés automatiquement pendant l'installation à partir de différents paramètres de la machine (exemple : taille des caches). ATLAS est sous licence libre et sa distribution est autorisée. On le retrouve dans les principales distributions Linux. Des binaires Windows sont également disponibles.

GotoBLAS [Goto et van de Geijn, 2008a,Goto et van de Geijn, 2008b] est elle aussi libre. Développée par Kazushige Goto [49] de l'Université du Texas, cette implémentation des BLAS est disponible en téléchargement gratuit depuis `http://www.tacc.utexas.edu/tacc-projects/gotoblas2`. Contrairement à ATLAS, elle n'est pas intégrée directement dans les distributions Linux, sa licence n'autorisant pas sa distribution. Son installation et sa désinstallation demeurent très facile. Les algorithmes de GotoBLAS sont basés sur l'utilisation des blocs et sous-blocs de matrices. Dans un souci d'optimisation maximale, certaines parties ont été écrites en code assembleur. Avec la MKL d'Intel, elles sont à ce jour les versions les plus performantes (voir figures 5.3, 5.4, 5.2).

Une version "templatisée" des BLAS est également disponible : Linalg [Trebuchet, 2010]. Elle a été développée par Phillipe Trebuchet du LIP6 et se base sur le concept de « template » du C++ (patron en français). Les algorithmes mis en place dans Linalg sont basés sur ceux de la version de référence Netlib. Notons que le concept de template du C++ permet de définir un modèle de fonction dont le type de retour et le type des arguments n'est pas fixé. Il est alors possible de passer en paramètre des types et ainsi de définir des fonctions génériques. C'est une alternative à la surcharge (ou surdéfinition) de fonction, qui permet de gagner en performance, en temps de codage, et surtout en clarté [Kirschenmann, 2012]. Les templates font parties des grands apports du C++ par rapport au langage C. Le concept de template ne s'arrête pas aux fonctions, on peut aussi l'utiliser pour les classes et les structures. Linalg nous sera très utile dans nos travaux, elle permet d'avoir une version de "référence" en termes de temps de calcul pour la routine DgemmCadna. En effet, Linalg nous permet d'avoir une version de DgemmCadna avec un effort minimal.

A ces diverses versions, on peut ajouter les implémentations orientées architectures matérielles telle que PLASMA et MAGMA [Agullo *et al.*, 2009] que nous avons mentionnées plutôt. Citons également la bibliothèque cuBLAS [NVIDIA, 2013] de NVIDIA qui a été créée afin de profiter des performances des processeurs graphiques NVIDIA.

Afin d'évaluer les différentes implémentations des BLAS, des tests de performances ont été réalisés. Les principaux résultats obtenus sont présentés dans la prochaine section.

5.2.2 Tests de performance des BLAS

Nous avons arbitrairement choisi de tester une routine pour chaque niveau des BLAS : xAXPY pour le niveau 1, xGEMV pour le niveau 2, xGEMM pour le niveau 3, ceci dans le but d'avoir une radiographie représentative des BLAS. La routine xAXPY permet d'effectuer le produit d'un scalaire et d'un vecteur : $y = \alpha x + y$; xGEMV permet d'effectuer une opération matrice/vecteur : $y = \alpha \times A \times x + \beta \times y$ et xGEMM pour le produit matriciel. La lettre x au

49. `http://en.wikipedia.org/wiki/Kazushige_Goto`

début des noms des routines est remplacée par la S quand il s'agit de la simple précision et D pour la double précision. Les tests ont été effectués en simple puis en double précision.

Quatre implémentations différentes ont été comparées :

- Netlib (installée à partir de la version disponible à `http://www.netlib.org/blas/`) ;
- ATLAS version 3.8.3-27 (version disponible sur Debian Squeeze) ;
- MKL de la suite logiciel Intel® C++ Composer XE 2011 Update 2 ;
- GotoBLAS2.

Comme nous l'évoquons au début de ce chapitre, les tests ont été effectués sur la HP Z600. Sa puissance crête théorique est estimée à 153.6 Gflops en simple précision et 8 threads peuvent y être exécutés simultanément grâce à la technologie Hyper-Threading d'Intel. Nous avons principalement utilisé le compilateur GNU GCC version 4.6 et le compilateur Icc 12.0.2 pour Intel MKL.

L'histogramme de la *figure* 5.2 présente la vitesse d'exécution -nombre d'opérations à virgule flottante par seconde (*Gflops*)- pour une matrice de taille 8192×8192 (respectivement vecteur de taille 8192). Les figures 5.3 et 5.4 présentent les résultats complets.
L'analyse des résultats permet de faire les constats suivants :

- Netlib est la version la moins performante, constat somme toute logique. Netlib est une version non optimisée qui sert de référence pour toutes les implémentations des BLAS.
- ATLAS n'est pas très performante, ce constat n'était pas prévisible : il s'explique par le fait que nous utilisons la version disponible directement dans notre distribution Linux (Calibre 7), cette version n'a pas été compilée spécialement pour notre machine. Pour obtenir de meilleures performances, il aurait fallu réinstaller ATLAS en local. Nous avons utilisé cette version pré-compilée d'ATLAS parce qu'il est difficile (voire impossible) d'installer correctement ATLAS en local sur nos machines avec un compte utilisateur. Ceci reflète les contraintes des systèmes d'informations (SI) d'une entreprise comme EDF, où toute bibliothèque informatique doit d'abord avoir été certifiée et qualifiée par les autorités compétentes avant d'être installée.
- Les exécutions sont deux fois plus rapides en simple précision. C'est tout à fait normal puisque les opérations en double précision nécessitent deux fois plus de ressources (registres).
- Les versions GotoBLAS et MKL sont les versions les plus performantes. Ces versions atteignent des performances légèrement supérieure à 140 Gflops pour la fonction SAXPY (voir *figure* 5.6a) soit un peu plus de 91% de performance crête de la machine.

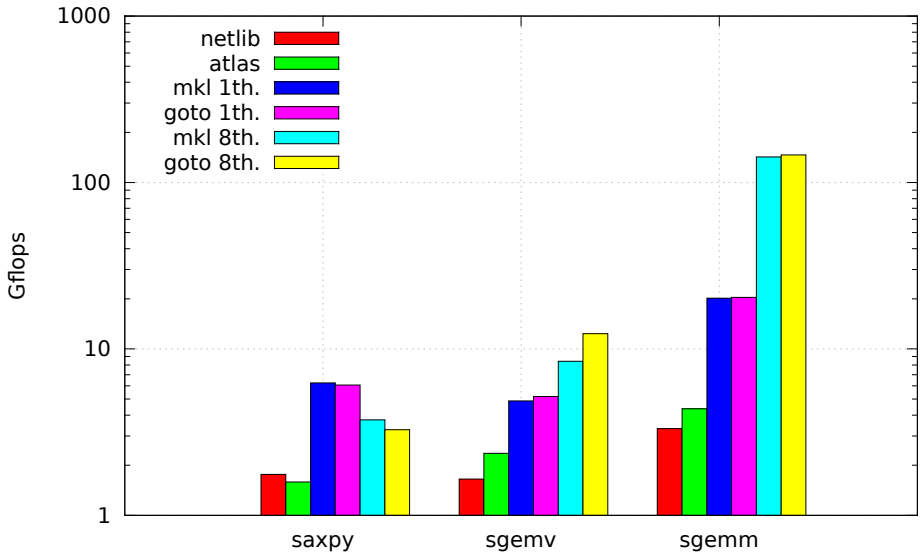
En outre, il n'est pas superflu de remarquer que la version de Goto (version gratuite) concurrence celle d'Intel (version payante). Cette remarque est principalement d'ordre philosophique. Doit-on y trouver une source de motivation supplémentaire à encourager les projets open source ? Pour la petite histoire, Mr Kazushige Goto a arrêté de maintenir sa version des BLAS. Il a dans un premier temps été embauché par Microsoft et est employé d'Intel depuis Juillet 2012.

Fermons cette courte parenthèse et continuons l'analyse de nos résultats. Nous pouvons ainsi noter que pour la routine du niveau 1 (*figure* 5.6a et *figure* 5.7a), les exécutions séquentielles de GotoBLAS et MKL sont plus efficaces que les exécutions *multi-thread*. On peut naturellement s'interroger sur la pertinence de l'optimisation des routines BLAS. *Pourquoi l'exécution single-thread est-elle la plus performante pour le niveau 1 ? Pourquoi l'exécution multi-thread est très efficace pour le niveau 3 et moindre pour le niveau 2 ?*

Quelques éléments de réponses sont apportés dans [Clint Whaley *et al.*, 2001]. En effet, les gains de performance des versions optimisées sont étroitement liés aux différents niveaux des

Figure 5.2 – Comparaison des versions des BLAS : FLOPS pour les routines xAXPY, xGEMV et xGEMM en simple et double précision (échelle semi-logarithmique). On considère ici des matrices carrées de taille 8192 et des vecteurs de taille 8192. On met en évidence ici les implémentations les plus performantes. On remarquera que toutes les versions sont meilleures en simple précision, la double nécessitant plus de ressources.

(a) Simple Précision



(b) Double Précision

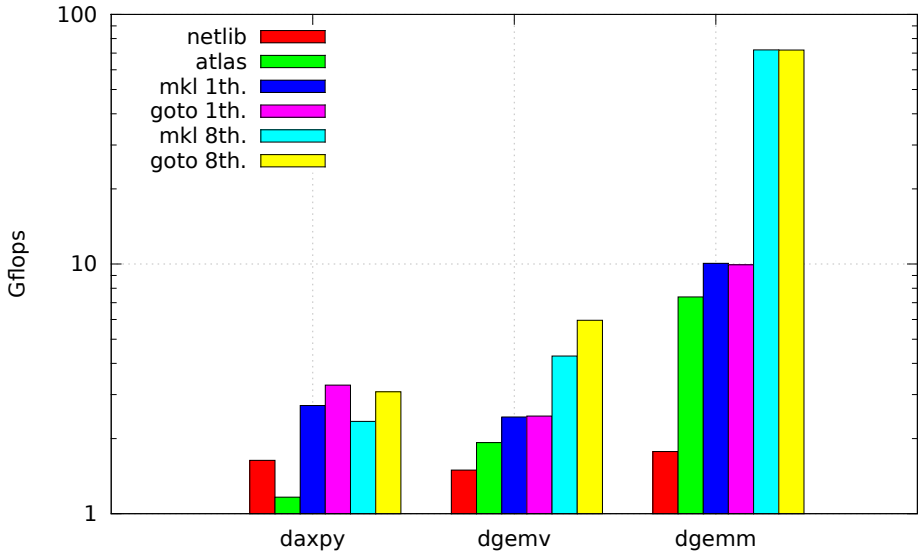
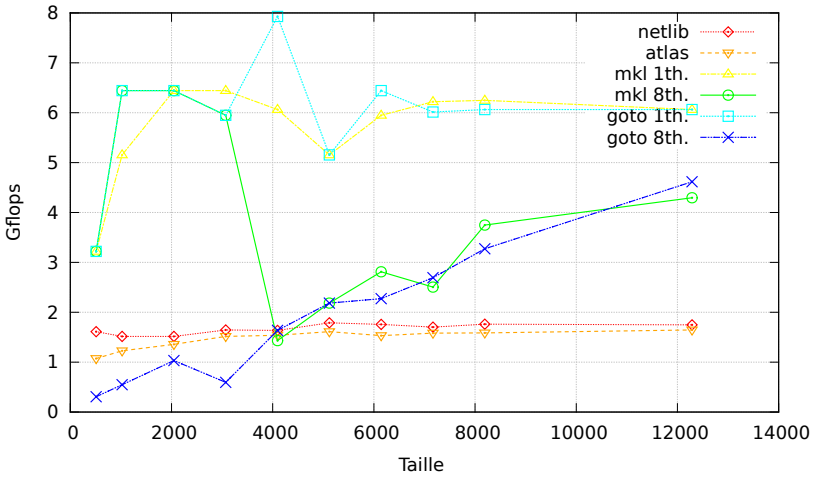
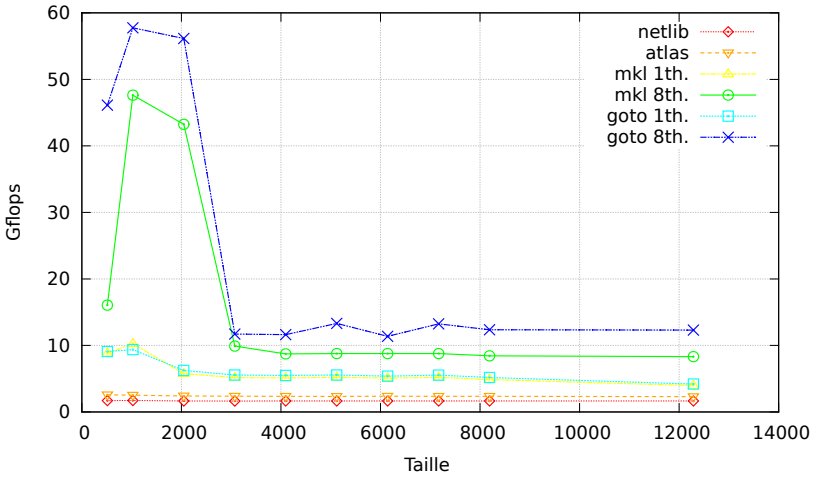


Figure 5.3 – Comparaison des versions des BLAS : FLOPS pour les routines SAXPY, SGEMV et SGEMM

(a) saxpy



(b) sgemv



(c) sgemm

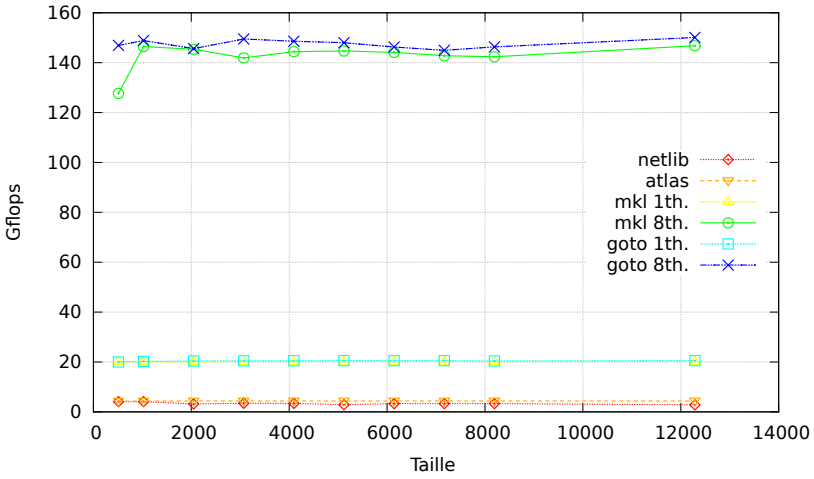
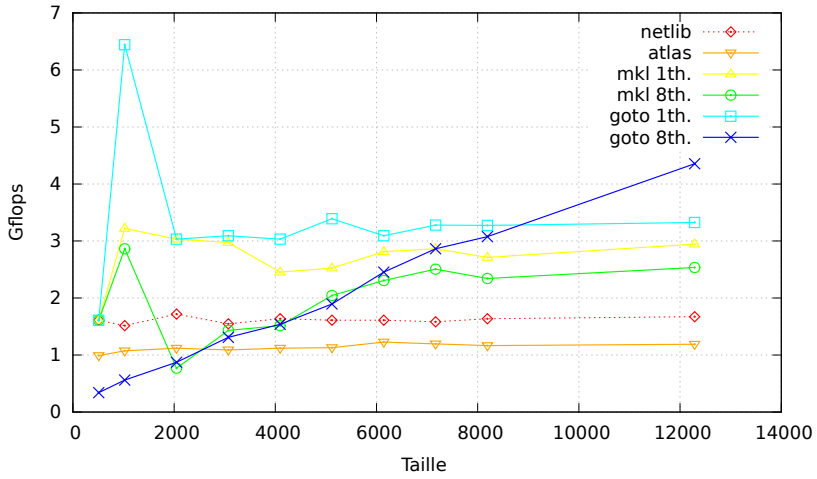
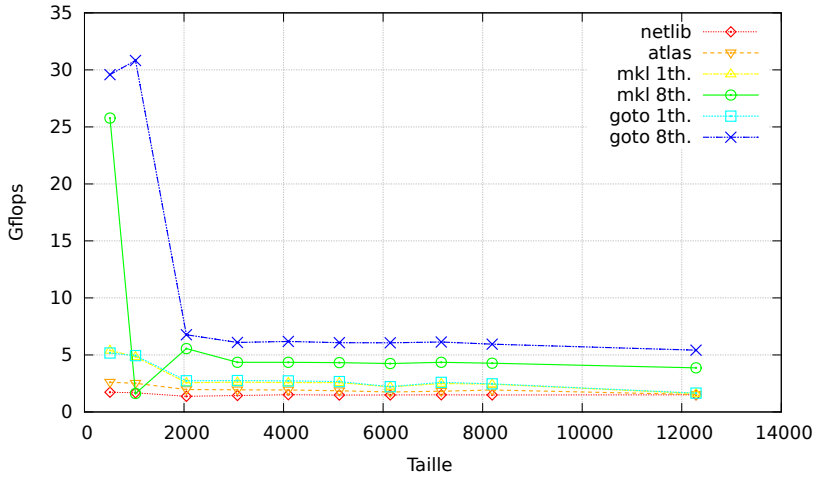


Figure 5.4 – Comparaison des versions des BLAS : FLOPS pour les routines DAXPY, DGEMV et DGEMM

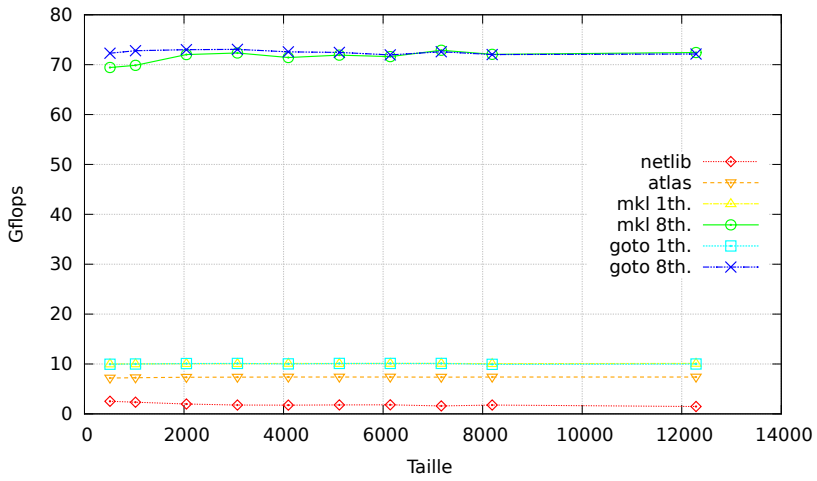
(a) daxpy



(b) dgemv



(c) dgemm



BLAS, aux nombres d'accès mémoire et au nombre d'opérations en arithmétique flottante (voir *tableau* 5.4).

Tableau 5.4 – Nombre d'accès mémoire et d'opérations flottantes en fonction du niveau des BLAS

Blas	Mops	Flops	Flops/Mops
Niveau 1 (AXPY)	$3n$	$2n$	$2/3$
Niveau 2 (GEMV)	n^2	$2n^2$	2
Niveau 3 (GEMM)	$4n^2$	$2n^3$	$n/2$

Pour les routines du niveau 1, les gains sont minuscules ($0 - 15\%$) car on ne peut y appliquer que des techniques d'optimisation classiques telles que les déroulements de boucle. La complexité de ces routines étant de *O(N)*, aucune optimisation ne peut être faite pour améliorer l'utilisation de la mémoire qui, rappelons le, est le facteur limitant. Pour le niveau 2, on peut réduire le coût des accès mémoire du vecteur de $O(N^2)$ à *O(N)* en ne parcourant le vecteur qu'une seule fois [Clint Whaley *et al.*, 2001, p 26] ; à chaque fois qu'on accède à un champ du vecteur, on effectue tous les calculs qui nécessitent cet élément. Cependant, les coûts (en accès mémoire) liés à la matrice sont irréductibles. On peut alors espérer obtenir des accélérations raisonnables mais rien d'exceptionnel.

Les routines du niveau 3 ont une complexité de $O(N^3)$. En réorganisant l'algorithme (utilisation des blocs) et en travaillant sur les trois boucles imbriquées, il est possible obtenir de biens meilleures accélérations. Citons à ce propos Whaley qui explique dans [Clint Whaley *et al.*, 2001] :

> *« Finally, the level 3 BLAS can display orders of magnitude speedups. To simplify greatly, these operations can be blocked such that the natural* $O(N^3)$ *fetch costs become essentially* $O(N^2)$*. further, the triply-nested loops used here are almost always too complex for the compiler to figure out without hints from the programmer and thus the* $O(N^3)$ *computation cost can be greatly optimized as well. »*

De fait, on comprend mieux pourquoi les versions optimisées sont très efficaces pour le DGEMM (voir *figure* 5.7c).

5.2.3 Conclusion

L'existence de plusieurs versions des BLAS a principalement été motivée par l'importance de l'algèbre linéaire dans le calcul scientifique et l'évolution des processeurs. Plusieurs travaux ont alors été consacrés à l'optimisation des BLAS et ont donné naissance aux différentes versions que nous connaissons aujourd'hui. Cependant, il importe de remarquer que l'optimisation n'impacte réellement que les performances des routines du niveau 3. C'est la raison pour laquelle nous avons trouvé pertinent de travailler principalement sur une routine du niveau 3 : DGEMM. Notons également que la routine DGEMM est une des plus importantes des routines BLAS, les principales opérations d'algèbre linéaire peuvent se ramener généralement à un ou plusieurs produits de matrice.

Avant de nous consacrer pleinement à l'implémentation de l'arithmétique stochastique discrète dans la routine DGEMM, intéressons nous d'abord à la compatibilité de l'arithmétique stochastique discrète et des routines BLAS dans la prochaine section.

5.3 Les routines BLAS et l'arithmétique stochastique discrète

Dans cette section, notre objectif est d'apporter des éléments de réponse à la question suivante : *les routines BLAS sont elles compatibles avec l'arithmétique stochastique ?* Il s'agit ici de déterminer l'impact réel de l'utilisation couplée de la bibliothèque CADNA et des routines d'algèbre linéaire. En d'autres termes, notre objectif est de répondre à la question : *quelles peuvent être les conséquences d'une implémentation directe de la DSA dans les BLAS ?* En ce sens, les versions de BLAS suivantes : Netlib, Linalg avec les types standard et Linalg utilisée avec les types stochastiques sont comparées. De plus, nous considérons les deux modes de fonctionnement de CADNA que nous avons définis en section 3.3 : avec et sans auto-validation. Rappelons que le mode auto-validation permet l'auto-validation de la multiplication et de la division comme l'exige la méthode de CESTAC (section 3.2.4) et la détection de toutes les instabilités numériques.

Le *tableau* 5.5 présente un récapitulatif des surcoûts par rapport à Netlib. Nous présentons les résultats complets de notre expérimentation dans les histogrammes des figures 5.6 et 5.7. La *figure* 5.5 met un peu plus en exergue les surcoûts puisque l'échelle utilisée n'est plus semi-logarithmique.

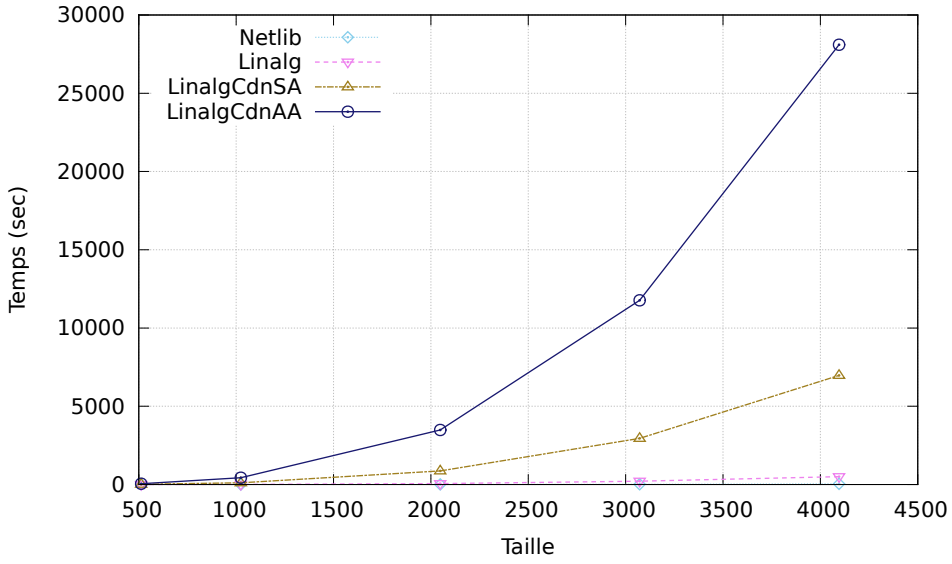
Tableau 5.5 – Tableau comparatif des temps d'exécution des routines xAXPY, xGEMV, xGEMM en simple et double précision pour une matrice carrée et d'un vecteur de taille 4096. Nous comparons quatre différentes implémentations Netlib, Linalg avec les types standards, Linalg avec les types stochastiques avec auto-validation (Avec A-V) ou sans (Sans A-V) ; les valeurs présentées entre parenthèse représentent le surcoût par rapport à la routine Netlib.

	Netlib	Linalg		Linalg + CADNA			
				Sans A-V		Avec A-V	
SAXPY	5e-06	1.74e-05	(3.5)	0.0004	(80)	0.00098	(196)
SGEMV	0.020	0.098	(4.9)	1.683	(84.15)	3.817	(190)
SGEMM	40.013	504.327	(12.6)	6974.15	(174)	11773.5	(294)
DAXPY	5e-06	1.9e-05	(3.8)	0.0004	(80)	0.0009	(196)
DGEMV	0.022	0.103	(4.68)	1.690	(84.15)	5.383	(244)
DGEMM	77.343	513.275	(6.64)	6956.39	(89.94)	22100.3	(285)

Ces tests montrent que l'implémentation directe de CADNA dans les routines BLAS engendre un important surcoût quelque soit le niveau visé. Les résultats soulignent également que ces surcoûts sont encore plus importants pour le niveau 3 des BLAS (voir *tableau* 5.5). Il n'est pas inutile de noter l'important surcoût dû à l'auto-validation CADNA. En effet, on constate un facteur 3 entre les versions avec ou sans auto-validation (cf. *tableau* 5.5). Près de $2/3$ du temps d'exécution y est donc consacré. L'utilisation des "templates" peut être considérée comme une solution viable techniquement pour associer l'arithmétique stochastique discrète aux routines d'algèbres linéaire, mais les surcoûts rendent cette solution rédhibitoire. L'utilisation des "templates" soulève aussi un autre point celui de son incompatibilité avec le langage Fortran, qui rappelons le, est l'un des langages le plus utilisé pour les codes de simulation numérique industriels. En effet, on peut facilement faire appel à une fonction écrite en "C" dans un code en Fortran sans que cette opération n'impacte considérablement les performances. Il est cependant difficile, avec les technologies actuelles, de faire appel à des fonctions "template" dans une application écrite en Fortran.

Figure 5.5 – Surcoût en temps de calcul dû à l'utilisation de CADNA dans les BLAS pour la routine xGEMM. Afin de mettre en évidence l'importance du surcoût, l'échelle semi-logarithmique n'est pas utilisée ici. La courbe LinalCdnAA représente la version Linalg intégrant les types stochastiques avec auto-validation et LinalCdnSA la version sans auto-validation.

(a) sgemm



(b) dgemm

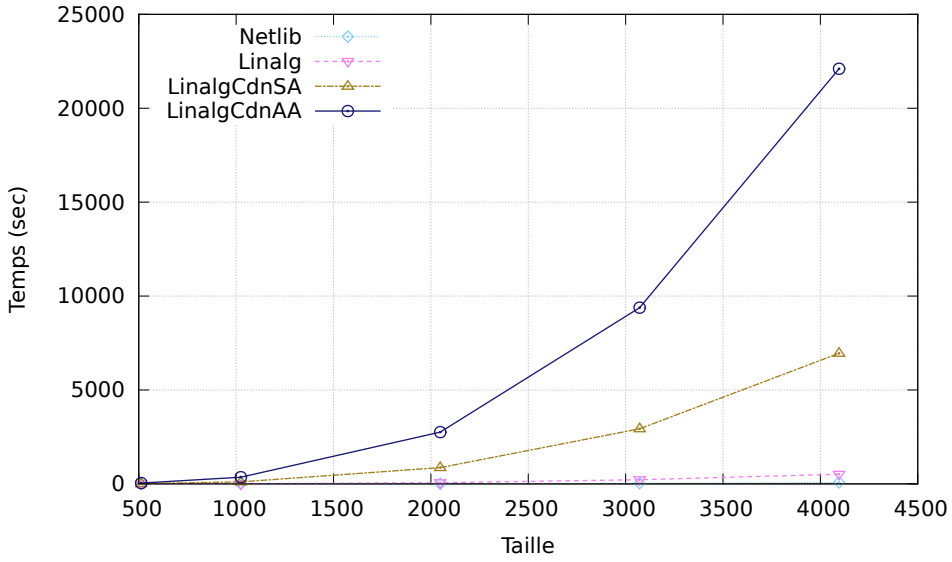
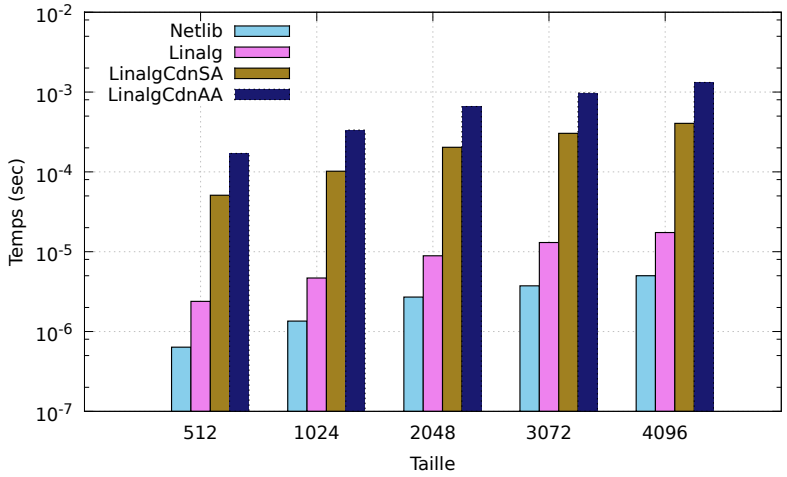
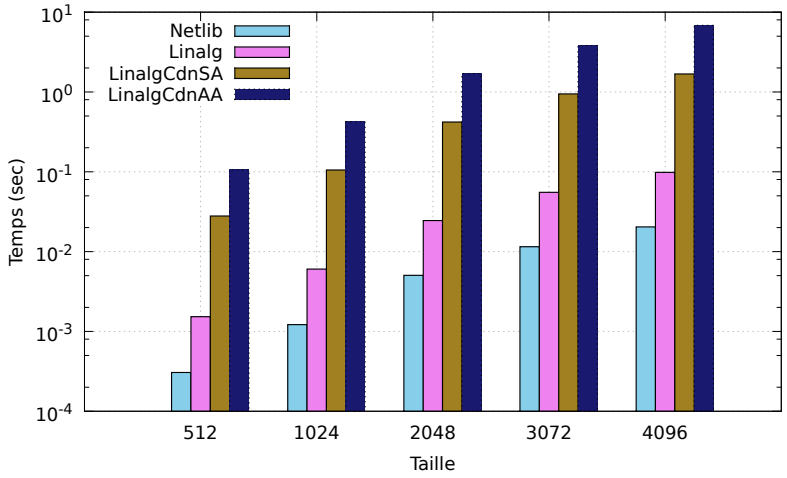


Figure 5.6 – Surcoût en temps de calcul dû à l'utilisation de CADNA dans les BLAS pour les routines AXPY , GEMV et GEMM en simple précision (échelle semi-logarithmique). La courbe LinalCdnAA représente la version Linalg intégrant les types stochastiques avec auto-validation et LinalCdnSA la version sans auto-validation.

(a) saxpy



(b) sgemv



(c) sgemm

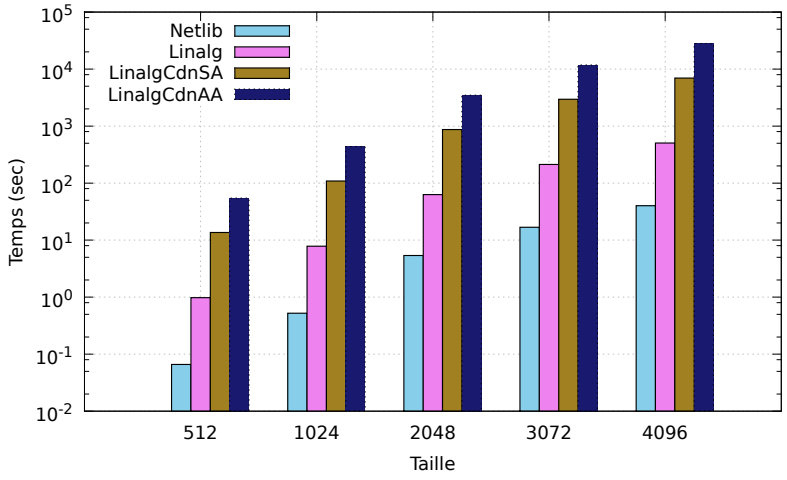
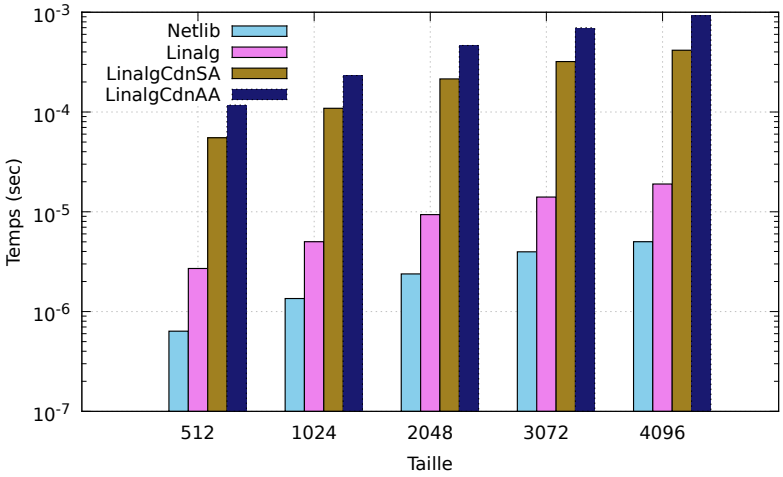
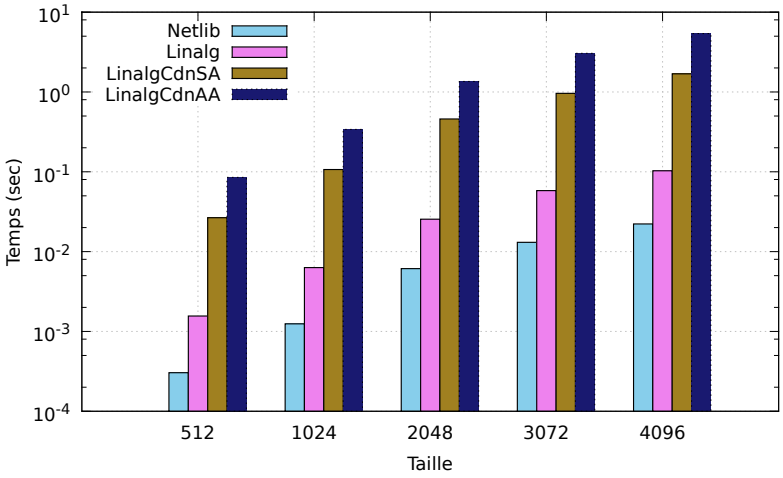


Figure 5.7 – Surcoût en temps de calcul dû à l'utilisation de CADNA dans les BLAS pour les routines AXPY , GEMV et GEMM en double précision (échelle semi-logarithmique). La courbe LinalCdnAA représente la version Linalg intégrant les types stochastiques avec auto-validation et LinalCdnSA la version sans auto-validation.

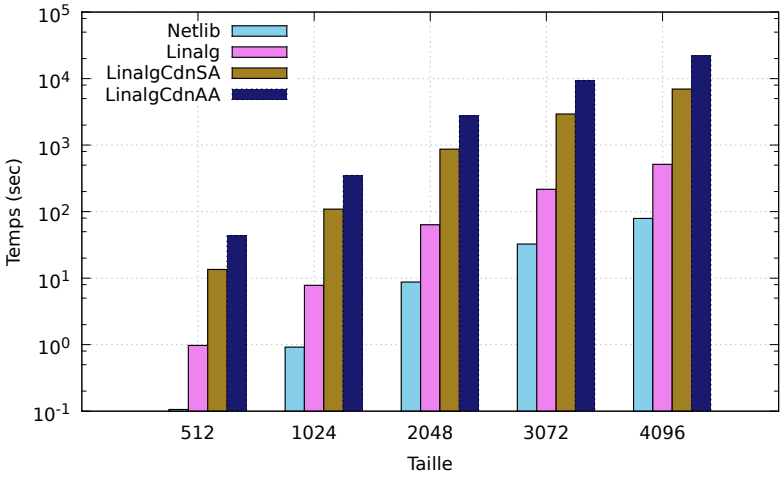
(a) daxpy



(b) dgemv



(c) dgemm



En résumé, nous partons du constat suivant : l'implémentation directe (ici avec les templates) de la DSA dans les routines BLAS engendre un surcoût important : environ un facteur 90 si on ne considère que les calculs, le facteur avoisinant 250 dès lors qu'on actionne la validation automatique de CADNA. Aussi, rappelons que la version Netlib que nous considérons comme référence ici est de loin la moins performante des implémentations BLAS (voir section 5.2.2). Dans la prochaine section, nous introduisons une version dite naïve de DgemmCadna. A partir de cette version et des résultats présentés dans les deux dernières sections, nous nous proposons de trouver les véritables raisons de ces surcoûts.

5.4 La routine DgemmCadna

5.4.1 DgemmCadnaV1

Un objectif majeur de notre travail est de développer une routine DGEMM compatible avec la bibliothèque CADNA : DgemmCadna. De manière générale, pour implémenter l'arithmétique stochastique discrète dans une fonction, il est juste nécessaire de modifier la définition de la fonction -le prototype- en remplaçant les types flottants par des types stochastiques (*double_st* ou *float_st*). Le prototype de la routine DgemmCadna ressemblerait alors à l'extrait de code 5.1.

Source 5.1 – Prototype de la routine DgemmCadna

```
void cblas_dgemm (
     const enum CBLAS_ORDER Order,
     const enum CBLAS_TRANSPOSE TransA,
     const enum CBLAS_TRANSPOSE TransB,
     const int M,
     const int N,
     const int K,
     const SCALAR alpha,
     const double_st * A,
     const int lda,
     const double_st * B,
     const int ldb,
     const SCALAR beta,
     double_st * C,
     const int ldc)
```

Nous avons choisi de commencer nos développements avec un produit matriciel basique formé trois boucles imbriquées (voir extrait de code 5.2). Cette première version que nous appelons DgemmCadnaV1, permettra d'évaluer le surcoût lié à l'utilisation des opérations surchargées CADNA et des changements de modes d'arrondi qui découlent de ces opérations.

Source 5.2 – Implémentation naïve du produit matriciel avec CADNA

```
int dgemmcadnaV1(int n, double_st alpha, double_st *A, double_st *B,double_st ↩
    beta,double_st *C)
{
  int i, j, k;
  for (i = 0; i < n; i++){
      for (j = 0; j < n; j++){
    C[i*n+j] = beta * C[i*n+j];//beta;
      }
      for (j = 0; j < n; j++){
    for (k = 0; k < n; k++){
        C[i*n+j] += alpha* A[i*n+k] * B[k*n+j] ;
```

```
      } /* for k */
       } /* for j */
  } /* for i */
  return 0;
}
```

5.4.2 L'influence de la méthode CESTAC

Afin d'évaluer l'influence de la méthode, la première version DgemmCadnaV1 a été comparée à une deuxième version DgemmCadnaV2 dans laquelle aucune surcharge CADNA n'est utilisée (aucun changement de mode d'arrondi également). Rappelons que dans l'implémentation de la méthode CESTAC, une opération d'arithmétique élémentaire est remplacée par 3 exécutions synchrones combinées avec l'arithmétique aléatoire. Dans DgemmCadnaV2, les types stochastiques sont mis à contribution mais l'arithmétique aléatoire n'est pas utilisée (voir extrait de code 5.3).

Source 5.3 – Implémentation naïve du produit matriciel avec CADNA sans changement du mode d'arrondi

```
int dgemmcadnaV2(int n, double_st alpha, double_st *A, double_st *B,
double_st beta, double_st *C)
{
 int i, j, k;
  for (i = 0; i < n; i++){
      for (j = 0; j < n; j++){
    C[i*n+j].x = beta.x * C[i*n+j].x; //beta;
    C[i*n+j].y = beta.y * C[i*n+j].y;
    C[i*n+j].z = beta.z * C[i*n+j].z;
    for (k = 0; k < n; k++){
        C[i*n+j].x += alpha.x* A[i*n+k].x * B[k*n+j].x ;
        C[i*n+j].y += alpha.y* A[i*n+k].y * B[k*n+j].y ;
        C[i*n+j].z += alpha.z* A[i*n+k].z * B[k*n+j].z ;
      } /* for k */
   } /* for j */
      } /* for i */
  return 0;
}
```

Comparer ces deux premières versions, permet une évaluation précise du coût des changements de mode d'arrondi. Nous avons trouvé pertinent de ne s'attarder que sur les versions ne faisant pas appel à l'auto-validation de CADNA puisque ce mode sert également à tracer les instabilités numériques. On observe une importante différence entre les temps d'exécution des deux versions sur la *figure* 5.8 et dans le *tableau* 5.6. Nous notons un rapport temps $V1/V2$ supérieur à 7. Cet important rapport ne peut s'expliquer que par les nombreux changements de mode d'arrondi. Plus de 85% du temps d'exécution de la V1 sont dus aux modes d'arrondi choisis aléatoirement (voir Figure 5.8). En effet, pour changer de mode d'arrondi, on fait appel à une fonction système. Cette dernière vide automatiquement le pipeline des instructions. Dans le pire des cas, le processeur n'exécutera qu'une seule opération à la fois. Dans ces conditions, il est impossible d'espérer des performances correctes avec notre routine DgemmCadna.

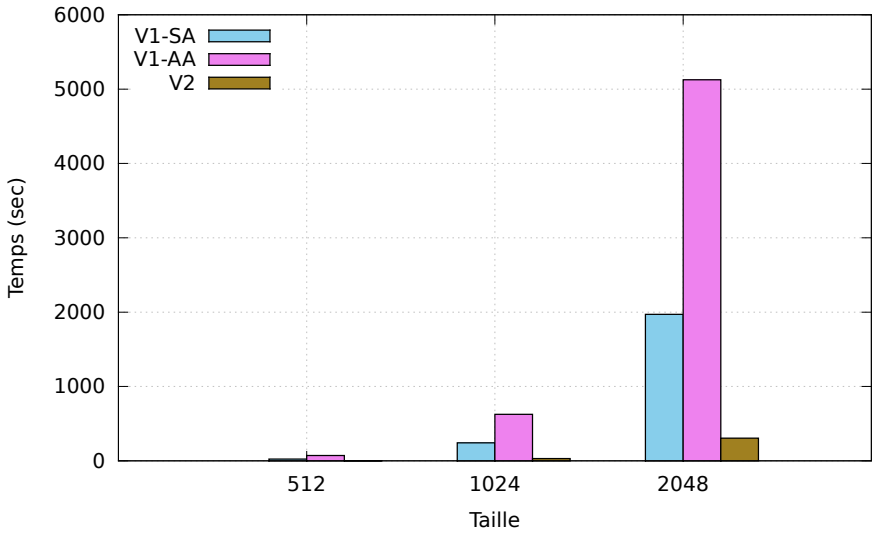
Nous nous devons donc de limiter l'impact de la méthode CESTAC. Dans cette optique, nous allons modifier légèrement la méthode. L'idée principale est de faire moins de changement d'arrondi et de profiter au maximum du parallélisme interne des processeurs. Nous

Tableau 5.6 – Évaluation du surcoût du à la méthode CESTAC

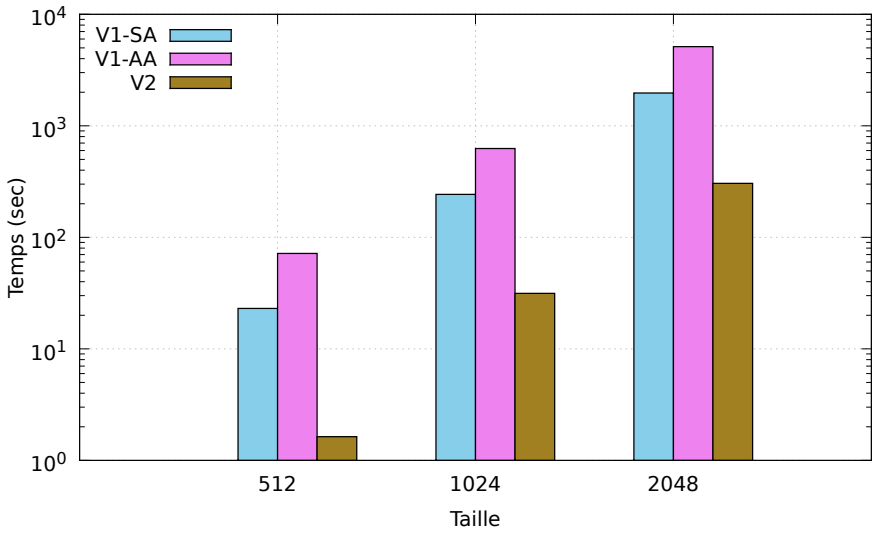
Taille	DgemmCadnaV1		DgemmCadnaV2
	Sans A-V	Avec A-V	
512	23.0184	71.7546	1.62926
1024	242.81	625.422	31.4558
2048	1971.23	5124.98	304.632

Figure 5.8 – Histogrammes pour l'évaluation du surcoût dû à la méthode CESTAC ; on remarquera qu'il a fallu passer en échelle semi-logarithmique pour mettre en évidence les divers rapports.

(a) échelle normale



(b) échelle semi-logarithmique



devons cependant veiller à ne pas dégrader les caractéristiques probabilistes de la méthode CESTAC. La nouvelle méthode est expliquée en détail dans la section section 5.6.

Le surcoût que nous avons évalué ici ne prend pas en compte le mode d'auto-validation de CADNA. Nous ne travaillerons pas sur cet aspect de notre routine, l'idée est de minimiser au maximum le temps de calcul. Dans la suite de ce chapitre, nous n'utilisons que le mode de fonctionnement de CADNA sans auto-validation.

L'optimisation de notre routine DgemmCadna ne consistera uniquement pas à limiter le surcoût CADNA. L'objectif est d'avoir une fonction qui ne dégradera pas considérablement les performances du code dans lequel elle sera utilisée. Aussi, des modifications d'ordre algorithmique doivent être apportées pour espérer de meilleures performances.

5.4.3 DgemmCadnaV1 et les différentes implémentations

Les principaux codes de calcul scientifique font appel aux différentes routines BLAS et particulièrement à xGEMM. Il est donc nécessaire de s'assurer que notre routine DgemmCadna ne détruira pas complètement les performances des codes qui lui feront appel. DgemmCadnaV1 est alors considérée comme la version de départ. Les développements serviront à améliorer le temps d'exécution de cette routine. Dans cette optique, plusieurs modifications seront apportées à l'implémentation du produit matriciel. Les temps d'exécution seront comparés aux différentes versions des BLAS (Netlib, ATLAS, MKL, et GotoBLAS) et à l'implémentation Linalg couplée à CADNA (LinalgCdn).

Le *tableau* 5.7 confirme les observations faites à la section 5.3. On note également un rapport supérieur à 1000 entre DgemmCadnaV1 et les versions GotoBLAS et MKL. Les raisons de ces importants rapports sont résumées ci-dessous.

- L'utilisation de CADNA : chaque opération arithmétique est faite 3 fois, on utilise 3 fois plus de nombres flottants et 4 fois plus de mémoire. Ceci étant, le véritable surcoût provient, comme expliqué dans la section 5.4.2, des changements de mode d'arrondi.
- Les 3 boucles internes de DgemmCadnaV1 : ces boucles entraînent un nombre important de défauts de cache. Rappelons que les autres versions de DGEMM ont été très optimisées et profitent entièrement des capacités de performance offertes par la machine.
- Une mauvaise utilisation de la mémoire : c'est une conséquence directe des trois boucles imbriquées. Celles-ci provoquent continuellement des défauts de cache et de TLB (*The Translation Look-aside Buffer*, voir section 5.5.1). L'utilisation des types stochastiques soulève aussi des problèmes de saut en mémoire. En effet, les types stochastiques sont des structures non homogènes composées d'un entier et de trois flottants.

Divers travaux ont étés réalisés ces dernières années afin d'optimiser les routines BLAS. Ces travaux se basent notamment sur une meilleure exploitation des caractéristiques des nouvelles architectures. Dans la prochaine section, nous expliquons comment l'accès à la mémoire

Tableau 5.7 – Comparaison des temps d'exécution de DgemmCadnaV1 et des versions référence.

Taille	DgemmCadnaV1	LinalgCdn	Netlib	Goto	MKL
				Séquentiel	
512	23.0184	13.5067	0.106008	0.026967	0.0268694
1024	242.81	109.074	0.87176	0.214558	0.214428
2048	1971.23	869.126	8.69573	1.70324	1.70753

peut être le principal frein à l'obtention d'une routine performante. Les solutions choisies pour optimiser DgemmCadna sont ensuite exposées.

5.5 Vers une optimisation de DgemmCadna

La multiplication de matrices est l'une des opérations les plus populaires de l'algèbre linéaire. Elle est d'autant plus importante qu'elle est l'opération principale (le noyau) de plusieurs algorithmes tels que la résolution de systèmes linéaires, les calculs des valeurs propres et le problème des moindres carrés. Un passage en revue des principaux outils d'algèbre linéaire montre qu'ils sont, pour la plus part, basés sur les algorithmes par bloc afin d'exploiter au mieux les mémoires hiérarchiques [Kurzak *et al.*, 2009]. Parmi les logiciels libres, LAPACK [Anderson *et al.*, 1992, Anderson *et al.*, 1999] et ScaLAPACK [Blackford *et al.*, 1997] en sont la preuve concrète. Ces dernières décomposent la matrice en blocs de sous-matrices carrées ou rectangulaires. Les boucles internes (niveau le plus bas) de ces implémentations portent sur des opérations sur des sous-matrices ; ces opérations se font grâce aux routines BLAS et surtout à xGEMM. De manière générale, toutes les opérations du niveau 3 des BLAS peuvent être décomposées en plusieurs xGEMM et quelques routines du niveau 1 et 2. Partant de ce constat, il est alors indispensable d'avoir une routine xGEMM très performante.

Implémenter un produit matriciel ayant des performances optimales est un problème très complexe. Citons à ce sujet Goto [Goto et van de Geijn, 2008b] :

> *« Implementing matrix multiplication so that near-optimal performance is attained requires a thorough understanding of how the operation must be layered at the macro level in combination with careful engineering of high-performance kernels at the micro level. »*

La taille des blocs est alors le paramètre le plus important. L'utilisation des blocs permet une meilleure utilisation de la mémoire. Ainsi, on exploite mieux les caractéristiques des machines contemporaines basées sur le principe de la mémoire partagée avec un comportement NUMA (Non Uniform Memory Acces). Ces machines sont composées de plusieurs processeurs, qui eux-mêmes renferment plusieurs cœurs. Chacun d'eux est associé à une unité mémoire interconnectée par un système de cohérence de cache leur donnant accès à l'intégralité de la mémoire de manière transparente [Drepper, 2007, Faverge, 2009]. Ces types d'architectures ont donc une structure très hiérarchisée dont les coûts d'accès mémoire varient en fonction de la zone mémoire concernée. Il faut alors choisir la taille de bloc afin que toutes les sous-matrices concernées par le calcul puissent tenir dans la zone mémoire ciblée. Généralement, le noyau du calcul (produit de deux sous-matrices) s'effectue sur des données présentes dans le cache L1 (la zone la plus performante en accès mémoire).

De nombreux travaux ont été consacrés à l'optimisation du produit matriciel sur diverses architectures matérielles (CPU, GPU, Processeur CELL [Bourgerie *et al.*, 2010]). Ces travaux ont donné lieu à plusieurs publications et à diverses implémentations (voir section 5.2.1). La tendance émergente en algèbre linéaire est l'utilisation de structures de données spécialisées telles que le *Block Data Layout* (BDL) [Park *et al.*, 2003] et l'expression d'algorithmes directement en termes de noyau de calcul dit "*kernel*" [Kurzak *et al.*, 2009]. Récemment, les auteurs de [Gepner *et al.*, 2012] ont montré qu'on obtenait les meilleures performances pour le DGEMM avec une implémentation hybride basée d'une part sur l'algorithme récursif de Strassen pour le découpage des blocs couplé avec l'utilisation d'Intel MKL pour les produits de matrices des niveaux les plus bas (kernels) et d'autre part sur l'utilisation des instructions vectorielles AVX disponibles sur les dernières architectures d'Intel. On peut également citer les récents travaux

présentés dans [Van Zee *et al.*, 2013] basés eux sur la bibliothèque *BLIS : A Framework for Generating BLAS-like Libraries* [Van Zee et van de Geijn, 2012]. Ces travaux montrent qu'avec la bibliothèque BLIS et très peu d'effort, il est très facile d'obtenir des performances qui peuvent rivaliser avec les meilleures implémentations actuellement disponibles pour les routines BLAS 3. En fait, BLIS est en quelque sorte une "ré-implémentation" de GotoBLAS qui, à l'inverse de la version originale, est plus lisible. Ce qui permet une meilleure "ré-utilisabilité" et un portage facile vers de nouvelles architectures. BLIS repose sur des micro-kernels qui sont des découpages des kernels de Goto. Le code est alors plus compréhensible. Ainsi, il suffit tout simplement de travailler sur les micro-kernels pour optimiser ou porter la bibliothèque sur une nouvelle architecture. Rappelons que le projet GotoBLAS n'est plus maintenu et qu'il a été remplacé par le projet OpenBLAS [Xianyi *et al.*, 2012a] dont l'objectif est de porter les codes sources sur les nouvelles architectures [Xianyi *et al.*, 2012b]. Par ailleurs, d'autres travaux sont menés pour obtenir des performances raisonnables quelque soit l'architecture matériel (CPU, GPU, etc) [Garg et Hendren, 2013] ou pour minimiser les communications dans les calculs [Lipshitz, 2013]. Enfin, certaines études [Revol et Théveny, 2013, Ozaki *et al.*, 2012a, Ozaki *et al.*, 2012b, Li *et al.*, 2002] ont été réalisées pour obtenir un produit de matrice rapide et précis. Ces travaux sont basés sur les méthodes permettant d'améliorer la précision d'un calcul (arithmétique d'intervalles, arithmétique multiprécision et algorithmes compensés).

Notre problématique est légèrement différente de celle des travaux que nous venons de mentionner. Nous évaluons la précision des résultats alors que l'objectif des travaux [Revol et Théveny, 2013, Ozaki *et al.*, 2012b] est d'augmenter la précision avec la multiprécision et les algorithmes compensés. Nous avons pour objectif d'optimiser un produit matriciel pour une structure de donnée particulière (les types stochastiques). Ces types de données ne sont pas homogènes : ils sont composés soit de trois *doubles* et deux *entiers*, soit de trois *float* et un *entier*. Nous devons également prendre en compte l'arithmétique stochastique discrète qui définit les opérations mathématiques élémentaires pour ces types. Nous nous sommes inspiré, dans un premier temps, des travaux sur l'optimisation du DGEMM et avons travaillé dans une deuxième étape sur la minimisation du surcoût lié à l'utilisation de la méthode CESTAC.

Nous nous sommes largement inspiré des travaux présentés dans [Clint Whaley *et al.*, 2001, Goto et van de Geijn, 2008a, Goto et van de Geijn, 2008b, Gottschling *et al.*, 2009, Kurzak *et al.*, 2009, Stewart, 1998]. Nous remarquons que le leitmotiv de toutes ces implémentations était de ***réduire le nombre et le coût des accès mémoire***. La démarche à suivre pour l'optimisation d'un produit matriciel peut être résumée en cinq principales étapes [Kurzak *et al.*, 2009] :

1. Algorithme classique : on considère une implémentation naïve du produit matriciel avec trois boucles (voir extrait de code 5.2).
2. Utilisation des blocs : on utilise un algorithme par blocs (tiling). La taille du bloc est un paramètre à déterminer. Il dépend des caractéristiques de la mémoire de notre machine cible.
3. Déroulement des boucles internes : le passage à un algorithme par blocs introduit 3 boucles internes qui servent à calculer des produits de sous-matrices de A et B et met à jour une sous-matrice de C avec les résultats partiels. L'objectif ici est de dérouler (sans l'aide du compilateur) complètement ces trois boucles pour obtenir un seul morceau de code qui constituera notre noyau de calcul.
4. Déroulement des boucles externes : l'objectif est de parvenir à une seule boucle afin de minimiser le coût des boucles. Cependant, cette étape est très délicate. En effet, un déroulement agressif peut nuire aux performances de notre routine.
5. Optimiser le noyau de calcul.

5.5.1 L'accès mémoire sur les nouvelles architectures

Afin de concevoir des microprocesseurs de plus en plus performants, les constructeurs ont introduit du parallélisme au sein même des microprocesseurs : ce qui a donné naissance aux processeurs multi-cœurs. La multiplication du nombre des processeurs et aujourd'hui la multiplication du nombre de cœurs de calcul dans les processeurs a introduit une sophistication des puces pour que ces cœurs puissent échanger efficacement et accéder rapidement à la mémoire. Cette sophistication a apporté une topologie fortement hiérarchique dans les nouvelles architectures utilisant ces microprocesseurs et des temps d'accès à la mémoire non-uniformes [Drepper, 2007]. On parle d'architectures NUMA (Non-Uniform Memory Access).

Chaque cœur est associé à une unité mémoire et il est connecté aux autres cœurs par un système de cohérence de cache leur donnant accès à l'intégralité de la mémoire de manière transparente [Drepper, 2007,Faverge, 2009]. Ce type d'architecture implique alors une structure très hiérarchisée dont les coûts d'accès à la mémoire varient grandement d'une unité mémoire à une autre [Antony *et al.*, 2006]. De plus, la bande passante est également variable à cause de l'entrelacement des accès qui peuvent se perturber entre eux. Ces architectures imposent de prendre en compte le placement des données lors de leur allocation et de leur utilisation pour réduire au maximum la distance unité de calcul/mémoire ainsi que les échanges de données entre les chipsets.

Pour accéder à une donnée, un processeur communique avec ses mémoires caches (L1, L2, L3) qui communiquent elles, avec la mémoire principale de la machine. Si la donnée recherchée n'est pas disponible dans le cache, on obtient un *défaut de cache*. C'est un phénomène qui entraîne l'arrêt des activités du processeurs jusqu'à ce que la donnée recherchée soit rangée dans la mémoire cache (copie de la mémoire principale vers le cache). Accéder à la mémoire principale peut être dix fois plus lent qu'accéder à la mémoire cache. Les défauts de cache peuvent dégrader considérablement les performances d'un code de calcul. Il est alors important de minimiser le nombre de défauts de cache.

Pour ranger les données de la mémoire principale vers le cache, les adresses virtuelles des données doivent être converties en adresse physique. La mémoire physique est organisée en pages de tailles égales. La mémoire virtuelle est aussi organisée en pages de même taille. Chaque page virtuelle a un numéro précis qui l'identifie. Pour que le processeur puisse avoir accès à une donnée d'une page virtuelle, il faut que l'adresse de cette page soit convertie en adresse de page physique. Pour éviter une conversion à chaque accès à une nouvelle page, il existe une table (de correspondance) qui contient toutes les conversions. Cette table est stockée sur la mémoire principale machine, ceci rajoute alors un coût à la conversion mémoire virtuelle/mémoire physique. Pour éviter ce surcoût, une plus petite table, *The Translation Look-aside Buffer* (TLB), sauvegarde les adresses des dernières tables utilisées. A chaque fois que le processeur trouve l'adresse d'une table virtuelle dans le TLB la conversion est rapide. Dans les cas où le processeur ne le trouve pas dans le TLB, il se passe un *défaut de TLB*. Le *translation Look-aside Buffer* est en quelque sorte une mémoire cache du processeur. La plus grande différence entre le défaut de TLB et le défaut de cache est qu'un défaut de cache ne va pas forcément suspendre le CPU. Goto expliquait les conséquences des défauts de TLB dans [Goto et van de Geijn, 2008b] en ces termes :

> *« The most significant difference between a cache miss and a TLB miss is that a cache miss does not necessarily stall the CPU. A small number of cache misses can be tolerated by using algorithmic prefetching techniques as long as the data can be read fast enough from the memory where it does exist and arrives at the CPU by the time it is needed for computation.*

A TLB miss, by contrast, causes the CPU to stall until the TLB has been updated with the new address. In other words, prefetching can mask a cache miss but not a TLB miss. »

Il est alors nécessaire d'adapter les algorithmes pour que ceux-ci travaillent au maximum sur des données en cache.

5.5.2 Algorithmes par blocs (*tiling*)

5.5.2.1 Un algorithme itératif

L'idée de base est qu'un produit matriciel peut être décomposé en plusieurs produits de sous-matrices. Soit A, B, C trois matrices de taille (n, n). Ces matrices peuvent être décomposées en blocs comme ci-dessous :

$$\begin{bmatrix} C_{11} & C_{12} & \dots & C_{1N} \\ C_{21} & C_{22} & \dots & C_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ C_{N1} & C_{N2} & \dots & C_{NN} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} & \dots & A_{1N} \\ A_{21} & A_{22} & \dots & A_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ A_{N1} & A_{N2} & \dots & A_{NN} \end{bmatrix} \times \begin{bmatrix} B_{11} & B_{12} & \dots & B_{1N} \\ B_{21} & B_{22} & \dots & B_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ B_{N1} & B_{N2} & \dots & B_{NN} \end{bmatrix}$$

où chaque bloc C_{ij} est calculé avec la formule suivante

$$C_{ij} = \sum_{k=1}^{N} A_{ik} B_{kj} \tag{5.1}$$

L'algorithme du produit matriciel devient alors :

Source 5.4 – Algorithme produit matriciel par blocs

```
int SIZEBLOCK = ....;  // A determiner
int i,j,k,ii,jj,kk ;

for (i = 0; i < n/SIZEBLOCK; i++)
 for (j = 0; j < n/SIZEBLOCK; j++)
  for (k = 0; k < n/SIZEBLOCK; k++)

   for (ii = 0; ii < SIZEBLOCK; ii++)
    for (jj = 0; jj < SIZEBLOCK; jj++)
     for (kk = 0; kk < SIZEBLOCK; kk++)

        C[(i*n + j)*SIZEBLOCK +ii*n + jj] += A[(i*n+k)*SIZEBLOCK + ii*n ↩
            +kk] + B[(k*n+j)*SIZEBLOCK + kk*n + jj] ;
```

L'utilisation des blocs est une technique d'optimisation de l'utilisation mémoire très connue. L'objectif est de minimiser les défauts de TLB et de cache en travaillant sur des données déjà présentes en mémoire cache. On suppose qu'en travaillant sur des blocs, le processeur ne chargera en mémoire qu'une partie de la matrice. L'idée est qu'à chaque fois qu'on calcule un résultat partiel de la matrice C (voir équation 5.1) les trois blocs C_{ij}, A_{ik} et B_{kj} soient présents en mémoire cache. En fait, on exploite ici la *localité temporelle des données*.

Le paramètre le plus important est donc la taille des blocs. Sur nos nouvelles machines, la mémoire cache est subdivisée en 3 niveaux : niveau L1, niveau L2 et niveau L3. Les différences entre les trois niveaux de cache et les différentes interactions avec le CPU sont expliquées dans [Drepper, 2007]. Retenons que le niveau L1 possède la taille plus petite et la moins coûteuse en accès mémoire.

Tableau 5.8 – Comparaison de DgemmCADNA par blocs de 4, 8, 16 et 32.

Taille	B4	B8	B16	B32
512	20.5852	22.33	20.5841	20.6072
1024	177.25	183.623	177.124	177.847
1536	598.323	624.895	596.826	598.181
2048	1420.34	1477.82	1418.65	1422.21
4096	11472.3	12215.1	11407.4	11512.7

Dans notre cas, nous souhaitons que 3 blocs de matrices stochastiques puissent tenir dans le cache L1 qui a une capacité mémoire de 64 Kio. La moitié de la capacité du cache L1 (soit 32 Kio) est réservée aux instructions machine [Intel, 2012]. Soit $SIZEBLOCK$ la taille du bloc, $SizeDst$ la taille d'un double_st et $SizeL1$ la taille disponible du cache L1, il faut que :

$$3 \times SIZEBLOCK \times SIZEBLOCK \times SizeDst < SizeL1/2 \tag{5.2}$$

Cette relation est vérifiée pour $SIZEBLOCK = 18$. Pour un souci d'alignement mémoire, il est préférable de travailler avec un bloc de taille 16 (voir *tableau* 5.8). Le tableau 5.9 montre les résultats comparatifs de la nouvelle version B16 avec la première version DgemmCadnaV1 et Linalg. B16 est légèrement plus performant que DgemmCadnaV1 mais moins que Linalg. Par la suite, nous appelons la version B16 : DGB16 pour DGemm par Blocs de 16.

Tableau 5.9 – Comparaison DgemmCadnaV1 à DGB16 (DgemmCADNA par blocs de 16) : GainV1B16 représente le rapport temps d'exécution DgemmCadnaV1/DGB16.

Taille	DgemmCadnaV1	DGB16	GainV1B16
512	23.0184	22.6516	1,016
1024	242.81	183.461	1,32
2048	1971.23	1468.2	1,34

5.5.2.2 Un algorithme utilisant un découpage récursif

Un moyen très simple pour exploiter la localité des données est la récursivité. De surcroît, elle permet d'utiliser les différents niveaux de caches. L'idée serait alors de diviser successivement la matrice en quatre sous-matrices à chaque étape de la récursivité (voir équation 5.3 et équation 5.4) et de recommencer jusqu'à ce qu'on obtienne des blocs pouvant tenir dans le cache L1.

$$\left[\begin{array}{c|c} C_{00} & C_{01} \\ \hline C_{10} & C_{11} \end{array}\right] = \left[\begin{array}{c|c} A_{00} & A_{01} \\ \hline A_{10} & A_{11} \end{array}\right] \times \left[\begin{array}{c|c} B_{00} & B_{01} \\ \hline B_{10} & B_{11} \end{array}\right] \tag{5.3}$$

$$\left[\begin{array}{c|c} \begin{array}{c|c} & \\ \hline & \end{array} & \\ \hline & \end{array}\right] \tag{5.4}$$

A chaque niveau de récursivité, les résultats des blocs de C sont calculés comme indiqué ci-dessous :

$$\begin{aligned} C_{00} &= A_{00} \times B_{00} \\ C_{00} &= A_{01} \times B_{10} \\ \\ C_{01} &= A_{00} \times B_{01} \\ C_{01} &= A_{01} \times B_{11} \\ \\ C_{10} &= A_{10} \times B_{00} \\ C_{10} &= A_{11} \times B_{10} \\ \\ C_{11} &= A_{10} \times B_{01} \\ C_{11} &= A_{11} \times B_{11} \end{aligned} \tag{5.5}$$

Une version de l'algorithme récursif appelée DGBR16 (DGemm par Blocs Récursif de taille 16) avec une taille de bloc égale à 16 a été implémentée. Le *tableau* 5.10 présente les résultats de cette version. La comparaison avec les gains obtenus *DGB16* (voir *tableau* 5.9) montre une légère amélioration : on constate une amélioration de 54% par rapport à *DGB16*.

Tableau 5.10 – Comparaison DgemmCadnaV1 à DGBR16 : GainBR16 représente le rapport temps d'exécution DGBR16/DgemmCadnaV1.

Taille	DgemmCadnaV1	DGBR16	GainBR16
512	23.0184	14.7567	1,62
1024	242.81	119.129	2,04
2048	1971.23	951.174	2.07

5.5.2.3 Un algorithme itératif adapté aux trois niveaux de cache

Dans l'optique de mieux exploiter la localité des données, un algorithme itératif par blocs sur trois niveaux (étages) a été implémenté. Cette version est appelée DGBI (DGemm par Blocs Itératif de taille 16 avec découpage récursif). L'idée ici est de profiter des apports du découpage récursif mais seulement sur trois niveaux. Chaque niveau de découpage se fait en fonction de la mémoire cache ciblée. Les trois niveaux sont présentés ci-dessous :

1. Premier niveau de partitionnement pour le Cache L3 : objectif = avoir 3 blocs A_{ik}, B_{kj} et C_{ij} pouvant tenir dans le cache L3.

$$A(n*n) = \begin{bmatrix} A_{11} & A_{12} & \dots & A_{1N} \\ A_{21} & A_{22} & \dots & A_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ A_{N1} & A_{N2} & \dots & A_{NN} \end{bmatrix}$$

2. Deuxième niveau de partitionnement pour le Cache L2 : objectif = avoir 3 blocs pouvant tenir dans le cache L2.

$$A_{i,j} = \begin{bmatrix} AA_{11} & AA_{12} & \dots & AA_{1K} \\ AA_{21} & AA_{22} & \dots & AA_{2K} \\ \vdots & \vdots & \ddots & \vdots \\ AA_{K1} & AA_{K2} & \dots & AA_{KK} \end{bmatrix}$$

3. Troisième niveau de partitionnement pour le Cache L1 : objectif = avoir 3 blocs pouvant tenir dans le cache L1.

$$AA_{ii,jj} = \begin{bmatrix} AAA_{11} & AAA_{12} & \dots & AAA_{1P} \\ AAA_{21} & AAA_{22} & \dots & AAA_{2P} \\ \vdots & \vdots & \ddots & \vdots \\ AAA_{P1} & AAA_{P2} & \dots & AAA_{PP} \end{bmatrix}$$

Les tailles des blocs pour chaque niveau sont déterminées à partir des caractéristiques de notre machine (voir section A.2). L'objectif est que les données traitées au niveau L1 puissent profiter de la localité des données présentes dans le cache L2 et que celles traitées au niveau L2 puissent profiter du L3. Nous avons des blocs de taille 256 pour le premier niveau (L3), 32 pour le niveau (L2) et 16 pour le dernier niveau (L1) ; ces tailles ont été calculées à partir de l'équation 5.2. Les résultats de DGBI16 sont présentés dans le *tableau* 5.11.

Tableau 5.11 – Comparaison DgemmCadnaV1 à DGBI16 ; GainBI16 représente le rapport temps d'exécution DgemmCadnaV1/DGBI16.

Taille	DgemmCadnaV1	DGBI16	GainBI16
512	23.0184	13.7696	1.67
1024	242.81	113.644	2.13
2048	1971.23	911.48	2.16

Tableau 5.12 – Comparaison des différentes versions : DgemmCadnaV1, Linalg, DGB16, DGBR16, DGBI16.

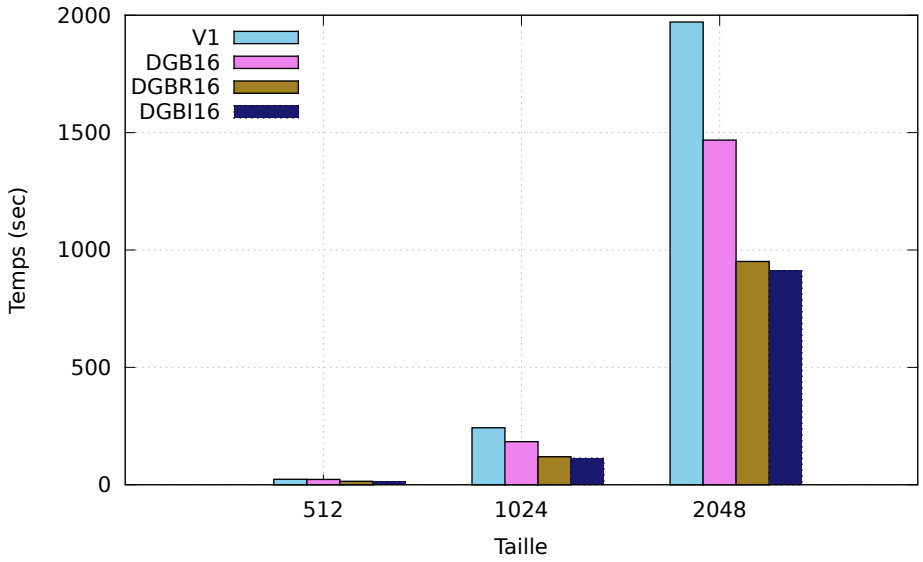
Taille	DgemmCadnaV1	LinalgCdn	DGB16	DGBR16	DGBI16
512	23.0184	13.5067	22.6516	14.7567	13.7696
1024	242.81	109.074	183.461	119.129	113.644
2048	1971.23	869.126	1468.2	951.174	911.48

Un comparatif entre les trois dernières implémentations est présenté dans *tableau* 5.12 et la *figure* 5.9. Les résultats des tableaux 5.9, 5.10, 5.11 et 5.12 montrent que l'utilisation des blocs est une solution envisageable (ou un début de solution) pour améliorer les performances de notre routine DgemmCadna. Néanmoins, on ne peut en aucun cas se contenter de ces faibles gains. Dans le meilleur des cas (DGBI16), on est 1,5 fois plus performant que la première version. Dans la prochaine section, nous mettons en œuvre des solutions pour une meilleure exploitation de la mémoire.

5.5.3 Vers une meilleure utilisation de la mémoire : *Block Data Layout* (BDL)

Nous avons montré dans la section précédente que l'utilisation des blocs permettait d'exploiter au maximum la localité temporaire des données pour une taille de cache donnée. Ces

Figure 5.9 – Comparaison DgemmCadnaV1 à DGB16, DGBR16 et DGBI16



algorithmes ont principalement pour objectif de minimiser les défauts de cache en diminuant la taille des matrices sur lesquelles on travaille simultanément. Il existe aussi d'autres techniques pour réduire les défauts de cache (utilisation des *padding*). Cependant, ces différentes techniques n'ont pas énormément d'influence sur les performances des TLB.

Dès que les tailles des matrices deviennent plus grandes, les performances du TLB deviennent plus importantes. Plus il y a de défaut de TLB, plus la performance globale sera considérablement dégradée [Goto et van de Geijn, 2008a]. Par conséquent, pour optimiser une application donnée, nous devrons considérer les défauts de cache et de TLB. Pour cela, il a été proposé dans [Park *et al.*, 2003] de modifier les modèles de stockage des matrices et le sens des boucles du produit matriciel.

De manière générale, on stocke en mémoire une matrice par un tableau 1D suivant deux formats classiques :

– *Column Major*

$$M(3*3) = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \quad \Rightarrow \quad M(3*3) = \begin{bmatrix} 1 & 4 & 7 & 2 & 5 & 8 & 3 & 6 & 9 \end{bmatrix}$$

– *Row Major*

$$M(3*3) = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \quad \Rightarrow \quad M(3*3) = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \end{bmatrix}$$

Supposons que lorsqu'on effectue un produit matriciel, on parcourt la matrice A ligne par ligne et la matrice B colonne après colonne. Si on utilise des blocs, on est alors obligé de faire de grands sauts en mémoire pour passer d'une ligne à une autre ou d'une colonne à une autre. On aura le même souci pour passer d'un bloc à un autre. Les sauts en mémoire auront pour conséquence des défauts de cache et TLB.

Les BDL (Block Data Layout) définissent des modèles de structures (stockage) de données. Avec les BDL, une grande matrice est subdivisée en plusieurs sous-matrices. Les éléments de ces sous-matrices sont stockés en mémoire de manière contiguës. Les différentes sous-matrices sont elles stockées suivant le format row-major.

Soit la matrice $A(n \times n)$ subdivisée en $N \times N$ sous-matrices A_{ij} de $p \times p$ éléments :

$$A(n \times n) = \begin{bmatrix} A_{11} & A_{12} & \dots & A_{1N} \\ A_{21} & A_{22} & \dots & A_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ A_{N1} & A_{N2} & \dots & A_{NN} \end{bmatrix} \qquad A_{ij}(p \times p) = \begin{bmatrix} 11 & 12 & \dots & 1p \\ 21 & 22 & \dots & 2p \\ \vdots & \vdots & \ddots & \vdots \\ p1 & p2 & \dots & pp \end{bmatrix}$$

Les éléments de chaque sous-matrices A_{ij} sont sauvegardés de manière contiguë :

$$A_{ij} = \begin{bmatrix} 11 & 12 & \dots & 1p & 21 & 22 & \dots & 2p & \dots\dots & p1 & p2 & \dots & pp \end{bmatrix}$$

En mémoire les sous-matrices sont stockées en *row-major* :

$$A(n \times n) = \begin{bmatrix} A_{11} & A_{12} & \dots & A_{1N} & A_{21} & A_{22} & \dots & A_{2N} & \dots\dots & A_{N1} & A_{N2} & \dots & A_{NN} \end{bmatrix}$$

Ce modèle de stockage est nommé *Block Row Major Layout (BRML)*. Il existe aussi un autre modèle de donnée Morton Data Layout [Gottschling *et al.*, 2009]. Dans ce format, on subdivise la matrice originale en 4 sous-matrices qui sont stockées en mémoire de manière contiguë. L'opération est répétée récursivement sur les sous-matrices. Au dernier niveau de récursivité, les éléments de la sous-matrice sont stockés de manière contiguë également.

L'utilisation des BDL améliore considérablement les performances du TLB et minimise les défauts de cache sur les machines à mémoire hiérarchique [Park *et al.*, 2003]. L'utilisation du BDL implique une recopie et une réorganisation des matrices dans notre cas. La matrice A qui est parcourue ligne par ligne est stockée suivant le format BRML. Nous avons modifié légèrement le format BRML pour la matrice B. Comme elle est parcourue colonne par colonne, les éléments des blocs sont stockés en *column-major*.

$$A(4 \times 4) = \left[\begin{array}{cc|cc} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ \hline 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{array}\right] \quad \Rightarrow \quad A'(4 \times 4) = [1\,2\,5\,6\,3\,4\,7\,8\,9\,10\,13\,14\,11\,12\,15\,16]$$

$$B(4 \times 4) = \left[\begin{array}{cc|cc} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ \hline 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{array}\right] \quad \Rightarrow \quad B'(4 \times 4) = [1\,5\,2\,6\,3\,7\,4\,8\,9\,13\,10\,14\,11\,15\,12\,16]$$

La seconde idée proposée par [Park *et al.*, 2003], est d'améliorer le sens des boucles du produit matriciel. En effet, ces boucles définissent le mode de parcours des matrices. Avec un ordre de parcours bien réfléchi en accord avec le modèle de stockage, on diminue le nombre d'accès mémoire en lecture (accès aux éléments de A ou B). Par contre, le nombre d'accès en écriture (accès aux élément de C) est constant.

Tableau 5.13 – Comparaison DgemmCadnaV1 à différentes implémentations de DGBRLMn (DGemm utilisant le mode de stockage BRML avec des sous-matrices de taille n)

Taille	DgemmCadnaV1	DGBRML4	DGBRML16
512	23.0184	13.7079	13.7022
1024	242.81	109.461	109.664
2048	1971.23	860.369	871.953

Tableau 5.14 – Tableau récapitulatif des différentes versions de DgemmCadna. GainBRML4 est le rapport entre les temps d'exécution de DgemmCadnaV1 et DGBRML4.

Taille	DgemmCadnaV1	DGB16	DGBR16	DGBI16	DGBRML4	GainBRML4
512	23.0184	22.6516	14.7567	13.7696	13.7079	1.67
1024	242.81	183.461	119.129	113.644	109.461	2.21
2048	1971.23	1468.2	951.174	911.48	860.369	2.29

Source 5.5 – Boucles internes

```
for(int i = 0; i < nb_block; i++){
    for(int k = 0; k < nb_block; k++){
      for(int j = 0; j < nb_block; j++){
         Cij += Aik * Bkj //noyau de calcul
```

Le noyau de calcul a également été optimisé. Les boucles internes ont disparu. Elle ont été déroulées à la main à l'aide de l'arithmétique des pointeurs [Kurzak *et al.*, 2009]. Avec ce modèle de stockage et ce sens de boucle, la matrice A est parcourue une seule fois et dans le sens où les éléments ont été stockés . La matrice B sera parcourue plusieurs fois mais dans le sens où les éléments ont été stockés. On parle alors de *data reuse*. En effet, à chaque fois qu'on a accès à un vecteur de la matrice A, on effectue en même temps toutes les opérations dans lesquelles ce vecteur est impliqué.

Par exemple, quand on déroule les boucles de l'extrait de code 5.5 $nb\_block = 2$, on obtient ceci :

$$
\begin{aligned}
C_{00} &= A_{00} \times B_{00} \\
C_{01} &= A_{00} \times B_{01} \\
C_{00} &= A_{01} \times B_{10} \\
C_{01} &= A_{01} \times B_{11} \\
C_{10} &= A_{10} \times B_{00} \\
C_{11} &= A_{10} \times B_{01} \\
C_{10} &= A_{11} \times B_{10} \\
C_{11} &= A_{11} \times B_{11}
\end{aligned}
$$

On remarquera que le bloc A_{ij} est utilisé pour deux itérations successives.

Les résultats de cette dernière version DGBRMLn (DGemm utilisant le mode de stockage BRML avec des sous-blocs de taille n) sont présentés dans le *tableau* 5.13. Les résultats des deux tableaux 5.13 et 5.14 sont très encourageants : DGBRML4 et DGBRML16 sont les plus performantes. Malgré la copie et la réorganisation de nos matrices, on obtient de meilleurs résultats que les algorithmes par blocs classiques. Ces résultats mettent en évidence l'importance du modèle de stockage des données. Toutefois, notons que malgré tous les efforts entrepris, nous obtenons des gains que nous jugeons moyens (2.27 dans le meilleur cas). Ce constat peut être

justifié par l'utilisation des structures de données stochastiques. Ces structures ne sont pas homogènes ; il est donc difficile de profiter réellement des alignements mémoires quelle que soit les méthodes de stockage choisies. Une solution serait de modifier la définition des types stochastiques, par exemple remplacer l'entier *accuracy* par un flottant *accuracy* mais cela suppose qu'il faudrait repenser toute la programmation de CADNA qui fait intervenir des opérations bit à bit sur ce champ là.

Comme nous l'avons montré en section 5.4.2, l'arithmétique stochastique discrète est un frein important pour l'obtention d'une routine DgemmCadna performante. Le véritable problème se situe au niveau du changement du mode d'arrondi après chaque opération. Afin de mieux exploiter le parallélisme interne des processeurs (le pipeline des instructions), il faudrait faire le maximum d'opérations entre deux changements successifs du mode d'arrondi. Nous proposons alors une modification à l'implémentation de l'arithmétique stochastique discrète.

5.6 La méthode CESTAC modifiée

5.6.1 La nouvelle implémentation de la méthode CESTAC

Les principes de l'arithmétique stochastique ont été présentés dans le chapitre 3. Nous avons alors expliqué qu'à chaque opération mathématique élémentaire, il est nécessaire d'effectuer aléatoirement un changement du mode d'arrondi. A titre d'exemple considérons l'addition de deux nombres stochastiques a et b. Elle est faite comme indiqué dans extrait de code 5.6.

Source 5.6 – Addition de deux nombres stochastiques a et b

```
if (RANDOM) rnd_switch();
res.x=a.x+b.x;
if (RANDOM) rnd_switch();
res.y=a.y+b.y;
rnd_switch();
res.z=a.z+b.z;
```

Les quatre modes d'arrondi de la norme IEEE 754 peuvent se ramener à deux modes d'arrondi : vers $+\infty$ et vers $-\infty$ (voir chapitre 3). Ce constat nous amène à la conclusion suivante : *pour une opération mathématique élémentaire entre deux vecteurs stochastiques, les opérations ne se font en réalité qu'avec deux modes d'arrondi différents*. De fait, on peut alors regrouper les opérations en deux groupes et ne faire que deux changements de mode d'arrondi.

Pour une addition entre deux vecteurs stochastiques $C[4] = A[4] + B[4]$ on pourrait l'implémenter comme dans l'extrait de code 5.7. :

Source 5.7 – Nouvelle implémentation de CESTAC

```
  if(random) rnd_switch()
  C[0].x = A[0].x + B[0].x ;
  C[0].z = A[0].z + B[0].z ;
  C[1].z = A[1].z + B[1].z ;
  C[2].x = A[2].x + B[2].x ;
  C[2].y = A[2].y + B[2].y ;
  C[3].x = A[3].x + B[3].x ;
  rnd_switch();
  C[0].y = A[0].y + B[0].y ;
  C[1].x = A[1].x + B[1].x ;
  C[1].y = A[1].y + B[1].y ;
  C[2].z = A[2].z + B[2].z ;
  C[3].y = A[3].y + B[3].y ;
  C[3].z = A[3].z + B[3].z ;
```

Cette nouvelle implémentation a pour vocation de permettre une meilleure exploitation du pipeline des instructions. En effet, nous effectuons plus d'une opération entre deux appels à la fonction de changement de mode d'arrondi *rnd_switch()*. Rappelons que l'appel à cette fonction nécessite de vider le pipeline des instructions (section 5.4). L'objectif étant de profiter au maximum de la taille du pipeline, nous choisissons de travailler sur des vecteurs de taille 4 ou 16 par la suite. On effectue ainsi 24 opérations mathématiques entre 2 changements d'arrondi lorsqu'on travaille sur des vecteurs de taille 16 et 6 opérations mathématiques entre 2 changements d'arrondi lorsqu'on considère des vecteurs de taille 4. Notons que les tailles sont des puissances de 2 afin de travailler sur des données alignées en mémoire.

Nous avons alors introduit cette nouvelle implémentation de CESTAC dans le noyau de calcul des versions DGBR16, DGBI16, DGBRML4 et DGBRML16. Ce noyau est dédié au calcul du résultat partiel d'un bloc de la matrice C (équation 5.1). Nous présentons les résultats dans le *tableau* 5.15.

Tableau 5.15 – Influence de la nouvelle implémentation de CESTAC : on modifie le noyau de calcul des versions DGBR16,DGBI16,DGBRML4 et DGBRML16.

Taille	DgemmCadnaV1	LinalgCdn	DGB16	Nouvelle Méthode CESTAC			
				DGBR16	DGBI16	DGBRML4	DGBRML16
512	23.0184	13.5067	22.6516	1.18531	1.17668	3.48546	0.935167
1024	242.81	109.074	183.461	10.3321	10.2815	27.9986	7.52359
2048	1971.23	869.126	1468.2	96.7742	96.2922	223.388	60.3249

Les résultats du *tableau* 5.15 sont indéniablement meilleurs que les résultats du *tableau* 5.14. On gagne un peu plus d'un facteur 20 entre les deux étapes. Ces dernières observations soulignent l'énorme influence des changements aléatoires de modes d'arrondi. D'autre part, remarquons qu'on obtient les meilleurs résultats avec les versions par bloc de 16 et principalement avec la version DGBRML16.

L'utilisation des algorithmes par blocs améliore considérablement les performances des codes si la taille des blocs est bien choisie. Cependant, en adaptant le stockage des matrices aux caractéristiques des machines, on peut espérer de meilleurs gains. Toutefois, la performance des versions par bloc de 16 soulève un autre problème, celui de la validité de la nouvelle implémentation CESTAC. En effet, contrairement à la version par bloc de 4, nous faisons peu de changements de mode d'arrondi.

Dans la prochaine section, nous abordons la question de la validité de la nouvelle implémentation de CESTAC et tenterons d'apporter des éléments de réponses pertinents en repartant des bases de la méthode CESTAC. Après cette parenthèse théorique, nous ferons un bilan complet du chapitre en section 5.7.

5.6.2 Sur la validité de la nouvelle implémentation

Dans l'optique d'améliorer les performance de DgemmCadna, une modification de la méthode CESTAC a été proposée dans la dernière section. Il importe alors de s'assurer que cette implémentation respecte les principes de base de la méthode. Rappelons d'abord quelques principes essentiels de la méthode CESTAC que nous avons exposés dans le chapitre 3.

Quelques rappels sur la méthode CESTAC

A partir du théorème 3.2.1, le résultat R d'une suite d'opérations informatiques peut être modélisé par :

$$R = r + \sum_{i=1}^{Sn} g_i(d) 2^{E_i - p} \varepsilon_i (\alpha_i - h_i) \tag{5.6}$$

où
- g_i représente des constantes ne dépendant que des données et de l'algorithme ;
- E_i les exposants des résultats intermédiaires ;
- α_i la quantité perdue lors de la troncature ou de l'arrondi ;
- h_i les perturbations aléatoires, ε_i les signes des résultats intermédiaires ;
- r le vrai résultat mathématique ;
- n le nombre d'opérations pendant l'exécution.

Une démonstration de ce théorème a été faite dans [Chesneaux, 1988, p.15]. Aussi, il a été montré que l'équation 5.6 a été établie sur la base de deux hypothèses fondamentales :

i) les erreurs d'arrondi α_i sont indépendantes centrées et uniformément réparties ;

ii) l'approximation au premier ordre en 2^{-p} dans la modélisation du résultat informatique R est valide.

Sur l'implémentation d'origine

Dans l'implémentation de base de la DSA, R est remplacé par un triplet R_x, R_y, R_z. En conséquence, l'équation 5.6 devient :

$$\begin{aligned} R_x &= r + \sum_{i=1}^{Sn} g_{x_i}(d) 2^{E_{x_i} - p} \varepsilon_i (\alpha_{x_i} - h_{x_i}) \\ R_y &= r + \sum_{i=1}^{Sn} g_{y_i}(d) 2^{E_{y_i} - p} \varepsilon_i (\alpha_{y_i} - h_{y_i}) \\ R_z &= r + \sum_{i=1}^{Sn} g_{z_i}(d) 2^{E_{z_i} - p} \varepsilon_i (\alpha_{z_i} - h_{z_i}) \end{aligned} \tag{5.7}$$

Dans les faits, $h_i \in \{-1; 1\}$; h_{x_i} et h_{y_i} sont choisis ; $h_{z_i} = \overline{h_{y_i}}$. De fait, choisir h_i implique choisir un mode d'arrondi.

Sur la nouvelle implémentation

Considérons maintenant les opérations par groupe de 4 comme si on travaillait simultanément sur des vecteurs de taille 4. L'équation 5.6 est équivalent à la suivante :

$$R = r + \sum_{k=1}^{Sn'} \sum_{j=1}^{4} p_{k_j} \tag{5.8}$$

où $Sn' = Sn/4$ et $p_{k_j} = g_{k_j}(d)2^{E_{k_j}-p}\varepsilon_{k_j}(\alpha_{k_j} - h_{k_j})$ et que

$$\begin{aligned} R_x &= r + \sum_{k=1}^{Sn'}\sum_{j=1}^{4} p_{k_{x_j}} \\ R_y &= r + \sum_{k=1}^{Sn'}\sum_{j=1}^{4} p_{k_{y_j}} \\ R_z &= r + \sum_{k=1}^{Sn'}\sum_{j=1}^{4} p_{k_{z_j}} \end{aligned} \tag{5.9}$$

avec

$$\begin{aligned} p_{k_{x_0}} &= g_{k_{x_0}}(d)2^{E_{k_{x_0}}-p}\varepsilon_{k_{x_0}}(\alpha_{k_{x_0}} - h_{k_{x_0}}) \\ p_{k_{y_0}} &= g_{k_{y_0}}(d)2^{E_{k_{y_0}}-p}\varepsilon_{k_{y_0}}(\alpha_{k_{y_0}} - h_{k_{y_0}}) \\ p_{k_{z_0}} &= g_{k_{z_0}}(d)2^{E_{k_{z_0}}-p}\varepsilon_{k_{z_0}}(\alpha_{k_{z_0}} - \overline{h_{k_{y_0}}}) \\ p_{k_{x_1}} &= g_{k_{x_1}}(d)2^{E_{k_{x_1}}-p}\varepsilon_{k_{x_1}}(\alpha_{k_{x_1}} - h_{k_{x_1}}) \\ p_{k_{y_1}} &= g_{k_{y_1}}(d)2^{E_{k_{y_1}}-p}\varepsilon_{k_{y_1}}(\alpha_{k_{y_1}} - h_{k_{y_1}}) \\ p_{k_{z_1}} &= g_{k_{z_1}}(d)2^{E_{k_{z_1}}-p}\varepsilon_{k_{z_1}}(\alpha_{k_{z_1}} - \overline{h_{k_{y_1}}}) \\ p_{k_{x_2}} &= g_{k_{x_2}}(d)2^{E_{k_{x_2}}-p}\varepsilon_{k_{x_2}}(\alpha_{k_{x_2}} - h_{k_{x_2}}) \\ p_{k_{y_2}} &= g_{k_{y_2}}(d)2^{E_{k_{y_2}}-p}\varepsilon_{k_{y_2}}(\alpha_{k_{y_2}} - h_{k_{y_2}}) \\ p_{k_{z_2}} &= g_{k_{z_2}}(d)2^{E_{k_{z_2}}-p}\varepsilon_{k_{z_2}}(\alpha_{k_{z_2}} - \overline{h_{k_{y_2}}}) \\ p_{k_{x_3}} &= g_{k_{x_3}}(d)2^{E_{k_{x_3}}-p}\varepsilon_{k_{x_3}}(\alpha_{k_{x_3}} - h_{k_{x_3}}) \\ p_{k_{y_3}} &= g_{k_{y_3}}(d)2^{E_{k_{y_3}}-p}\varepsilon_{k_{y_3}}(\alpha_{k_{y_3}} - h_{k_{y_3}}) \\ p_{k_{z_3}} &= g_{k_{z_3}}(d)2^{E_{k_{z_3}}-p}\varepsilon_{k_{z_3}}(\alpha_{k_{z_3}} - \overline{h_{k_{y_3}}}) \end{aligned} \tag{5.10}$$

Considérons maintenant l'équation 5.10, 8 h_i ont été choisis aléatoirement, et 4 dépendent du dernier mode d'arrondi. Pour que la méthode CESTAC soit valide, il est important qu'il y ait 2 modes d'arrondi à chaque étape : il faut alors ***au moins deux*** h_i ***différents***. Avec la nouvelle implémentation que nous avons présentée dans la section précédente, l'équation 5.10 peut être

réécrite comme ci-dessous :

$$
\begin{aligned}
p_{k_{x_0}} &= g_{k_{x_0}}(d)2^{E_{k_{x_0}}-p}\varepsilon_{k_{x_0}}(\alpha_{k_{x_0}} - h_1)\\
p_{k_{y_0}} &= g_{k_{y_0}}(d)2^{E_{k_{y_0}}-p}\varepsilon_{k_{y_0}}(\alpha_{k_{y_0}} - h_2)\\
p_{k_{z_0}} &= g_{k_{z_0}}(d)2^{E_{k_{z_0}}-p}\varepsilon_{k_{z_0}}(\alpha_{k_{z_0}} - h_1)\\
p_{k_{x_1}} &= g_{k_{x_1}}(d)2^{E_{k_{x_1}}-p}\varepsilon_{k_{x_1}}(\alpha_{k_{x_1}} - h_2)\\
p_{k_{y_1}} &= g_{k_{y_1}}(d)2^{E_{k_{y_1}}-p}\varepsilon_{k_{y_1}}(\alpha_{k_{y_1}} - h_2)\\
p_{k_{z_1}} &= g_{k_{z_1}}(d)2^{E_{k_{z_1}}-p}\varepsilon_{k_{z_1}}(\alpha_{k_{z_1}} - h_1)\\
p_{k_{x_2}} &= g_{k_{x_2}}(d)2^{E_{k_{x_2}}-p}\varepsilon_{k_{x_2}}(\alpha_{k_{x_2}} - h_1)\\
p_{k_{y_2}} &= g_{k_{y_2}}(d)2^{E_{k_{y_2}}-p}\varepsilon_{k_{y_2}}(\alpha_{k_{y_2}} - h_1)\\
p_{k_{z_2}} &= g_{k_{z_2}}(d)2^{E_{k_{z_2}}-p}\varepsilon_{k_{z_2}}(\alpha_{k_{z_2}} - h_2)\\
p_{k_{x_3}} &= g_{k_{x_3}}(d)2^{E_{k_{x_3}}-p}\varepsilon_{k_{x_3}}(\alpha_{k_{x_3}} - h_1)\\
p_{k_{y_3}} &= g_{k_{y_3}}(d)2^{E_{k_{y_3}}-p}\varepsilon_{k_{y_3}}(\alpha_{k_{y_3}} - h_2)\\
p_{k_{z_3}} &= g_{k_{z_3}}(d)2^{E_{k_{z_3}}-p}\varepsilon_{k_{z_3}}(\alpha_{k_{z_3}} - h_2)
\end{aligned}
\tag{5.11}
$$

où $h2 = \overline{h1}$.

L'équation 5.11 est équivalente à l'équation 5.12 ci-dessous :

$$
\begin{aligned}
&1^{ère}partie\\
&p_{k_{x_0}} = g_{k_{x_0}}(d)2^{E_{k_{x_0}}-p}\varepsilon_{k_{x_0}}(\alpha_{k_{x_0}} - h_1)\\
&p_{k_{z_0}} = g_{k_{z_0}}(d)2^{E_{k_{z_0}}-p}\varepsilon_{k_{z_0}}(\alpha_{k_{z_0}} - h_1)\\
&p_{k_{z_1}} = g_{k_{z_1}}(d)2^{E_{k_{z_1}}-p}\varepsilon_{k_{z_1}}(\alpha_{k_{z_1}} - h_1)\\
&p_{k_{x_2}} = g_{k_{x_2}}(d)2^{E_{k_{x_2}}-p}\varepsilon_{k_{x_2}}(\alpha_{k_{x_2}} - h_1)\\
&p_{k_{y_2}} = g_{k_{y_2}}(d)2^{E_{k_{y_2}}-p}\varepsilon_{k_{y_2}}(\alpha_{k_{y_2}} - h_1)\\
&p_{k_{x_3}} = g_{k_{x_3}}(d)2^{E_{k_{x_3}}-p}\varepsilon_{k_{x_3}}(\alpha_{k_{x_3}} - h_1)\\
&2^{ième}partie\\
&p_{k_{y_0}} = g_{k_{y_0}}(d)2^{E_{k_{y_0}}-p}\varepsilon_{k_{y_0}}(\alpha_{k_{y_0}} - \overline{h1})\\
&p_{k_{x_1}} = g_{k_{x_1}}(d)2^{E_{k_{x_1}}-p}\varepsilon_{k_{x_1}}(\alpha_{k_{x_1}} - \overline{h1})\\
&p_{k_{y_1}} = g_{k_{y_1}}(d)2^{E_{k_{y_1}}-p}\varepsilon_{k_{y_1}}(\alpha_{k_{y_1}} - \overline{h1})\\
&p_{k_{z_2}} = g_{k_{z_2}}(d)2^{E_{k_{z_2}}-p}\varepsilon_{k_{z_2}}(\alpha_{k_{z_2}} - \overline{h1})\\
&p_{k_{y_3}} = g_{k_{y_3}}(d)2^{E_{k_{y_3}}-p}\varepsilon_{k_{y_3}}(\alpha_{k_{y_3}} - \overline{h1})\\
&p_{k_{z_3}} = g_{k_{z_3}}(d)2^{E_{k_{z_3}}-p}\varepsilon_{k_{z_3}}(\alpha_{k_{z_3}} - \overline{h1})
\end{aligned}
\tag{5.12}
$$

Dans notre cas (équation 5.12), seul $h1$ est choisi aléatoirement. Dans la version d'origine, pour 12 opérations, 8 h_i sont choisis aléatoirement.

Cette nouvelle formulation affecte légèrement les hypothèses sur lesquelles l'équation 5.6 a été établie. Cependant, le point le plus important pour la validité de l'implémentation concerne

le nombre de modes d'arrondi choisis aléatoirement qui doit être élevé. Cette contrainte reste vérifiée dans le cas de la nouvelle implémentation. Par exemple, si l'on considère la multiplication de 2 matrices carrées (1024×1024), il y a $2 \times 1024 \times 1024 \times 1024$ opérations en virgule flottante (2 Gflops). Avec la DSA, il y a 3×2 Gflops. Avec l'implémentation d'origine, nous avons 4 Giga h_i choisis aléatoirement. Avec la nouvelle mise en œuvre, il y a 0.5 Giga h_i choisis aléatoirement.

5.7 Conclusion

Ce chapitre a été consacré à l'implémentation de l'arithmétique stochastique discrète dans les bibliothèques de calcul scientifique, notamment dans les routines BLAS et en particulier dans la routine de produit matriciel DGEMM.

Dans un premier temps, nous avons effectué des tests de performance des principales implémentations disponibles. Nos expérimentations ont montré que les versions GotoBLAS et MKL d'Intel étaient les plus performantes. Ces deux versions atteignent des performances quasi-maximales pour les routines du niveau 3 des BLAS (un peu plus de 90% de la performance crête théorique de la machine de test). Ces premières expérimentations permettent également de noter qu'il est plus efficace d'optimiser les routines du niveau 3, celles des niveaux 1 et 2 étant bloquées par les accès mémoire.

Dans un second temps, nous avons comparé les implémentations de référence des BLAS à des implémentations intégrant l'arithmétique stochastique discrète en utilisant les « templates ». Nous avons alors montré qu'une implémentation directe de la DSA dans les routines BLAS pouvait engendrer un surcoût très important. A titre d'exemple, nous avons obtenu un rapport supérieur à 1000 pour la routine DGEMM quand on se compare à l'implémentation GotoBLAS. Ce facteur est encore plus important dès lors que le mode d'auto-validation de la bibliothèque CADNA est activé. En effet, il a été remarqué que plus de 85% des temps d'exécution sont consacrés à ce mode quand il est utilisé. Des éléments de réponse à ces facteurs sont apportés dans la section 5.4.2.

Enfin, nous avons travaillé pour réduire ces importants surcoûts. Nous avons d'abord travaillé sur l'amélioration de l'utilisation de la mémoire en introduisant des blocs et puis en modifiant les modes de stockage des matrices. L'objectif de ce premier travail était d'adapter au maximum le stockage des matrices aux caractéristiques mémoire de notre machine. Un résultat récapitulatif de cette partie est présenté dans le *tableau* 5.16. Puis, nous avons proposé une nouvelle implémentation de la méthode CESTAC afin de réduire le surcoût dû aux changements de mode d'arrondi. Comme nous le présentons dans les tableaux 5.17 et 5.18, cette nouvelle implémentation améliore considérablement les résultats.

Toutes les modifications apportées ont ainsi permis d'améliorer les temps d'exécution (voir les tableaux récapitulatifs 5.16, 5.17 et 5.18). Finalement, nous avons obtenu un gain d'environ 30 par rapport à la première version (*tableau* 5.17). Par rapport à GotoBlas, le surcoût dû à l'implémentation de la méthode CESTAC a été réduit à 35 grâce à nos développements (surcoût initial de 1100). Rappelons que l'implémentation de la DSA implique un surcoût minimum puisqu'on effectue 3 fois plus d'opérations flottantes et 4 fois plus d'accès mémoire. Ces résultats montrent que l'utilisation des blocs, d'un modèle de stockage de matrice opportun et d'un bon sens pour les boucles internes contribuent considérablement à limiter le surcoût dû à la méthode CESTAC.

Nous avons envisagé d'améliorer notre routine en utilisant les instructions vectorielles (SSE 4.2). Nous avons commencé en implémentant un produit scalaire. Notons que l'utilisation des

Tableau 5.16 – Tableau récapitulatif des différentes versions de DgemmCadna après amélioration de l'utilisation de la mémoire (les boucles ne sont pas déroulées). Le temps de DgemmCadnaV1 a été normalisé à 1 afin de mettre en évidence les différents gains. Les autres versions sont DGB16 (Dgemm par Blocs de taille 16), DBGR16 (Dgemm par Blocs Recursifs de taille 16), DGBI16 (DGemm par Blocs Itératif de taille 16 avec découpage récursif), DGBRML4 et DGBRML16 (DGemm utilisant le mode de stockage BRML avec des sous-matrices de taille n)

Taille	LinalgCdn	DGB16	DGBR16	DGBI16	DGBRML4	DGBRML16
512	0.59	0.98	0.64	0.598	0.596	0.595
1024	0.45	0,76	0,49	0,468	0,450	0,451
2048	0,44	0,74	0,48	0,462	0.441	0,442

Tableau 5.17 – Tableau récapitulatif des différentes versions de DgemmCadna après l'introduction nouvelle implémentation de CESTAC. Nous confrontons nos routines aux versions GotoBLAS et Intel MKL. Le temps de DgemmCadnaV1 a été normalisé à 1 afin de mettre en évidence les différents gains.

Taille	LinalgCdn	DGB16	Nouvelle Méthode CESTAC				GOTO	MKL
			DGBR16	DGBI16	DGBRML4	DGBRML16	Mode Séquentiel	
512	0.59	0.98	0,0515	0,0511	0,1514	0,041	0,001171	0,001167
1024	0.45	0.76	0,0426	0,0423	0,1153	0,031	0,0008836	0,0008831
2048	0.44	0.74	0,04909	0,04884	0,0752	0,030	0,000864	0,000866

Tableau 5.18 – Tableau récapitulatif des différentes versions de DgemmCadna après l'introduction nouvelle implémentation de CESTAC. Nous confrontons nos routines à la version de Goto. Le temps d'exécution de GotoBLAS a été normalisé à 1 afin de mettre en évidence les différents facteurs et donc les gains.

Taille	DgemmCadnaV1	LinalgCdn	DGB16	DGBR16	DGBI16	DGBRML4	DGBRML16
1024	1131	508	855	48	47	130	35
2048	1157,3	510	862	56,81	56	131	35

instructions vectorielles dans un produit scalaire est le moyen le plus facile pour diviser les temps d'exécution en double précision par deux (SSE) ou par 4 (AVX). Cependant, les résultats obtenus avec les types stochastiques sont mauvais. Par exemple, pour un vecteur de taille 1000000, nous obtenons une accélération nettement inférieure à 1 soit 0.137575. Ceci s'explique notamment par la non homogénéité des types stochastiques. La vectorisation peut constituer une solution d'amélioration des performances de DgemmCadna, mais pour cela, il faudra trouver une manière efficace d'associer instructions SSE (ou AVX) aux types stochastiques. Toutefois, il existe une solution qui nous permettrait d'obtenir de meilleurs gains : optimiser le noyau de calcul en le codant à la main en langage assembleur. C'est ce qui a été fait pour les "kernel" de GotoBLAS. Une autre piste envisageable serait de gérer à la main les chargements des données en mémoire.

Implémenter un produit matriciel ayant des performances optimales est une tâche extrêmement compliquée. Toutefois, le titre d'un article [Yotov *et al.*, 2005] paru en 2005 était : *Is Search Really Necessary to Generate High-Performance BLAS ?* Les auteurs comparaient un modèle d'optimisation de routines dit "model-driven optimization" au modèle AEOS (Automated Empirical Optimization Software) utilisé dans ATLAS. Ils démontraient qu'il était possible d'avoir de bonnes performances sans pour autant avoir recours à des noyaux de calcul optimisés à la main. Mais c'était avant l'avènement de GotoBLAS (2008). C'était aussi avant les processeurs multi-cores et les instructions vectorielles types AVX, les accélérateurs de calcul (GPU, FPGA, etc) [Kirschenmann, 2012, Chap. 2] et aujourd'hui les processeurs many-cores (Intel Xeon Phi).

Nous pensons qu'on ne peut avoir une version unique de BLAS qui puisse atteindre des performances optimales quelques soient les architectures matérielles. La question serait alors *que faire pour ne pas avoir à re-développer toute la routine à chaque nouvelle architecture de processeur*. L'approche proposée par les BLIS peut éventuellement être une solution durable. Cependant, on sera toujours amené à traiter des cas particuliers comme celui que nous avons traité dans ce chapitre. On peut aussi évoquer d'autres cas singuliers comme ceux traités dans les codes de neutronique où les matrices ont des structures qu'on ne retrouve nul par ailleurs. Dans ces cas là, comme dans le cas général, il importera de s'adapter aux caractéristiques matérielles avec une contrainte de *ré-utilisabilité* et de *portabilité*. Et pourquoi ne pas généraliser le langage C++ qui permet d'extraire assez facilement la générécité des codes ?

Enfin, notons que la méthodologie mise en œuvre dans ce chapitre peut être étendue aux autres routines d'algèbre linéaire. L'objectif, à long terme, est l'obtention de bibliothèques de calcul scientifique compatibles avec l'outil CADNA afin de faciliter son utilisation dans les grands codes de calcul. Les deux prochains chapitres sont consacrés à l'étude la qualité numérique de codes industriels avec CADNA. Nous nous intéressons particulièrement au code Telemac-2D.

Chapitre 6

Etude de la propagation des erreurs d'arrondi dans un code industriel parallèle

Sommaire

Nous avons montré dans le chapitre 1 l'importance de la simulation numérique pour les acteurs industriels comme EDF. En effet, simuler permet de décider, d'anticiper et de prévoir les investissements pour l'avenir. Cependant, les résultats des logiciels de simulation numérique ne sont pas toujours fiables. Plusieurs étapes séparent l'étude du phénomène physique et l'obtention du code de simulation et chaque étape peut être éventuellement source d'erreurs. Les exemples présentés en section 1.4.1, montrent que cette incertitude sur la qualité des résultats est due en partie à la propagation des erreurs d'arrondis au cours de l'exécution des codes. Il est alors primordial de valider la qualité numérique des résultats des codes de calcul. Rappelons que la validation numérique d'un logiciel de simulation consiste à étudier sa capacité à produire des résultats justes malgré la propagation des erreurs d'arrondi.

Ce chapitre est consacré à l'étude de la propagation des erreurs d'arrondi dans un code industriel de simulation hydrodynamique : Telemac-2D. Ce travail a été motivé par les exigences de reproductibilité nécessaires aux études des chercheurs du Laboratoire National d'Hydraulique et d'environnement (LNHE) d'EDF R&D. Notre principal objectif est de détecter les potentielles sources d'instabilités numériques dans le code et de proposer des solutions afin de les éviter. Dans cette optique, nous avons mené une étude de la qualité numérique avec l'outil CADNA (que nous avons présenté au chapitre 3). Puis, nous avons travaillé à améliorer la précision des résultats sans pour autant négliger les contraintes de performance des codes industriels. Ce travail a été présenté à PARENG2013 [Montan *et al.*, 2013c].

Plan du chapitre : Nous commençons par présenter le système Telemac-Mascaret et ses applications dans la section 6.1. Une attention particulière est portée au code Telemac-2D que nous étudions ici (section 6.2). Nous présentons les équations résolues par le code, l'algorithme implémenté ainsi que le parallélisme mis en place. Nous évoquons également les problèmes de reproductibilité numérique des résultats issus de l'utilisation du parallélisme. Ensuite, nous montrons en section 6.3 comment utiliser et exploiter les diagnostics de l'outil de validation CADNA dans un tel code. Dans la section 6.4, les algorithmes compensés sont introduits. Nous montrons qu'ils peuvent aider à améliorer la précision des produits scalaires. Enfin, nous montrons que l'utilisation de ces algorithmes, en particulier dans un code industriel, n'influence presque pas les performances de ce dernier.

6.1 La suite Telemac-Mascaret

La suite logicielle Telemac-Mascaret [50] est un ensemble de solveurs et outils dédié à la résolution de problèmes issus de la mécanique des fluides à surface libre. Développée initialement par le Laboratoire National d'Hydraulique et d'Environnement (LNHE) d'EDF R&D, la suite logicielle est diffusée en open source depuis 2010 et est gérée par un consortium de six centres de recherche internationaux : Artelia (anciennement Sogreah, France), Bundesanstalt für Wasserbau (BAW, Allemagne), le Centre d'Études Techniques Maritimes et Fluviales (CETMEF, France), STFC Daresbury Laboratory (Royaume-Uni), HR Wallingford (Royaume-Uni) et EDF R&D.

50. voir http://www.opentelemac.org/

Tableau 6.1 – Les principaux modules de Telemac-Mascaret

Module	Rôle
Hydrodynamique	
ARTEMIS	Propagation des vagues dans les ports ou sur les côtes
MASCARET	Écoulements à surface libre unidimensionnel
Telemac-2D	Écoulements à surface libre bidimensionnel
Telemac-3D	Écoulements à surface libre tridimensionnel
TOMAWAC	Propagation des vagues dans la zone côtière
Transport / Dispersion	
SISYPHE	Transport des sédiments 2D
SEDI-3D	Transport de sédiments en suspension 3D
Pre-/post traitements	
STBTEL	Conversion de divers formats de géométrie
POSTEL-3D	Post-traitement de sections 2D à travers les résultats d'une simulation 3D

Le système Telemac-Mascaret est composé de plusieurs codes (encore appelés modules [51]). Chaque code est dédié à la résolution d'un problème physique particulier et certains codes peuvent être couplés entre eux. Le *tableau* 6.1 présente un bref aperçu des principaux codes disponibles dans le système Telemac-Mascaret. La discrétisation des équations dans les différents modules est basée sur la méthode des éléments finis. L'espace est discrétisé sous la forme d'un maillage non structuré d'éléments triangulaires.

Les différents codes du système sont basés sur le module *bief* (BIbliothèque d'Elements Finis) qui regroupe tous les algorithmes numériques et les fonctionnalités de base de l'algèbre linéaire nécessaires au système Telemac-Mascaret (produit scalaire, produit matrice-vecteur, résolutions de systèmes linéaires, etc). Signalons également que les différents codes du système s'exécutent sur Windows et Linux/UNIX et qu'ils ont déjà été exécutés sur divers supercalculateurs (Cray, Fujitsu, IBM, etc).

Les systèmes, tels que Telemac-Mascaret, sont devenus des outils indispensables pour les études liées à l'environnement. En effet, le phénomène physique modélisé dépend directement des termes sources utilisés. Par exemple, il est possible d'étudier la qualité de l'eau en prenant en compte tout le fonctionnement de l'éco-système. C'est ce qui a été fait dans le cas de l'Étang de Berre (voir exemple ci-dessous). Telemac-Mascaret est également utilisé pour les études d'impact et de dimensionnement des ouvrages (barrages), pour le calcul des marées et la simulation des crues. Ces études sont parfois commanditées par l'Autorité de Sûreté Nucléaire.

Un exemple d'application : l'Étang de Berre

D'une superficie de 155 km^2, l'étang de Berre est une vaste étendue d'eau saumâtre (980 millions de m^3 d'eau) située à l'ouest de Marseille et reliée à la mer Méditerranée par le canal de Caronte. Des études ont été menées par le LNHE avec le code Telemac-3D pour comprendre de façon plus précise le fonctionnement de l'éco-système de l'Étang Berre. En fait, à la suite d'une plainte issue d'un collectif de pêcheurs déposée le 15 décembre 1997, la Cour de Justice Européenne a condamné la France pour manquements à ses obligations résultant de l'appli-

51. Par la suite, nous désignerons les modules au sens implémentation informatique en italiques et les codes en écriture normale. Cette petite différence est due au fait qu'un code est composé de plusieurs modules informatiques.

cation du protocole relatif à la protection de la mer Méditerranée contre la pollution d'origine tellurique. Suite à l'arrêt de la Cour de Justice Européenne, une proposition transitoire a été faite par EDF sous le nom d'expérimentation de lissage. Cette proposition a servi de base aux propositions de la France transmises à la Commission Européenne en février 2005. Ce plan opérationnel vise à limiter les rejets instantanés d'eau douce et de limon dans l'étang de Berre, en moyenne annuelle comme en volume hebdomadaire. La modélisation avec Telemac-3D avait également pour but de calculer les niveaux probables de salinité de l'étang suite à la mise en œuvre du lissage proposé par EDF [Durand, 2011].

Dans le cadre de ce mémoire, nous nous proposons d'étudier la qualité numérique des résultats de Telemac-2D. En ce sens, le code Telemac-2D est présenté dans la prochaine section. Notons qu'un travail similaire a été effectué sur le module Telemac-3D [Denis, 2012, Denis et Montan, 2012].

6.2 Le logiciel Telemac-2D

Les logiciels Telemac-2D et Telemac-3D simulent les écoulements à surface libre. Pour cela, Telemac-3D résout les équations de Navier-Stokes qui constituent la pierre angulaire de la mécanique des fluides en calculant à chaque point du maillage la hauteur d'eau (h) et les composantes de la vitesse (u, v, w). Les nombreux logiciels de CFD [52] et ouvrages dans la littérature dédiés à la résolution de ces équations soulignent leurs extrêmes importances et la difficulté de leur résolution. De l'avis même des experts en CFD, résoudre les équations complètes de Navier-Stokes est extrêmement complexe : la résolution se fait généralement sous certaines hypothèses. Le logiciel Telemac-3D permet de résoudre les équations Navier-Stokes non hydrostatiques (équations complètes). Cependant, cette résolution se fait avec une petite subtilité pour le calcul de la troisième composante de la vitesse (w) qui n'est pas directement calculée mais déduite à partir du calcul des deux autres composantes et de l'équation de continuité. En fait, on parle d'équations non hydrostatiques pour signaler qu'aucune hypothèse n'est faite sur la pression hydrostatique.

Le code Telemac-2D est quand à lui dédié à la résolution des équations de Saint-Venant, qui sont en fait une forme simplifiée des équations de Navier-Stokes. En effet, les équations de Saint-venant sont obtenues en faisant une moyenne sur la verticale (intégration) des équations de Navier-Stokes. En plus de cette opération, l'hypothèse de pression hydrostatique est faite. Cette hypothèse consiste à simplifier la vitesse w en négligeant la diffusion, les termes sources et l'accélération. L'ouvrage de Jean-Michel Hervouet [Hervouet, 2007], père du système Telemac, présente de façon claire et précise les principes de la modélisation numérique faite dans les codes Telemac-2D et Telemac-3D, les différentes formes de ces équations et leurs résolutions. Nous présentons un aperçu des équations en annexe de ce document (annexe C).

6.2.1 Présentation générale

Les équations de Saint-Venant, encore appelées "de Barré de Saint-Venant", permettent de modéliser les écoulements à surface libre en eaux peu profondes, d'où leur appellation anglaise "Shallow water equations". Les équations se décomposent en deux parties : l'équation de continuité (qui exprime la conservation de la masse de fluide) et l'équation de quantité de mouvement (exprimant la relation fondamentale de la dynamique). L'équation de l'énergie, qui devrait exprimer les variations de la température est ignorée. A ces deux équations, s'ajoute

52. Computational Fluid Dynamics : abréviation utilisée pour désigner la mécanique des fluides

l'équation du traceur qui représente une température ou une grandeur physique quelconque contenue dans l'eau (colorant, sédiment). Dans le cas de Telemac-2D, la température est utilisée comme traceur responsable d'effets de flottabilité. Nous présentons ci-dessous ses quatre équations en coordonnées cartésiennes.

Équation de continuité :

$$\frac{\partial h}{\partial t} + \overrightarrow{u} \cdot \overrightarrow{grad}(h) + h div(\overrightarrow{u}) = S_{ce} \tag{6.1}$$

Équations de mouvement (selon x puis selon y) :

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} = -g\frac{\partial Z_s}{\partial x} + F_x + \frac{1}{h} div(h\nu_e \overrightarrow{grad}(u)) \tag{6.2}$$

$$\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} = -g\frac{\partial Z_s}{\partial y} + F_y + \frac{1}{h} div(h\nu_e \overrightarrow{grad}(v)) \tag{6.3}$$

Conservation du traceur :

$$\frac{\partial T}{\partial t} + \overrightarrow{u} \cdot \overrightarrow{grad}(T) = \frac{1}{h} div(h\nu_T \overrightarrow{grad}(T)) + \frac{(T_{sce} - T)S_{ce}}{h} \tag{6.4}$$

avec :
- h hauteur d'eau ;
- u, v composantes de la vitesse ;
- T température ;
- g accélération de la pesanteur ;
- ν_e, ν_T coefficients de diffusion de la vitesse et du traceur ;
- Z_s cote de la surface libre ;
- t temps ;
- x, y composantes d'espace horizontales ;
- S_{ce} source ou puits de fluide ;
- F_x, F_y termes source ou puits des équations dynamiques ;
- T_{sce} valeur du traceur à la source.

Telemac-2D résout ces quatre équations à l'aide de la méthode des éléments finis en calculant en chaque point du maillage la hauteur d'eau h et les deux composantes u et v de la vitesse. Les termes F_x, F_y sont des termes sources représentant le vent, la force de Coriolis, le frottement sur le fond, une source ou un puits de quantité de mouvement dans le domaine.

En ce qui concerne l'algorithme de résolution, les équations de départ sont les équations de Saint-Venant non conservatives en formulation hauteur-vitesse. Les termes de ces équations sont traités en une ou plusieurs étapes (cas de convection par la méthode des caractéristiques [Hervouet, 2003, p.165]). La méthode des caractéristiques est conseillée en raison de l'aspect hyperbolique des équations. Avec cette méthode, les équations sont traitées en deux étapes grâce à la méthode des pas fractionnaires [Hervouet, 2003, p.96] :

1. convection (transport des grandeurs physique h, u, v et T) ;
2. propagation, diffusion et termes sources des équations dynamiques.

Quand la méthode des caractéristiques n'est pas choisie, les termes de convection sont également traités dans la seconde partie. Au cours de cette étape basée sur la méthode des éléments

finis, les équations continues sont transformées en un système linéaire creux ayant pour inconnues les valeurs h, u et v aux nœuds du maillage. Le système est résolu généralement par une méthode itérative de type gradient conjugué. Le chapitre 4 de [Hervouet, 2003] est consacré entièrement à la résolution des équations "de Barré de Saint-Venant".

6.2.2 Le code et la gestion du parallélisme

Le code

Le code est un ensemble de sous-programmes fortran 90 organisé en plusieurs modules (modules fortran). Les programmes peuvent être modifiés par l'utilisateur selon les besoins spécifiques des modèles mis en œuvre. La *bief* demeure l'élément essentiel qui regroupe toutes les fonctions de base. Comme son nom l'indique, le module *telemac2d* regroupe les fonctions dédiées exclusivement à l'implémentation de l'algorithme Telemac-2D, dont le programme principal *homere_telemac2d(⋯)* et les principales routines qui y sont appelées. Ces fonctions font appel aux routines de la *bief*. Suivant, les besoins physiques, *telemac2d* peut être couplé avec les modules *sisyphe* (pour la prise en compte du transport des sédiments) et *tomawac* (pour la prise en compte de la propagation des vagues).

Une simulation Telemac-2D est lancée à partir d'un script écrit en Perl ou en Python et d'un cas test. Le cas test est composé de plusieurs fichiers dont trois obligatoires qui sont cités ci-dessous :

- le fichier des paramètres, qui contient la configuration de la simulation ;
- le fichier de maillage, qui contient les informations concernant le maillage ;
- le fichier des conditions aux limites, qui contient la description du type de chacune des frontières.

Le cas test peut contenir aussi un fichier de référence (qui contient le calcul de référence utilisé dans le cadre de la procédure de validation), un fichier Fortran, qui contient les sous-programmes particuliers à la simulation (sous-programme de Telemac-2D modifié ou spécifiquement créé) et d'autres fichiers pour des informations complémentaires sur la physique ou les modèles.

Le parallélisme

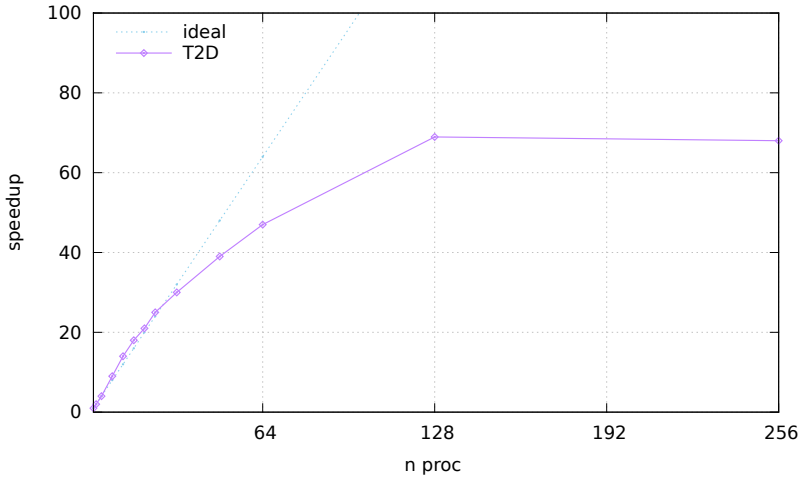
Historiquement, le code a été parallélisé sur des machines vectorielles mais les performances obtenues ont été jugées insuffisantes [Hervouet, 2003, p.233]. Telemac-2D a été testé récemment sur un maillage de 25 millions d'éléments finis sur 16384 processeurs (accélération de 3.4 entre 1024 et 4096 processeurs) montrant le potentiel de ce code pour simuler des phénomènes hydrodynamiques à grande échelle [Moulinec *et al.*, 2011b]. Le parallélisme actuel est basé une approche *décomposition de domaine*. L'idée est de distribuer à chaque processus une partie du domaine et de lui faire résoudre le problème d'hydrodynamique sur son sous-domaine. Ce type de parallélisme est qualifié de *parallélisme SPMD* pour Single Program Multiple Data : le programme confié à chaque processus est le même mais les données sont différentes.

La mise en place du parallélisme a été assez simple pour les algorithmes explicites[53] : chaque équation ne dépendant que des points du voisinage proche, on confie à chaque processus la résolution des équations d'un nombre de points et les données des voisins issues du pas de temps précédent. En revanche, il est plus difficile pour les algorithmes implicites qui nécessitent plus de communications et d'échanges de données entre les processus. C'est

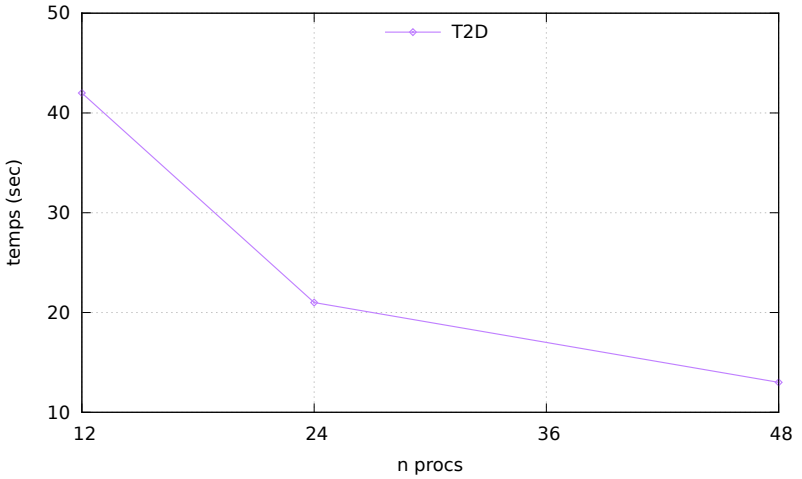
53. http://en.wikipedia.org/wiki/Explicit_and_implicit_methods

Figure 6.1 – Performance de Telemac-2D sur le cluster Ivanoe et la machine IBM iDataplex

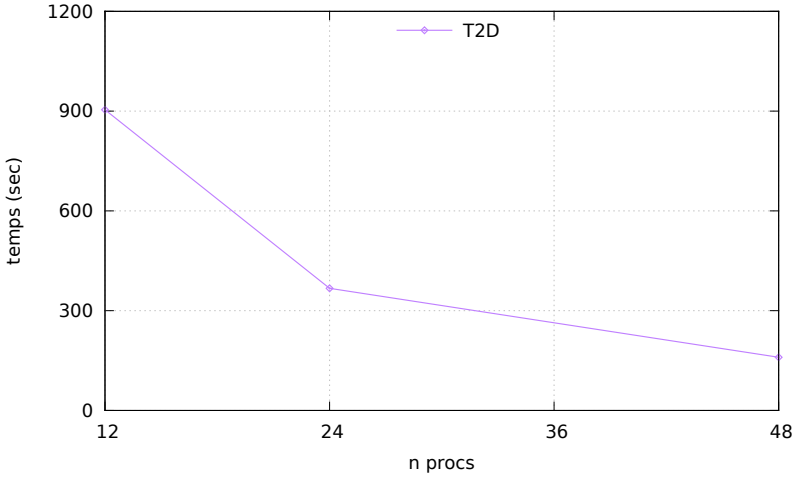
(a) Accélération (speedup) sur Ivanoe (cas de test Malpasset avec un maillage de 104000 éléments)



(b) Temps d'exécution sur IBM iDataplex pour un maillage de 0.1 million d'éléments (cas Malpasset)



(c) Temps d'exécution sur IBM iDataplex pour un maillage de 0.4 million d'éléments (cas Malpasset)



d'ailleurs ce qui justifie le problème de reproductibilité des résultats (voir section 6.2.4) entre l'exécution en parallèle et séquentielle. Les calculs ne font pas dans le même ordre alors que le calcul sur les nombres flottants n'est pas associatif (chapitre 2). Notons également, que deux étapes principales successives sont réalisées à chaque pas de temps : l'étape de calcul et l'étape de communication.

Finalement, le parallélisme peut se résumer à :

- la décomposition de domaine (ou partition) : le maillage non structuré est divisé en N sous-domaines SD_j ($1 \leq j \leq N$) en utilisant l'outil de partitionnement de graphe METIS [Karypis et Kumar, 1998] ;
- la communication entre processus basée sur l'échange de messages via le standard MPI ;
- l'adaptation de quelques algorithmes comme l'assemblage de vecteur, produit scalaire, etc.

Les résultats de nos expérimentations sur la performance de Telemac-2D sont présentés sur la *figure* 6.1. Ces performances ont été mesurées sur le cluster Ivanoe (annexe A.3) d'EDF R&D et sur la machine IBM iDataplex (annexe A.4) du STFC Daresbury Laboratory. Toutes les simulations effectuées dans ce chapitre portent sur le cas de test Malpasset qui est présenté en section 6.2.3. Les performances de Telemac-2D sont acceptables : l'accélération du code évolue normalement jusqu'à 48 processus (efficacité d'environ 1). Par contre, pour le cas test avec 104000 éléments, le code devient moins efficace à partir de 128 processus utilisés (*figure* 6.1a).

6.2.3 Un exemple d'application : La rupture du barrage Malpasset

L'accident de Malpasset est un exemple de rupture totale de barrage voûte[54]. Le barrage, situé dans la vallée du Reyran (département du Var) était construit pour l'irrigation et servait de réserve d'eau potable. Il avait une capacité de 55 millions de mètre cubes. Sa hauteur était de 66.5 mètres, sa longueur en crête 223 mètres, et l'épaisseur variait de 1.55 à 6.77 mètres. La rupture du barrage est survenue le 2 décembre 1959 après une période de pluies violentes entraînant des dégâts considérables (433 victimes, autoroute détruite sur 800 mètres, pont emporté, etc).

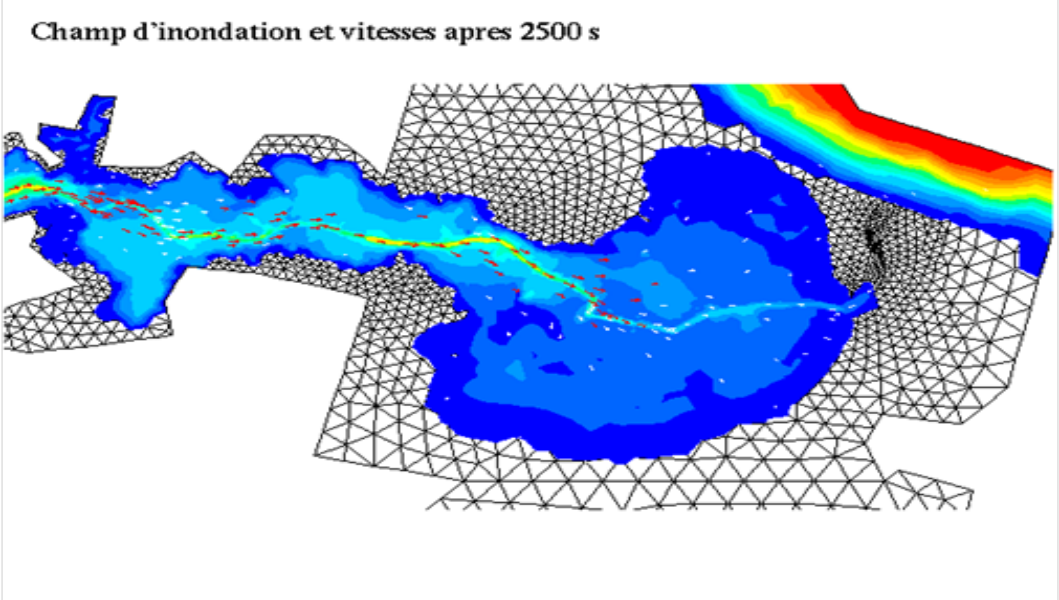
A la suite de l'accident, le Comité Technique Permanent des Barrages (CTPB) a été créé pour l'inspection des barrages de plus de 20 mètres. Aussi, une simulation de l'onde de rupture devrait être fournie. La simulation avec Telemac-2D a montré que le calcul bidimensionnel d'ondes de ruptures ou de la propagation de crues sur des domaines de plusieurs dizaines de kilomètres était réalisable par des applications industrielles. Cette simulation a également montré d'énormes améliorations par rapport aux simulations unidimensionnelles qui étaient faites auparavant. En effet, ce type de simulation prend en compte les cas où les géométries comportent des grandes plaines ou des courbes prononcées et la célérité des ondes de crue ou de rupture y est mieux respectée [Hervouet, 2000]. La *figure* 6.2 issue de cette simulation montre les champs d'inondation et vitesse après 2500 secondes.

6.2.4 Les problèmes numériques recensés dans le code

Comme nous l'avons mentionné plus haut, des études ont montré des différences entre les résultats issus d'exécutions en parallèle et en séquentiel. La *figure* 6.3 compare les listings de sortie de trois exécutions de Telemac-2D (séquentielle, 64 processus, 128 processus) sur la machine Ivanoe (annexe A.3). Le compilateur utilisé est GFortran avec les options -O3

54. Un barrage voûte est reconnaissable à sa forme arquée. Pour en savoir plus voir http://fr.wikipedia.org/wiki/Barrage_voûte

Figure 6.2 – Rupture du barrage de Malpasset : champ d'inondation et vitesse après 2500 secondes (Source LNHE).



-fconvert=big-endian -frecord-marker=4. Sur la *figure* 6.3, les lignes comportant des différences sont matérialisées en violet et les différences en rouge. On remarque des différences dans toutes les variables de sorties du code. Seule la première décimale est commune.

En fait, les codes de la taille de Telemac-2D sont généralement influencés par des instabilités numériques. Ces instabilités s'expliquent principalement par les notions exposées au chapitre 2. Dans le cas traité ici, il est surtout question de reproductibilité entre les versions séquentielles et les versions parallèles. D'après [Moulinec *et al.*, 2011a, Denis *et al.*, 2011], ces différences sont dues aux calculs des valeurs des points d'interface (voir *figure* 6.4). Ces valeurs sont calculées à partir de contributions de plusieurs processus (voir *figure* 6.4b). Les contributions sont assemblées grâce aux opérations de réduction du standard MPI. Il importe de remarquer que les opérations de réduction ne peuvent se faire dans un ordre déterministe (voir *figure* 6.4c). Ajoutons à cette remarque, les propriétés des opérations en arithmétique flottante (non-associativité) et on comprend mieux pourquoi avoir un code qui produit des résultats identiques aux digits près (exécutions en séquentiel comme en parallèle) peut s'avérer être une tache extrêmement difficile.

Les calculs des points d'interface ont lieu pendant le calcul de produits scalaires ou de sommations et les opérations d'assemblage de vecteurs élémentaires [Hervouet, 2003, p201-202]. Nous nous proposons de faire une étude du comportement numérique du code Telemac-2D avec l'outil CADNA (chapitre 3) et de confronter les résultats de cette étude aux conclusions de [Moulinec *et al.*, 2011a, Denis *et al.*, 2011] sur les raisons de la non-reproductibilité des résultats. Les principaux résultats de cette étude sont présentés dans la prochaine section.

6.3 Validation de Telemac-2D avec l'outil CADNA

6.3.1 Implémentation de CADNA dans les codes sources

L'outil CADNA et son fonctionnement ont été présentés dans le chapitre 3. Rappelons juste que cet outil implémente la méthode CESTAC dont l'idée de base est de faire chaque opération arithmétique 3 fois avec un mode d'arrondi choisi aléatoirement. Cette méthode permet ainsi de propager aléatoirement les erreurs d'arrondi et d'en déduire leurs impacts sur le résultat final. Afin de faciliter l'utilisation de CADNA dans les codes industriels, une extension CADNA_MPI pour le standard de communication MPI a été développée (chapitre 4). CADNA utilise des types spéciaux *double_st* pour la double précision. Comme expliqué dans les exemples de la section 3.3, l'implémentation de CADNA dans un code s'effectue en incluant le module CADNA et en remplaçant les types *double precision* par *double_st*. Cette opération de remplacement de types de données peut normalement se faire assez facilement en utilisant le programme *sed* [55] disponible sur toutes les distributions linux. Cependant, nous avons rencontré quelques difficultés dans notre cas. Ces difficultés sont essentiellement dues à quelques spécificités du Fortran. Ces difficultés sont décrites ci-dessous.

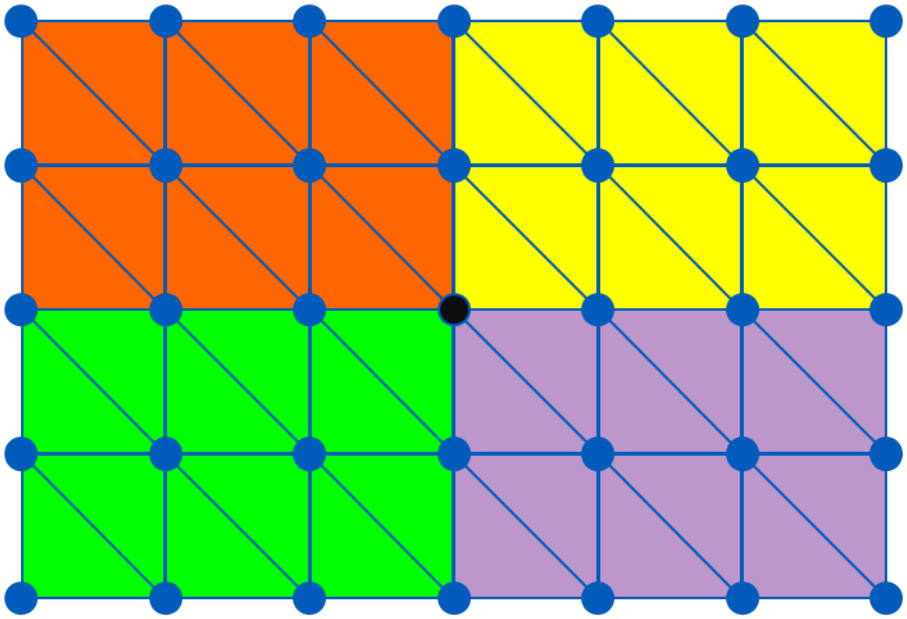
1. Le module de base du système Telemac est la *bief*. Toutes les structures de données utilisées dans tout le système Telemac sont définies dans le module *bief_def* qui est alors appelé dans la *bief*. Le module *bief* est ensuite appelé dans presque tous les autres fichiers sources du système. Normalement pour implémenter CADNA dans le code Telemac-2D, il aurait fallu remplacer uniquement les types standards *double precision* par les types stochastiques *double_st* dans le module *bief_def*, et y appeler le module CADNA (ou

55. sed (abréviation de Stream EDitor) est un programme informatique permettant d'appliquer différentes transformations prédéfinies à un flux séquentiel de données textuelles. voir http://en.wikipedia.org/wiki/Sed

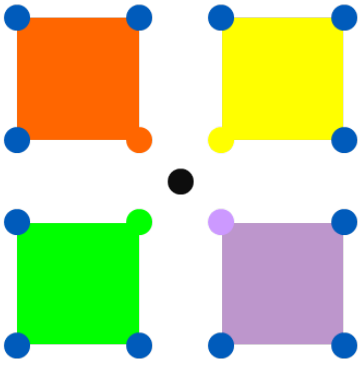
Figure 6.3 – Comparaison de listing de sortie de Telemac-2D : séquentiel, 64 procs et 128 procs

Figure 6.4 – Décomposition de domaine et assemblage aux interfaces [Denis *et al.*, 2011]

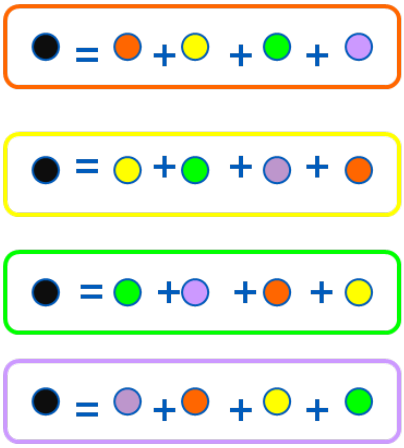
(a) Focus sur un domaine partagé sur 4 processus : la partie de chaque processus est matérialisée par une couleur. Le point d'interface aux quatre sous-domaines est le point noir au milieu



(b) zoom sur les quatre mailles du centre : les valeurs du point d'interface dépendent des valeurs des points locaux. Ces valeurs sont obtenues par réduction (par exemple une somma-tion) des quatre valeurs locales



(c) Ordres possibles pour l'opération de réduction



CADNA_MPI) pour activer l'utilisation de CADNA. En fait, afin d'optimiser le temps de compilation, les modules *bief_def* et *bief* ne sont pas inclus dans tous les fichiers. Par conséquent, le module CADNA a été rajouté à la main dans tous les fichiers qui le nécessitaient.

2. Le Fortran 77 autorise des affectations avec le mot clé *DATA*. Ce type d'affectation n'est pas autorisé pour les structures de données (comme les types stochastiques). Il a donc fallu modifier les assignements de ce type. Cette modification a été faite à la main fichier par fichier.
3. Des difficultés ont également été rencontrées dans l'utilisation des fonctions "intrinsèques" du Fortran (min,max). Les versions d'origine de ces fonctions peuvent avoir plus de deux paramètres contrairement aux versions définies dans CADNA. Ces dernières ne peuvent calculer que le minimum (respectivement le maximum) de deux valeurs alors qu'on peut calculer le minimum de n valeurs avec la fonction min de fortran. Pour cela, nous avons réécrit certains morceaux de codes à la main.

En revanche, l'implémentation de CADNA_MPI a été moins compliquée. Dans le système Telemac-Mascaret, il existe un module *parallel* qui est une surcouche pour les routines de communication. Ce module permet ainsi au code d'être indépendant des bibliothèques de communication. Cette surcouche a par exemple facilité le passage de PVM à MPI. A l'aide du programme *sed*, les MPI_DOUBLE ont été remplacés par des MPI_DOUBLE_ST.

6.3.2 Surcoût dû à l'utilisation de CADNA

Le *tableau* 6.2 présente la durée pour une simulation réalisée sur 40 pas de temps sur la Z600 (annexe A.2). Une exécution de la version normale de Telemac-2D dure environ 1 seconde en exécution séquentielle (nettement moins pour l'exécution parallèle), ce qui permet de mettre en évidence le surcoût en termes de durée d'exécution des simulations utilisant CADNA. Ce même calcul nécessite 76 secondes pour 4000 pas de temps pour une exécution séquentielle. La version intégrant CADNA nécessite quand à elle en séquentielle 8961 secondes pour le mode avec auto-validation et 3360 secondes pour le mode de base de CADNA. On note un rapport d'environ 44 entre les deux temps de calcul des exécutions quand on utilise le mode sans auto-validation de CADNA et un rapport de 117 quand on utilise le mode avec l'auto-validation (ces deux modes de fonctionnement de CADNA ont été définis en section 3.3).

Tableau 6.2 – Surcoût dû à l'utilisation de CADNA pour une simulation sur 40 pas de temps. Les mesures ont été recueillies sur la Z600. Le mode "Avec A-V" indique le mode d'auto-validation est activé. Dans le mode "Sans A-V", l'auto-validation est désactivé.

	Telemac-2D	Telemac-2D CADNA					
		Sans A-V			Avec A-V		
N Proc	1	1	2	4	1	2	4
Temps (s)	1	30	20	10	76	44	28

L'inconvénient principal de l'utilisation de CADNA est donc le surcoût qu'il engendre. Nous avons expliqué les raisons de ce surcoût dans le chapitre 5. Toutefois, nous insistons ici sur l'objectif fondamental des versions de codes intégrant complètement CADNA : ces versions sont dédiées à l'étude de la qualité numérique des résultats produits et dans un second temps au débogage numérique. De fait, ces versions n'étant pas destinées à la production, on peut alors relativiser les surcoûts engendrés.

6.3.3 Le diagnostic de CADNA et son analyse

Le principal apport de l'outil de validation CADNA réside dans le diagnostic qu'il propose à la fin des exécutions. Ce diagnostic est obtenu par l'activation du mode d'auto-validation. Voici ci-dessous le diagnostic proposé pour une simulation de 4000 avec un pas de temps $dt = 1\ seconde$:

```
LISTING DE CADNA  ----------------------------------
--------------------------------------------------
CADNA software --- University P. et M. Curie --- LIP6
CRITICAL WARNING: the self-validation detects major problem(s).
The results are NOT guaranteed
There are    366216713  numerical instabilities
          0  UNSTABLE DIVISION(S)
   13887117  UNSTABLE POWER FUNCTION(S)
   19508999  UNSTABLE MULTIPLICATION(S)
   44635599  UNSTABLE BRANCHING(S)
    6704983  UNSTABLE MATHEMATICAL FUNCTION(S)
       6770  UNSTABLE INTRINSIC FUNCTION(S)
  281473245  UNSTABLE CANCELLATION(S)
--------------------------------------------------
```

Cependant, il nous a été difficile d'obtenir ce diagnostic pour plusieurs raisons. En effet, nous avons remarqué que suivant les paramètres du cas de test (pas de temps et précision), certaines exécutions avec CADNA ne se terminaient pas suite à l'apparition de boucles infinies. En fait, ces boucles infinies sont dues à des zéros informatiques ($@.0$) qui se propagent pendant l'exécution. Rappelons qu'un zéro informatique est obtenu si le nombre de chiffres significatifs d'un résultat est inférieur à 0 ou si tous ses échantillons R_i sont nuls (voir section 3.2.5). Normalement, il aurait fallu procéder l'étude de façon incrémentale : pour chaque instabilité détectée, il faut la corriger avant de relancer l'exécution du code avec CADNA. Cette façon de faire est tout simplement irréalisable. En fait, il est difficile d'exploiter les instabilités détectées à la fin du programme. Par exemple, nous avons remarqué que des zéros informatiques apparaissent assez rapidement au cours des simulations. Les résultats après l'apparition de zéro informatique n'ont aucune crédibilité et les instabilités détectées ne sont pas significatives. C'est d'ailleurs la raison pour laquelle, il est marqué dans le listing CADNA :

```
The results are NOT guaranteed
```

Partant de cette observation, nous nous sommes concentré principalement sur les cinq premiers pas de temps. Les débogueurs Intel (idbc) et GNU (gdb) ont été utilisés pour tracer les instabilités et ainsi trouver les fonctions qui génèrent le plus de problèmes numériques. Le débogage numérique est expliqué en détail dans le prochain chapitre. Les instabilités détectées après un pas de temps sont des cancellations (élimination catastrophique) et des instabilités de branchement. Et puis, progressivement des multiplications et des fonctions mathématiques instables ont été détectées. Rappelons qu'une cancellation est détectée quand la différence entre les chiffres significatifs du résultat et des opérandes est supérieur à 4 (par défaut). Un branchement instable est détecté lorsque la différence entre les deux opérandes est un zéro informatique [CADNA Team, 2010]. Les cancellations détectées proviennent de la fonction *bief_valida*. C'est presque normal puisque cette fonction effectue des comparaisons entre des valeurs très proches.

Finalement, le débogage numérique à l'aide de CADNA a permis de faire l'observation suivante : *plus de* 30% *des instabilités détectées apparaissent dans les produits scalaires*. Les produits scalaires sont utilisés pour calculer le résidu dans les méthodes itératives et dans les résolutions de systèmes linéaires creux. Cette observation confirme les hypothèses de [Denis *et al.*, 2011] sur l'explication des instabilités numériques.

Dans la prochaine section (section 6.4), les différentes solutions pour améliorer la précision des produits scalaires sont présentées. Nous nous intéressons particulièrement aux algorithmes compensés. Nous montrons que ces types d'algorithmes constituent un compromis intéressant pour améliorer la précision des résultats sans détériorer les performances du code.

6.4 Les algorithmes compensés

6.4.1 Introduction

Le produit scalaire est une des tâches les plus élémentaires de l'analyse numérique. Il est intensément utilisé dans les principaux algorithmes d'algèbre linéaire (résolution de système linéaire dans les méthodes directes comme itératives, produit matrice-matrice, matrice-vecteur, etc). Rump propose un excellent aperçu des applications du produit scalaire dans l'introduction de son article [Rump, 2009]. L'article [Li *et al.*, 2002] et le livre [Higham, 2002] présentent de manière plus approfondie ce sujet.

L'ingrédient essentiel du produit scalaire est la sommation de nombres flottants. En fait, comme le montrent les auteurs de [Ogita *et al.*, 2005], le produit scalaire de deux vecteurs de taille n peut être transformé **sans erreur** en une somme d'un vecteur de taille $2n$. Par conséquent, améliorer la précision de la sommation de n éléments permet d'améliorer considérablement la précision de plusieurs applications. Higham consacre d'ailleurs à ce sujet tout un chapitre de son ouvrage [Higham, 2002]. L'importance de ces algorithmes est confirmée par le nombre considérable d'algorithmes de sommation qu'on recense dans la littérature. Les auteurs de [Langlois *et al.*, 2012a] expliquent qu'un nouvel algorithme de sommation flottante est proposé chaque année depuis les premiers travaux de Malcom sur le sujet [Malcolm, 1971].

Un des premiers algorithmes a donc été proposé dans [Malcolm, 1971]. Cet algorithme ainsi que ceux de Kulish et de Demmel et Hilda [Demmel et Hida, 2004] sont basés sur l'utilisation d'accumulateurs longs et la sommation par exposant (ou plage d'exposant). En d'autres termes, pour minimiser les erreurs, les nombres flottants sont répartis en plusieurs tableaux en fonction de leur exposant. La somme est alors effectuée dans un premier temps sur les flottants d'un même tableau (accumulateur), puis on accumule les sommes partielles. La version de Malcom nécessite un long accumulateur avec un tableau pour tous les exposants. Demmel et Hilda ont proposé au lieu d'un tableau pour chaque exposant, un tableau pour un intervalle d'exposant.

Les plus récents algorithmes sont : Sum2 [Ogita *et al.*, 2005], AccSum [Rump *et al.*, 2008], FastAccSum [Rump, 2009], iFastSum et HybridSum [Zhu et Hayes, 2009], OnLineExact [Zhu et Hayes, 2010] et tout dernièrement l'algorithme de sommation reproductible proposé dans [Demmel et Nguyen, 2013].

Sum2 est de la classe des algorithmes précis. Le résultat est aussi précis qu'un résultat calculé avec une précision double. En fait, les calculs sont faits avec la précision de travail mais avec un cumul des erreurs commises qui est ensuite ré-injecté dans le résultat final. On parle alors d'*algorithme de somme compensée*. Ce type d'algorithme est présenté dans la section 6.4.2. Ils ont pour principal inconvénient d'être dépendants du conditionnement du vecteur.

Les autres algorithmes (AccSum, FastAccSum, iFastSum, HybridSum, OnLineExact) sont considérés comme fidèles et correctement arrondis : le résultat calculé est l'un des deux flottants

encadrant le résultat exact (si le résultat est représentable en arithmétique IEEE 754 alors le résultat calculé est le résultat exact ; voir section 2.1.2). Ces algorithmes ont pour objectif d'avoir des résultats précis indépendemment du conditionnement de la somme. L'algorithme iFastSum [Zhu et Hayes, 2009] est basé sur l'itération de la distillation de Kahan, avec un contrôle dynamique de l'erreur résiduelle, pour converger vers un résultat correctement arrondi. La distillation permet de transformer sans erreur une somme $\sum x_i$ en une somme $\sum x_i^*$ plus stable et dont le résultat est plus précis. Le principe de la distillation est expliqué en section 6.4.4. Les algorithmes AccSum [Rump *et al.*, 2008] et FastAccSum [Rump, 2009] sont eux basés sur des extractions successives des opérandes. Les extractions sont faites de manière à ce que les sommes partielles (des parties extraites) soient exactes. HybridSum [Zhu et Hayes, 2009] et OnLineExact [Zhu et Hayes, 2010] combinent les deux approches. D'abord une extraction des exposants des opérandes, qui permet d'accumuler (somme partielle) des opérandes dans un ou deux tableaux dont la taille dépend de la plage des exposants de la norme IEEE 754 (2048 en double précision). Le vecteur initial à sommer est ainsi transformé en un vecteur plus court sur lequel est appliquée la distillation jusqu'à obtention du résultat correct (iFastSum).

L'amélioration de la précision numérique des algorithmes de sommation pourrait également être obtenue en augmentant la précision (taille de la mantisse) initiale des nombres flottants comme c'est le cas dans la bibliothèque MPFR [Fousse *et al.*, 2007]. Basée sur la bibliothèque multi-précision GNU MP, MPFR est une bibliothèque C portable pour le calcul sur les nombres flottants en précision arbitraire avec arrondi correct. L'utilisation de la bibliothèque MPFR ne garantit pas l'exactitude de la sommation contrairement à la bibliothèque MPFI [Revol et Rouillier, 2005] qui combine l'arithmétique d'intervalles et la précision multiple. Cependant, comme signalé en section 2.2.1, le principal inconvénient de l'arithmétique d'intervalles est qu'il surestime l'erreur. Une surcharge en temps de calcul et de mémoire doit être également attendu lors de l'utilisation de ces bibliothèques. En plus de ces surcoûts, il est difficile de les mettre en œuvre dans les codes industriels existants. Ces derniers sont pour la plus grande majorité écrits en Fortran. Par conséquent, le moyen naturel d'avancer est de mettre en œuvre les algorithmes de sommation compensée travaillant à la précision de la machine qui ne sont pas intrusifs, en particulier dans un code industriel.

Dans la suite de ce chapitre, nous nous intéressons aux algorithmes Sum2 et Dot2 présentés dans [Ogita *et al.*, 2005]. Nous avons fait le choix (arbitraire) de travailler avec ces algorithmes parce qu'ils sont efficaces, très simples à implémenter et ne nécessitent que des opérations d'arithmétiques élémentaires. Il convient aussi de rappeler que le travail réalisé ici avait pour but principal de montrer l'intérêt des algorithmes compensés pour les applications industrielles.

6.4.2 La base des algorithmes compensés

Reprenons ici les notations utilisées dans les chapitres 2 et 3. L'ensemble des nombres flottants est désigné $\mathbb{F}$, eps est l'erreur relative due à l'arrondi (unité d'arrondi), et $fl(\cdot)$ correspond au résultat d'une opération en arithmétique flottante réalisée à la précision de travail (simple ou double précision par exemple).

Considérons un vecteur $p \in \mathbb{F}^n$. L'algorithme récursif (classique) de sommation d'un vecteur de n éléments est défini dans l'**algorithme** 6.1. En fait, l'**algorithme** 6.1 calcule $r\hat{e}s = fl(\sum p_i)$. Les limites de l'arithmétique flottante (énoncées en section 2.1) et l'équation 2.4 montrent que la précision de $r\hat{e}s$ dépend du nombre de conditionnement de la somme. Sup-

Alg. 6.1: Algorithme classique de sommation d'un vecteur p de n éléments

$res = p_0$
for i =1 : n-1 **do**
  $res = res + p_i$
end for

posons res le vrai résultat mathématique de $\sum p_i$, il a été montré que l'erreur relative vérifie l'équation suivante [Rump, 2009, section 1] :

$$\left|\frac{\hat{res} - res}{res}\right| \sim eps \cdot cond(\sum p_i) = eps \cdot \frac{\sum |p_i|}{|\sum p_i|} \tag{6.5}$$

En résumé, comme nous l'avons mentionné en section 2.2.1, un nombre de conditionnement élevé implique une perte de précision importante (voire totale) sur les résultats calculés. En fait, le nombre de cancellations susceptibles est proportionnel au nombre de conditionnement. Les algorithmes compensés ont pour objectif de minimiser l'erreur relative à la précision de travail. Notons qu'en double précision IEEE 754, cette précision est de $eps = 2^{-53} \approx 10^{-16}$.

6.4.3 Transformation exacte de l'addition et de la multiplication

Comme mentionné précédemment, la somme de deux flottants a et b n'est généralement pas un nombre flottant mais l'erreur commise en est un. On peut alors écrire :

$$x + y = a + b, \quad \text{avec} \quad x = fl(a + b) \quad \text{et} \quad y \quad \text{un nombre flottant} \tag{6.6}$$

Une telle transformation qui, en utilisant uniquement la précision de travail et des opérations arithmétiques élémentaires permet de calculer x et y en fonction de a et b est appelée *Error-Free Transformation* (EFT) dans [Ogita *et al.*, 2005]. EFT est traduit ici en transformation exacte. L'**algorithme** 6.2 (FastTwoSum) de Dekker [Dekker, 1971], permet de calculer x et y si $|a| \geq |b|$.

Alg. 6.2: =

FastTwoSum(a, b)][x,y] = FastTwoSum(a, b)
  $x = fl(a + b)$
  $y = fl(b - (x - a))$

L'**algorithme** 6.3 (TwoSum) proposé par Knuth [Knuth, 1969] permet d'établir la transformation exacte de l'addition. Contrairement à l'**algorithme** 6.2, il ne nécessite pas de test de branchement. TwoSum nécessite $6 flops$ (floating-point operations).

Alg. 6.3: =

TwoSum(a,b)][x,y]=TwoSum(a,b)
  $x = fl(a + b)$
  $z = fl(x - a)$
  $y = fl((a - (x - z)) + (b - z))$

La transformation exacte de la multiplication a été mise au point par Dekker et Veltkamp dans [Dekker, 1971]. Cette transformation nécessite de découper les deux entrées en deux parties. Soit p la taille de la mantisse à précision courante et $s = \lceil p/2 \rceil$ (en double précision IEEE 754 $p = 53$, $s = 27$), l'**algorithme** 6.4 (Split) de [Dekker, 1971] permet de découper un flottant a en deux flottants x et y.

Alg. 6.4: =

Split(a)][x,y]=Split(a)

$c = fl(factor \cdot a) \qquad //factor = 2^s + 1$
$x = fl(c - (c - a))$
$y = fl(a - x)$

A partir de cet algorithme, la transformation exacte de la multiplication est formulée dans l'**algorithme** 6.5 (TwoProduct). Cette transformation nécessite $17 flops$ mais en utilisant une FMA (section 2.1.4) elle ne nécessitera que $2 flops$. En effet quand le FMA est disponible, l'erreur commise y peut être calculer directement par l'opération $y = a \times b - x$. L'**algorithme** 6.5 peut alors être remplacer par l'**algorithme** 6.6.

Alg. 6.5: =

TwoProduct(a,b)][x,y]=TwoProduct(a,b)

$x = fl(a \times b)$
$[ah, al] = Split(a)$
$[bh, bl] = Split(b)$
$y = fl(al \times bl - (((x - ah \cdot bh) - al \cdot bh) - ah \cdot bl))$

Alg. 6.6: =

TwoProductFMA(a,b)][x,y]=TwoProductFMA(a,b)

$x = fl(a \times b)$
$y = FMA(a, b, -x)$

Notons que l'**algorithme** 6.5 ne nécessite aucun test de branchement et n'utilise que des opérations élémentaires d'arithmétique. En absence d'underflow, Dekker montre dans [Dekker, 1971] :

$$x + y = a \cdot b, \quad \text{avec} \quad x = fl(a \cdot b) \tag{6.7}$$

A partir des algorithmes 6.5 et 6.3, les algorithmes de sommation et de produit scalaire compensés sont définis dans la prochaine section.

6.4.4 Algorithme de produit scalaire compensé

L'**algorithme** 6.3 (TwoSum) permet de calculer l'addition de deux nombres flottants. On peut alors, à partir de ce dernier, calculer une bonne approximation de $\sum p_i$. [Ogita *et al.*, 2005] propose de faire une itération de TwoSum et de cumuler les erreurs pour améliorer la qualité du résultat final de $fl(\sum p_i)$. Associer en cascade des briques de la fonction TwoSum constitue la

transformation exacte de la sommation de n flottants. Elle se fait avec l'**algorithme** 6.7 (VecSum) ci-dessous :

Alg. 6.7: p = VecSum(p)

for i =2 to n **do**
$[p_i, p_{i-1}] = TwoSum(p_i, p_{i-1})$
end for

Le vecteur p est transformé sans modification de la somme $\sum p_i$ et p_n est remplacé par $fl(\sum p_i)$. Cette transformation a été appelée algorithme de distillation par Kahan [Kahan, 1987]. A partir de l'algorithme VecSum, on obtient l'algorithme de sommation compensée (**algorithme** 6.8).

Alg. 6.8: =

Sum2(p)][res]=Sum2(p)
for i =2 to n **do**
$[p_i, p_{i-1}] = TwoSum(p_i, p_{i-1})$
end for
$res = fl((\sum_{i=1}^{n-1} p_i) + p_n)$

Le produit scalaire de deux vecteurs x et y est, en fait, la somme des produits $x_i \cdot y_i$. La combinaison de l'**algorithme** 6.8 et l'**algorithme** 6.5, permet de définir un algorithme de produit scalaire compensé [Ogita *et al.*, 2005]. Notons que Sum2 nécessite $7(n-1)flops$ et Dot2 $25n - 7flops$. L'algorithme classique de produit scalaire (Dot) se fait en $2n - 1flops$.

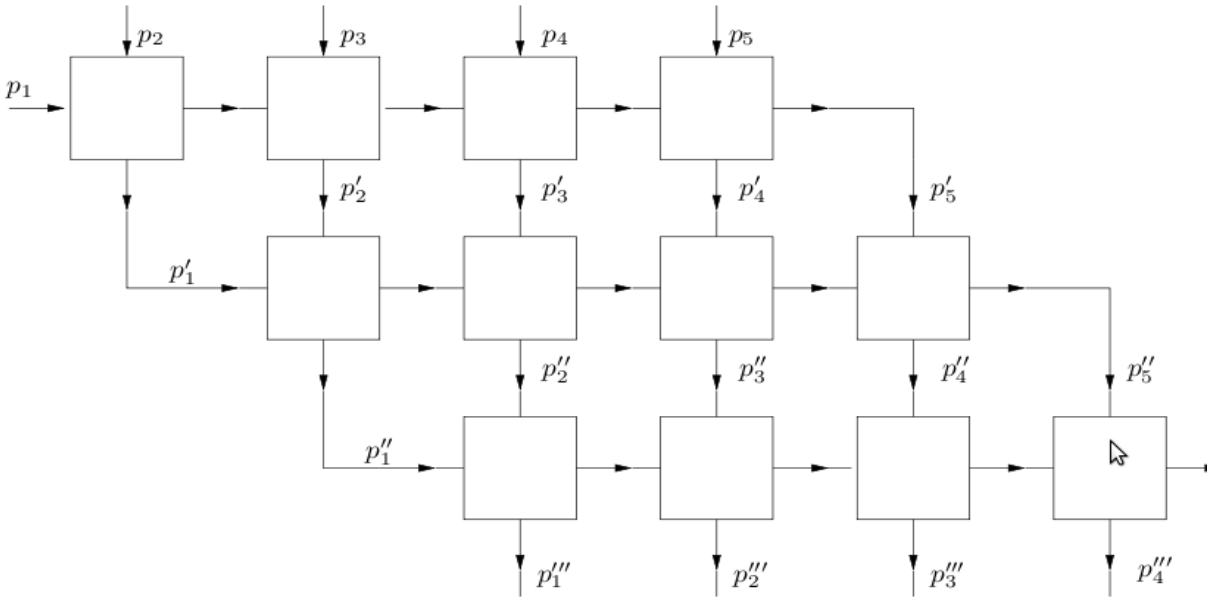
Alg. 6.9: =

Dot2(x, y)][res]=Dot2(x, y)
$[p, s] = TwoProduct(x_1, y_1)$
for i =2 to n **do**
$[h, r] = TwoProduct(x_i, y_i)$
$[p, q] = TwoSum(p, h)$
$s = fl(s + (q + r))$
end for
$res = fl(p + s)$

Les deux algorithmes compensés : Sum2 et Dot2 présentés ci-dessus permettent d'améliorer la précision des sommations et du produit scalaire classique. En fait, ces algorithmes améliorent le résultat calculé en simulant le double de la précision de travail utilisé. Ogita et al montrent que si la sommation des p_i n'est pas mal conditionnée (c'est à dire pour un nombre de conditionnement inférieur à eps^{-1}, l'**algorithme** 6.8 (Sum2) est largement suffisant. En d'autres termes, pour une sommation d'un vecteur p tel que $\frac{\sum |p_i|}{|\sum p_i|} < eps^{-1}$, le résultat obtenu avec Sum2 est ce qu'on peut avoir de plus précis. En revanche, pour les vecteurs possédant des nombres de conditionnement plus grand (donc très mal conditionné), rien n'est garanti. Dans ce cas, Ogita et al proposent que le vecteur p soit transformé avec VecSum en un vecteur p' où p'_n est le résultat de l'algorithme classique de sommation $fl(\sum p_i)$. En fait, VecSum transforme

sans erreur le vecteur p mal conditionné en un nouveau vecteur p' avec une somme identique mais ayant un nombre de conditionnement amélioré d'un facteur eps et donc un résultat d'une meilleure précision (le double de la précision initiale). C'est le principe le transformation exacte en cascade de vecteur de Ogita et al (voir *figure* 6.5). A partir de cette transformation, ils introduisent les algorithmes compensés $Kfois$ (avec $K-1$ transformations exactes) SumK (**algorithme** 6.10) et DotK (**algorithme** 6.11). Ces algorithmes simulent $Kfois$ la précision de travail. Par exemple, pour $K=2$, SumK est équivalent à Sum2.

Figure 6.5 – Transformation exacte en cascade de vecteur proposée dans [Ogita *et al.*, 2005]



Alg. 6.10: =

SumK(p, K)[res]=SumK(p, K)
  $[p, r_1] = TwoProduct(x_1, y_1)$
  for k =1 : K-1 **do**
    $p = VecSum(p)$
  end for
  $res = fl((\sum_{i=1}^{n-1} p_i) + p_n)$

Les algorithmes présentés dans cette section reposent sur de solides démonstrations mathématiques. Nous avons choisi de les présenter ici sans insister sur leurs démonstrations. Comme nous l'avons mentionné plus tôt, notre objectif ici est de faire découvrir ces algorithmes et de montrer leurs intérêts. Le lecteur intéressé pourra lire les publications de Rump à ce sujet. Nous conseillons en particulier l'article [Ogita *et al.*, 2005] qui explique clairement et démontre les fondements des algorithmes compensés. Les performances des algorithmes compensés sur les nouvelles architectures ont été étudiées dans [Langlois *et al.*, 2012b].

Alg. 6.11: =

DotK(x, y, K)[res]=DotK(x, y, K)
$[p, r_1] = TwoProduct(x_1, y_1)$
for i =2 : n **do**
$[h, r_i] = TwoProduct(x_i, y_i)$
$[p, r_{n+i-1}] = TwoSum(p, h)$
$s = fl(s + (q + r))$
end for
$r_{2n} = p$
$res = SumK(r, K - 1)$

6.4.5 Expérimentation des algorithmes compensés de produit scalaire.

Dans cette section, les algorithmes compensés de produit scalaire DotK présentés en section 6.4.4 ont été expérimentés. Les tests ont été réalisés sur la machine HP Z600 (annexe A.2). Le compilateur GCC a été utilisé avec les options -march=native -O3 -fomit-frame-pointer -funroll-loops. Nous avons analysé les performances aussi bien en termes de temps d'exécution que de précision numérique. Nous comparons l'erreur relative des résultats de l'algorithme classique de produit scalaire aux algorithmes compensés $Kfois$. Les tests ont été effectués sur des vecteurs aléatoires extrêmement mal conditionnés générés par l'algorithme GenDot [Ogita *et al.*, 2005].

La figure 6.6 présente l'évaluation de l'erreur relative en fonction du nombre de conditionnement. Quand l'erreur relative est supérieure à 2 (ce qui veux dire que le résultat est complètement faux), nous la fixons à 2. Cette figure confirme les explications de la section précédente : en augmentant K, le calcul devient plus précis. Par exemple dans le cas de Dot2, on remarque que l'erreur est inférieure à 1 tant que le conditionnement est inférieur à 10^{16} ($\approx eps^{-1}$). La même observation est faite pour Dot3 tant que le conditionnement est inférieur à 10^{32} ($\approx (eps^2)^{-1}$).

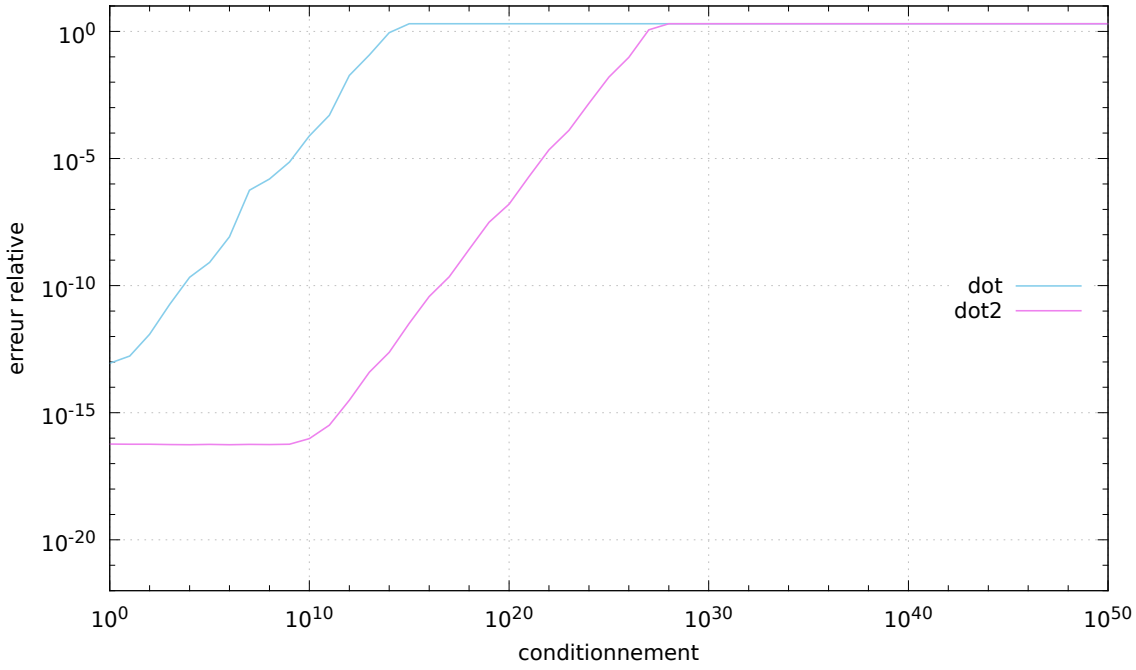
Nous comparons également le temps d'exécution de différentes implémentations (Netlib BLAS, XBLAS, MPFR, MPFI) de la fonction Dot et les algorithmes compensés DotK ($K = 2, 3, 4, 5$). Les résultats sont présentés dans la *figure* 6.7. Dans le *tableau* 6.4, les temps d'exécution du produit scalaire classique ont été normalisés à 1, ce qui met en évidence les rapports entres les différentes implémentations. On remarque que Dot2 est un bon compromis entre performance et précision. Le facteur 7 pour Dot2 est un peu surprenant puisque le dot classique se fait en $2n - 1$ opérations et Dot2 en $25n - 7$ opérations. On devait espérer un facteur 12.5. Cette différence s'explique par deux raisons. D'abord, les algorithmes compensés sont très adaptés aux caractéristiques des machines actuelles. Le compilateur arrive assez facilement à extraire le niveau de parallélisme. A cette première explication, s'ajoute la non linéarité entre le nombre de flops d'une fonction et la durée d'exécution de cette fonction [Langlois *et al.*, 2012b].

Tableau 6.3 – Comparaison de divers implémentations de la fonction Dot : les temps d'exécution du Dot classique ont été normalisés à 1.

Taille	dot2	cblas	xblas	mpfr	mpfi
1000	7.83333	1.08333	13.8611	919.25	1705.92
5000	7.16837	1.0051	12.5816	854.546	1572.59
10000	7.44444	1.01058	13.037	896.96	1612.32

Figure 6.6 – Comparaisons des algorithmes de produit scalaire $K fois$ compensés. L'erreur relative est fixée à 2 quand elle est supérieure à 2. On remarque que la précision du résultat dépend du nombre de conditionnement.

(a) Dot classique et Dot2



(b) DotK ($K = 2, 3, 4, 5$)

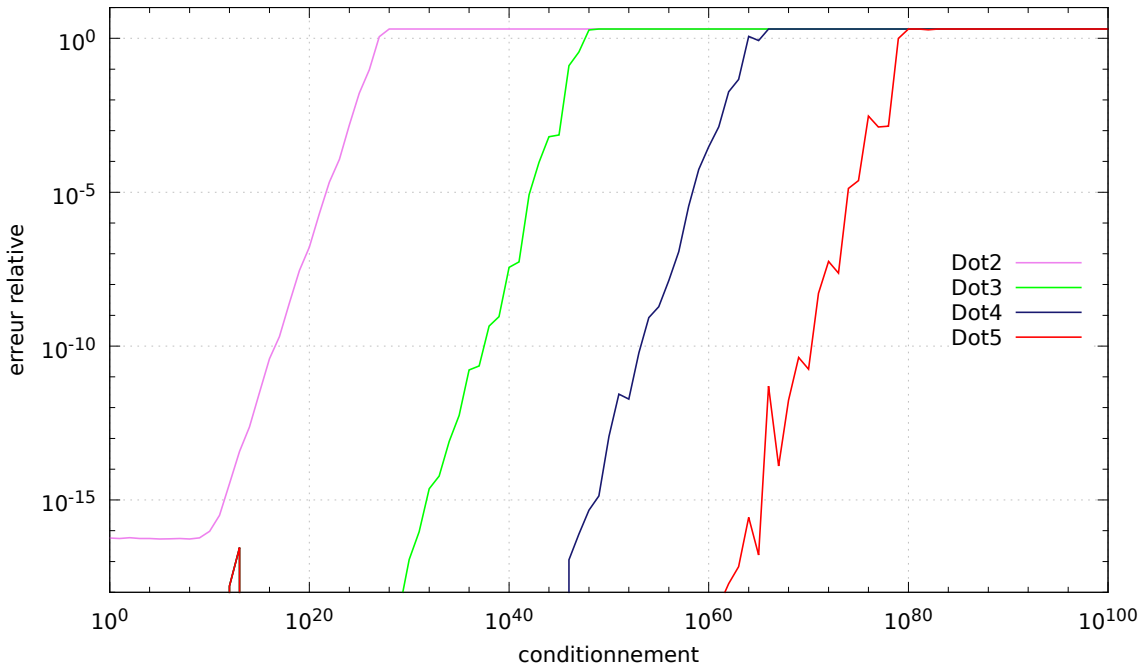
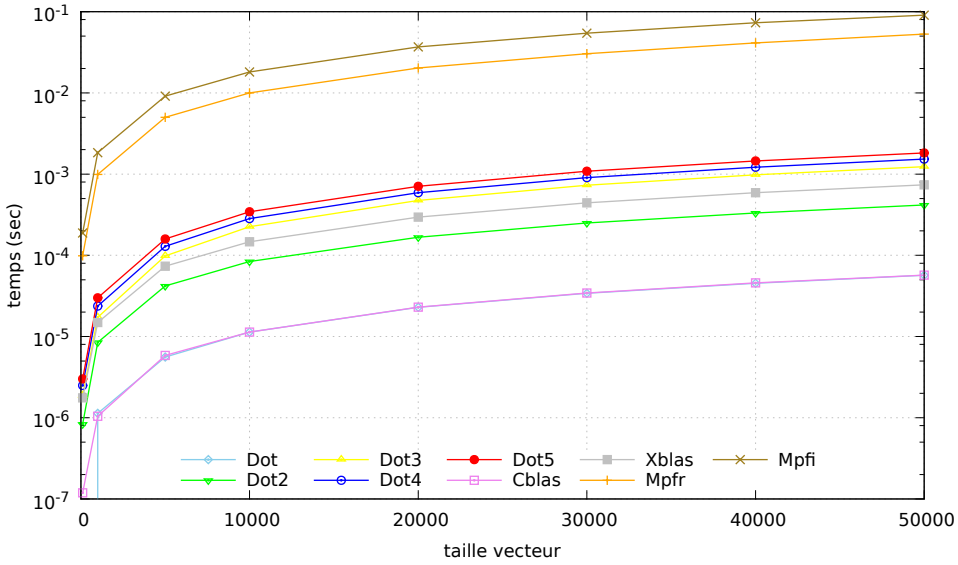


Figure 6.7 – Comparaison de diverses implémentations de la fonction Dot.



6.4.6 Performance de Telemac avec les algorithmes compensés

Les algorithmes compensés ont été implémentés à deux endroits du code Telemac-2D :

1. la fonction Dot classique a été remplacée par une fonction Dot2 de l'**algorithme** 6.9 ;
2. Sum2 a également été implémenté dans la fonction *Paraco* de Telemac-2D. Cette fonction permet d'effectuer l'assemblage de contributions provenant de divers processus. Dans la version initiale, afin de s'assurer que les valeurs d'interface soient identiques sur chaque processus, on procède de la façon suivante : le calcul d'un point d'interface commence par une sommation classique en parallèle sur tous les processus le contenant puis la valeur maximum des sommations est retenue. En termes d'implémentation, cette opération est effectuée en faisant appel deux fois de suite à la fonction Paraco. Nous avons supprimé cette opération. Dans notre version, le calcul des points d'interface est fait uniquement avec Sum2.

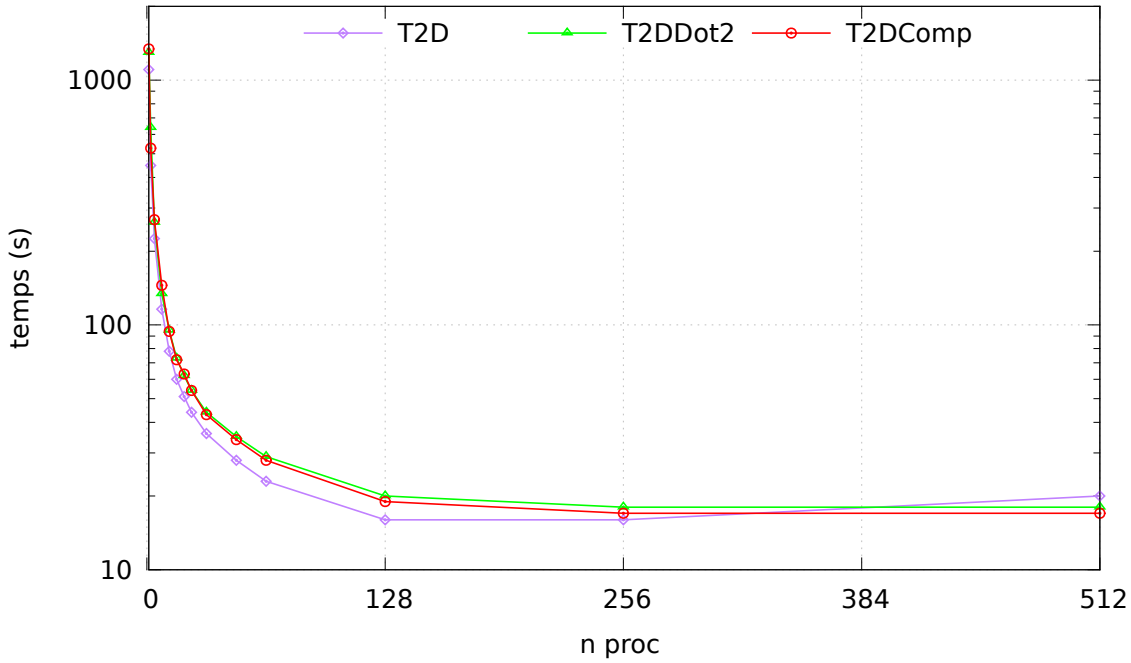
Dans le cadre de ce travail, trois versions du code Telemac-2D sont comparées : T2D est la version normale de Telemac, T2DDot2 une version dans laquelle le produit scalaire classique a été modifié par l'**algorithme** 6.9 (Dot2) et une dernière version T2DComp dans laquelle le produit scalaire et les sommations réalisées lors des assemblages ont été remplacés par Sum2 et Dot2. Les performances de ces trois versions sont évaluées aussi bien en termes de temps d'exécution que de qualité numérique (reproductibilité).

Sur la durée des exécutions

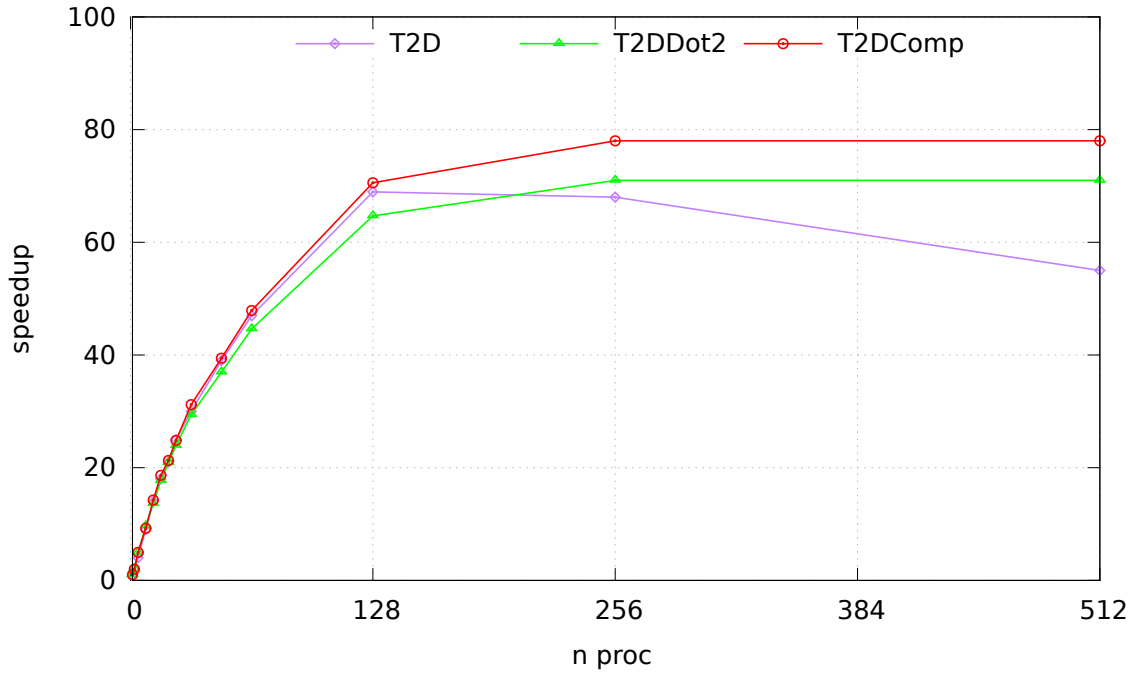
La *figure* 6.8 présente les performances (temps d'exécution et accélération) des trois versions précédemment citées. Les courbes *figure* 6.8a et *figure* 6.8b montrent que l'utilisation des algorithmes compensés ne dégrade pas la performance des codes. Il est par exemple difficile de noter quelque différence sur la *figure* 6.8a. En revanche, on observe une très légère différence entre les trois courbes d'accélération (voir *figure* 6.8b). On remarque que $T2DComp \approx T2Dot2 > T2D$ en termes de temps d'exécution et $T2DComp > T2D > T2DDot2$ quand on considère les accélérations.

Figure 6.8 – Performance de Telemac-2D avec les algorithmes compensés : T2D, T2DDot2, T2DComp.

(a) Temps d'exécution (échelle semi-logarithmique)



(b) Speedup



Les temps d'exécutions T2DComp et T2Dot2 sont très proches parce qu'ils exécutent à peu près le même nombre d'opérations en arithmétique flottante (flops). En fait, la version initiale fait appel à Dot (produit scalaire classique) et la fonction Paraco est utilisée deux fois pour les points d'interface. Dans T2DDot2, la fonction Dot est remplacée par Dot2. Dans T2DComp, outre le remplacement de Dot par Dot2, Sum2 remplace les deux appels à la fonction Paraco. Toutefois, on pouvait imaginer un important facteur entre les versions compensées (T2DComp, T2Dot2) et T2D. On n'observe finalement qu'une différence de 2 à 3% en termes de temps d'exécution. Rappelons que la fonction Sum2 nécessite $7(n-1)flops$ et que Dot2 nécessite $25n-7flops$ alors que la somme classique Sum nécessite $n-1flops$ et le Dot classique nécessite $2n-1flops$. En fait, ces performances sont dues à une utilisation intelligente des algorithmes compensés. Seules les fonctions susceptibles de générer des instabilités numériques ont été modifiées. Il faut ajouter à cela le fait que les algorithmes compensés s'optimisent très facilement sur les architectures actuelles [Langlois *et al.*, 2012b].

En ce qui concerne les accélérations, T2DComp est légèrement supérieure à T2D pour un nombre de processus inférieur à 128 (voir *figure* 6.8b). En fait, l'utilisation des algorithmes compensés entraîne une augmentation du nombre de flops. T2DComp profite mieux du parallélisme : elle effectue plus d'opérations pour une durée identique. C'est la raison pour laquelle pour un nombre de processus supérieur à 128, les versions T2DComp et T2DDot2 sont plus efficaces.

Sur la qualité numérique

Nous comparons ici trois exécutions des trois versions faites sur la machine Ivanoe en séquentiel, avec 64 processus et 128 processus. Les figures 6.9, 6.10 et 6.11 comparent respectivement les sorties des trois exécutions pour les versions T2D, T2DDot2 et T2DComp. Dans la *figure* 6.12, nous comparons les exécutions T2D séquentiel, T2D 128 procs, T2DDot2 128 procs et T2DComp 128 procs.

Malgré nos efforts, on remarque qu'il n'y a toujours pas de reproductibilité des résultats entre les exécutions séquentielles et parallèles. En fait, ce résultat était prévisible puisque nous nous sommes occupé que des 30% des instabilités détectées par CADNA. Cependant, en analysant les sorties, on remarque que la version T2DComp est la plus "reproductible" : deux décimales communes aux différentes exécutions (voir *figure* 6.11). Cette version présente plus de décimales communes dans les sorties comparée à la version normale de Telemac (voir *figure* 6.9). Quelques similitudes sont aussi à remarquer entre T2DComp 128 procs et T2D 128 procs.

En fait, nous avons analysé ici les résultats de la phase de validation de Telemac-2D. Les valeurs présentées sont la différence entre les résultats de référence (issus du fichier de référence du cas de test) et les résultats calculés. On peut alors considérer que plus la différence est petite plus la simulation est précise. Considérons alors la somme des différences de chaque exécution (voir *tableau* 6.4). La version qui présente les plus petites différences est la T2D.

De ces évaluations, il ressort que la version T2DComp est un excellent compromis entre performance et précision. D'une part, elle est plus performante que T2D d'un point de vue accélération. En sus, elle est plus reproductible. Elle fait gagner une décimale par rapport à T2D. En revanche, il apparaîtrait que la version T2D est la plus fiable si **on se fie à la procédure de validation et au fichier de référence**. La version T2DComp calcule des résultats plus proches des valeurs de référence comparée aux autres versions. Insistons particulièrement sur ce dernier

Figure 6.9 – Comparaison du listing de sortie de Telemac-2D : séquentiel, 64 procs et 128 procs

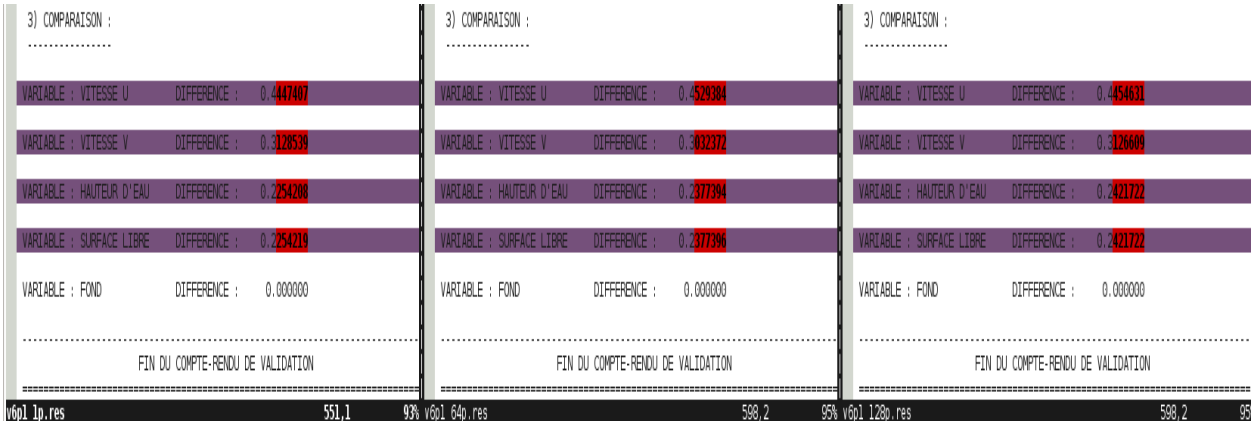


Figure 6.10 – Comparaison du listing de sortie de Telemac-2D (T2DDot2) : séquentiel, 64 procs et 128 procs

Figure 6.11 – Comparaison du listing de sortie de Telemac-2D (T2DComp) : séquentiel, 64 procs et 128 procs

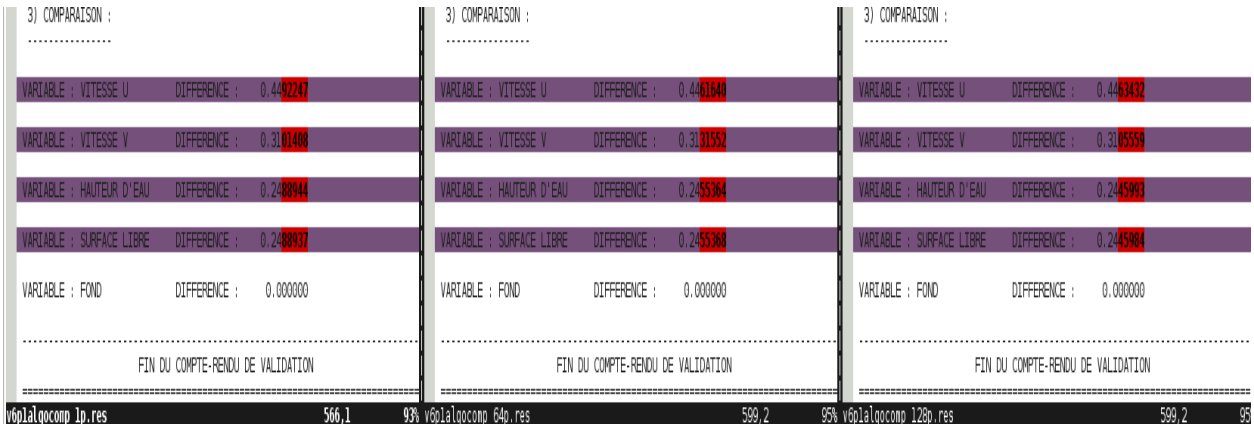


Figure 6.12 – Comparaison du listing de sortie de Telemac-2D : T2D séquentiel, T2D 128 procs, T2DDot2 128 procs et T2DComp 128 procs

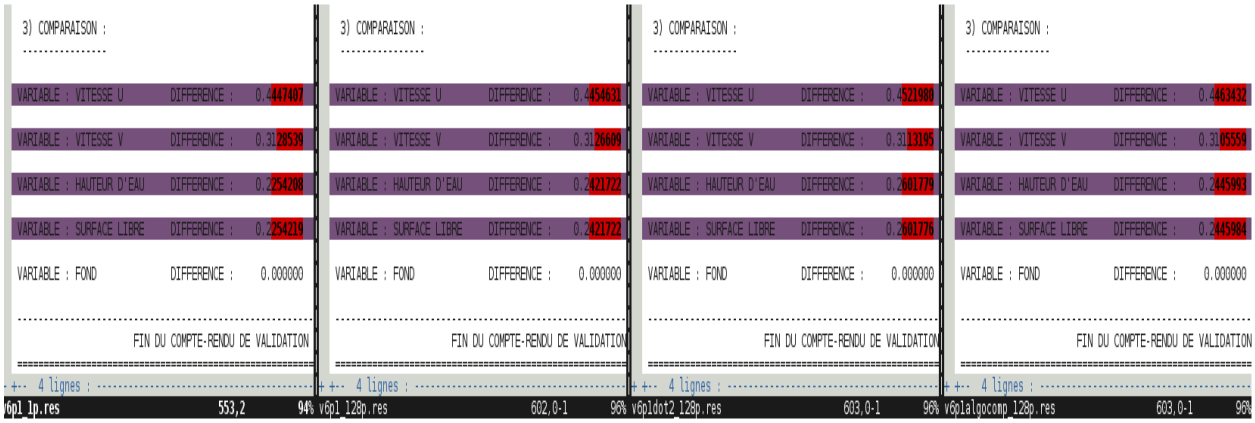


Tableau 6.4 – Comparaison de la somme des différences issues de la phase de validation pour chaque version du code en fonction du nombre de processus

1	64	128	256	512
T2D				
1.2084373	1.2316546	1.2424684	1.25377112	1.2259212
T2DDot2				
1.2571536	1.2581771	1.283873	1.2405839	1.2132397
T2DComp				
1.2571536	1.2503924	1.2460968	1.2345325	1.2345953

point car la crédibilité de cette observation repose sur la façon dont les résultats de référence ont été conçus.

6.5 Conclusion

Le travail présenté dans ce chapitre a été consacré à l'étude de la qualité numérique du code Telemac-2D à l'aide de l'outil de validation numérique CADNA. Cette étude a montré que plus de 30% des instabilités numériques provenaient des produits scalaires. Cette observation a permis de confirmer les hypothèses des premiers travaux réalisés sur le code [Denis et Montan, 2012, Moulinec *et al.*, 2011a]. Nous avons alors présenté les algorithmes de produit scalaire compensé que nous avons implémentés à des endroits particuliers du code.

Ce travail a ainsi permis de montrer que :

i) l'utilisation des algorithmes compensés n'influence presque pas les performances du code ;

ii) l'utilisation des algorithmes compensés permet d'avoir des résultats plus reproductibles.

En somme, le travail présenté dans ce chapitre montre qu'il est possible d'améliorer la qualité numérique des codes sans dégrader les performances de ces derniers. Il permet aussi de confirmer que l'outil CADNA est adapté aux études industrielles, mais il est nécessaire de développer une plate-forme industrielle de vérification numérique basée sur CADNA. Le retour d'expérience de ce travail nous a permis de mettre en place une procédure de validation des

codes industriels à EDF. Dans le prochain chapitre, nous décrivons cette procédure et quelques idées pour une plate-forme de vérification numérique industrielle basée sur le logiciel CADNA.

Chapitre 7

Retour sur l'implémentation de CADNA dans les codes industriels

Sommaire

Au cours de nos travaux, l'outil de validation numérique CADNA a été implémenté dans plusieurs codes parmi lesquels Telemac-2D (chapitre 6) et Telemac-3D [Denis, 2012, Denis et Montan, 2012]. L'objectif de ce chapitre est de faire un retour d'expérience succinct de notre utilisation industrielle.

Plan du chapitre : Nous commençons par proposer une procédure pour l'étude de la qualité numérique des codes de calcul industriels avec l'outil CADNA (section 7.1). Les sections 7.2 et 7.3 sont consacrées à l'utilisation générale de l'outil : d'abord l'implémentation de l'outil dans un code et ensuite l'analyse des diagnostics CADNA et le débogage numérique . La section 7.4 traite de la question de l'évolution ou l'amélioration de CADNA pour une utilisation industrielle. Quelques pistes sont proposées.

7.1 Comment valider un code avec CADNA ?

Tout d'abord, il convient de souligner que cette procédure répond à un besoin de l'entreprise EDF. Au chapitre 6, nous avons présenté nos travaux sur l'étude de la qualité numérique du code Telemac-2D. Dans ce travail, nous avons voulu, dans un premier temps, analyser d'une seule traite tout le code sur une simulation complète. Cette envie s'est heurtée à l'apparition des zéros informatiques et des NaNs au cours des simulations. En fait, l'apparition de ces valeurs rend obsolète le diagnostic CADNA. C'est pourquoi, le manuel d'utilisation de CADNA propose d'effectuer le débogage numérique de manière incrémentale (à chaque instabilité détectée, il faut la corriger avant de poursuivre l'étude). Cette façon de faire n'étant pas très adaptée à notre cas (un nombre important d'instabilités détectées), nous nous sommes concentré sur les premiers pas de temps. Ce travail a permis d'identifier les voies et moyens pour exploiter efficacement l'outil CADNA. Nous résumons en quelques points une procédure de validation de qualité numérique avec CADNA. Nous détaillons certains points de la procédure dans les sections à venir. Cette procédure est un complément au manuel d'utilisateur du logiciel.

1. Déterminer l'architecture du code : il est important de comprendre les interactions entre les différents fichiers sources et les fichiers de définition ; ceci permet de déterminer les principaux fichiers à modifier par la suite.
2. Implémenter CADNA (voir section 7.2) : il faut se baser sur l'architecture du code et le langage de programmation.
3. Implémenter un post-traitement ou un suivi des principales variables du codes (voir section 7.2) : l'idée ici est d'implémenter des petites routines qui permettent de suivre les variables importantes. On pourrait par exemple faire attention à l'évolution du nombre de chiffres significatifs d'une variable donnée au cours des exécutions. L'utilisation des diagnostics intermédiaires est également conseillée. Ces diagnostics peuvent être obtenus en faisant appel à la fonction *cadna_end()* à divers endroits du code. Ceci permettra un suivi de l'évolution des instabilités numériques détectés.
4. Exécution avec CADNA : nous conseillons plusieurs exécutions en faisant varier les paramètres d'initialisation de CADNA.
5. Analyse des diagnostics CADNA en fonction des paramètres $cadna\_instability$ et $cancel\_level$ (section 7.3).

6. Débogage numérique (voir section 7.3) : utiliser les débogueurs pour avoir la trace des instabilités détectées, vérifier le contenu des variables concernées par l'instabilité.

7. Corrections des problèmes numériques.

7.2 L'implémentation de CADNA dans les codes

L'implémentation de CADNA dans un code (ou *cadnatisation* [56] d'un code) peut s'avérer facile ou complexe suivant la taille du code, l'architecture du code et le langage de programmation utilisé. Une façon, un peu brute, mais efficace de procéder est de remplacer tous les types standards (*double*, *float*) par des types stochastiques (*double_st*, *float_st*) dans tous les fichiers sources, d'inclure les modules CADNA et de modifier le programme principal du code en y appelant les fonctions d'initialisation et de terminaison. Le remplacement des types peut se faire avec un script shell (voir extrait de code 7.1) en utilisant la fonction *sed* de Linux.

Source 7.1 – Exemple de script pour remplacement des types

```
#!/bin/bash
for file in *.c
do
echo 'Traitement de $file ...'
sed -e 's/double/double_st/g' "$file" > "$file".tmp && mv -f "$file".tmp "$file"
sed -e 's/float/float_st/g' "$file" > "$file".tmp && mv -f "$file".tmp "$file"
sed -e 's/MPI_DOUBLE/MPI_DOUBLE_ST/g' "$file" > "$file".tmp && mv -f ↩
    "$file".tmp "$file"
sed -e 's/MPI_FLOAT/MPI_SINGLE_ST/g' "$file" > "$file".tmp && mv -f ↩
    "$file".tmp "$file"
```

Un code écrit en C++ est, par exemple, très facile à *cadnatiser* à cause du polymorphisme (utilisation des "templates") que propose ce langage. En revanche, pour un code écrit en Fortran, certaines parties devront être modifiées à la main. Pour ses besoins internes, EDF a conçu un outil python qui permet une *cadnatisation* automatique de code Fortran.

Il importe aussi de connaître l'architecture globale du code. Bien souvent, il n'est pas nécessaire d'implémenter CADNA dans tous les modules (fichiers). Par exemple, dans le cas du code Telemac-2D, nous ne l'avons implémenté que dans les modules de base *bief*, *parallel* et *telemac2d*. Pour cette raison, il est préférable de confier la *cadnatisation* à un développeur du code ou à un ingénieur qui maîtrise l'architecture du code et les interactions entre les différents modules.

En résumé, le principe de l'implémentation de CADNA est simple à mettre en œuvre mais elle dépend essentiellement de la structure du code. Toutefois, cette implémentation est limitée par l'utilisation d'outils externes par les codes. Le travail présenté dans le chapitre 5 a été fait dans ce sens, mais nous ne nous sommes intéressé qu'à une seule routine d'une bibliothèque. Certains codes proposent la possibilité de désactiver les outils externes. Dans ces cas, il est possible de valider le code avec CADNA. Dans le cas contraire, seules les parties du code ne faisant pas appel à des bibliothèques externes peuvent être validées. Nous parlerons dans ces cas de qualité numérique locale (pour des parties du code) et globale (pour le code entier). La qualité globale dépend des qualités locales.

56. abus de langage qui désigne l'implémentation de CADNA dans un code

7.3 Le débogage numérique et l'analyse du diagnostic

Le diagnostic CADNA permet de mesurer la qualité numérique du code. Rappelons que CADNA permet de détecter sept types d'instabilités différents. Ci-dessous un exemple de diagnostic :

```
-------------------------------------------------
CADNA software --- University P. et M. Curie --- LIP6
There are        53584  numerical instabilities
          0  UNSTABLE DIVISION(S)
          0  UNSTABLE POWER FUNCTION(S)
          0  UNSTABLE MULTIPLICATION(S)
      39687  UNSTABLE BRANCHING(S)
          0  UNSTABLE MATHEMATICAL FUNCTION(S)
       6872  UNSTABLE INTRINSIC FUNCTION(S)
       7025  UNSTABLE CANCELLATION(S)
-------------------------------------------------
```

Après une première exécution avec CADNA, nous conseillons de se concentrer sur chaque catégorie d'instabilités détectées. La fonction *cadna_init(numb_instability, cadna_instability, cancel_level, init_random)* (respectivement *cadna_mpi_init())* permet de spécifier le type d'instabilité que l'on souhaite détecter. Il importe de porter une attention particulière aux instabilités de branchement et aux divisions instables. Les deux paramètres forts intéressants à utiliser sont $cadna\_instability$ et $cancel\_level$.

CADNA signale une cancellation si la différence entre le nombre de chiffres significatifs des opérandes et celui du résultat est supérieur ou égal à $cancel\_level$. Par défaut, $cancel\_level = 4$. Initialiser ce paramètre à 7 (en simple précision) ou 15 (en double précision) permet alors de détecter toutes les opérations avec perte totale de précision. Nous conseillons, par exemple, de comparer les différents diagnostics en fonction de la valeur de $cancel\_level$. Si le code est itératif, on peut porter une attention particulière au nombre de cancellations détectées en fonction du nombre de pas de temps.

Le paramètre $cadna\_instability$ permet de définir les types d'instabilités à détecter. Par défaut, l'outil détecte toutes les instabilités. L'extrait de code 7.2 propose quelques exemples de valeur pour $cadna\_instability$ pour ne détecter uniquement qu'un type d'instabilité (branchement, fonctions intrinsèques, fonctions mathématiques).

Source 7.2 – Exemple de valeurs pour cadna_instability

```
!ONLY BRANCHING
      instab=cadna_division+cadna_power+cadna_mathematic
      instab= instab +cadna_intrinsic+cadna_cancellation

!ONLY INTRINSIC
     instab=cadna_division+cadna_power+cadna_branching
      instab = instab +cadna_mathematic+cadna_cancellation

!ONLY MATH
      instab=cadna_division+cadna_power+cadna_branching
      instab = instab+cadna_intrinsic+cadna_cancellation

      CALL CADNA_INIT(-1,instab,15,51)
```

Pour chaque instabilité détectée, il est possible de la tracer et de trouver la ligne de code source qui la produit. C'est ce que nous appelons *débogage numérique*. Pour cela, nous utiliserons un débogueur. Le débogueur conseillé dans la documentation officielle est celui de GNU (gdb). Nous conseillons d'utiliser le débogueur d'Intel (idb, idbc en version shell). En effet, nous avons observé quelques bugs liés aux versions de gdb. Nous proposons ci-dessous deux scripts qui permettent de tracer toutes les instabilités dans un code. La ligne "bt 3" dans l'extrait de code 7.3 permet de signifier le niveau de trace souhaité. Des exemples d'utilisation des scripts sont présentés dans l'extrait de code 7.5.

Source 7.3 – Exemple de script pour idbc

```
set $maxlines=300000
print $maxlines
break instability_
run
while 1
bt 3
cont
end
```

Source 7.4 – Exemple de script gdb

```
break instability_
run
while 1
where
cont
end
```

Source 7.5 – Exemple de lancement des scripts (idb.in script pour idbc et gdb script pour gdb)

```
idbc out22382_intel_64_11.exe <idb.in >> trace_instability.out &

gdb out22382_intel_64_11.exe <gdb.in >> trace_instability.out &
```

Nous présentons ci-dessous un exemple de trace d'instabilité. Deux informations peuvent être déduites de cette sortie. D'abord, on remarque que l'instabilité détectée est produite par la fonction *bief_valida()*à la ligne *140* du fichier *bief_valida.f*. Puis, il s'agit d'une instabilité due à une soustraction (une cancellation).

```
Breakpoint 1, instability_ () in /local00/home/S51270/MesTravauxLocal/
WorkSpace/TelemacCadna/malpassetCadna/cas.txt31991_tmp/out31991_intel_
64_11.exe

#0  0x0000000000a6a514 in instability_ () in /local00/home/S51270/Mes
TravauxLocal/WorkSpace/TelemacCadna/malpassetCadna/cas.txt31991_tmp/out
31991_intel_64_11.exe

#1  0x0000000000a4f3b0 in cadna_sub_mp_sub_dst_dst_ () in /local00/home
/S51270/MesTravauxLocal/WorkSpace/TelemacCadna/malpassetCadna/cas.txt31
991_tmp/out31991_intel_64_11.exe
```

```
#2  0x000000000051a772 in bief_valida (varref= (...), textref=(...), ur
ef=<no value>, refformat=(...), varres=<no value>, textres=(...), ures=
8, resformat=(...), maxtab=100, np=13541, it=1, maxit=1, acomparer=(...),
 .tmp.TEXTREF.len_V$593=32, .tmp.REFFORMAT.len_V$599=8, .tmp.TEXTRES.len
_V$59d=32, .tmp.RESFORMAT.len_V$5a2=8) at /local00/home/S51270/MesTravau
xLocal/WorkSpace/TelemacCadna/v6p1WithCadna/bief/bief_v6p1/sources/bief_
valida.f:140
Continuing.
```

7.4 Vers une plate-forme de vérification numérique industrielle

La bibliothèque CADNA est un outil formidable pour la validation numérique. Son principal apport se retrouve dans le diagnostic qu'il propose à la fin de chaque exécution. Nous insistons particulièrement sur cet apport. A notre connaissance, il n'existe aucun outil libre qui offre des fonctionnalités similaires à ceux de CADNA. Cependant, comme nous l'avons montré dans le chapitre 6, le diagnostic n'est pas toujours facile à exploiter. Le manuel de référence de CADNA, propose de faire un débogage incrémental du code. Une telle opération n'est pas évidente lorsqu'on traite un très gros code particulièrement un code industriel et que ce dernier génère un nombre important d'instabilités. Nous pensons qu'il existe quelques limites pour un usage industriel complète du logiciel. Nous proposons ici quelques pistes et idées d'amélioration.

Un outil de post-traitement. L'idée ici serait de rendre automatique les opérations décrites en section 7.3. En fait, l'opération de vérification numérique est complètement décorrélée de la conception du code. Dans le monde industriel, utiliser CADNA reviendrait à faire travailler un numéricien et un développeur. En termes de coût financier, cela reviendrait à payer deux ingénieurs pour une seule tâche. L'outil de post-traitement permettrait d'éviter ce coût. On pourrait alors imaginer un outil qui prendra en paramètre le nom de l'exécutable et les paramètres et qui en sortie fournira des histogrammes sur les instabilités détectées, les fonctions qui les génèrent et les lignes de code responsables.

Plusieurs niveaux de validation. Dans certains cas, CADNA détecte des instabilités qui, techniquement en sont, mais à y voir de près peuvent être ignorées. Par exemple, après une soustraction entre deux valeurs proches, le résultat stochastique res peut être $(0;0;-\epsilon;0)$. Dans ce cas de figure, CADNA détectera une cancellation. L'ingénieur averti comprendra le pourquoi et l'ignorera. Ces genres d'opérations se font généralement pendant les procédures de validation des codes où les résultats de la simulation sont comparés à des valeurs de référence. Nous proposons plusieurs niveaux de validation(plus ou moins stricte). On pourrait y associer le paramètre $cadna\_cancel$ de la fonction d'initialisation. Par exemple, on peut considérer que dans le niveau le moins strict CADNA compare les différences entre les différents champs x, y et z. Si cette différence est supérieure à ϵ, on pourrait alors signaler une instabilité. Un niveau peut également être consacré à la gestion des zéros informatiques. Comme nous l'avons mentionné, l'apparition de zéros informatiques rend obsolète le reste de l'exécution et les résultats obtenus à la fin. En conséquence, les fonctions mathématiques génèrent des NaNs. Par exemple, si on souhaite calculer la racine carrée de $(0;0;-\epsilon;0)$ on obtiendra un NaN. L'idée serait de réfléchir à une gestion particulière des $@.0$ qui permettrait de terminer l'exécution du code avec des résultats un minimum exploitables.

Un outil de développement intégré complet (IDE). La dernière étape pour avoir un outil plus accessible serait de l'intégrer dans un IDE ou de développer un IDE attitré à CADNA. L'idée serait d'associer CADNA avec un débogueur. Il serait alors possible d'automatiser l'intégration de CADNA et le post-traitement des diagnostics.

Conclusion

Dans ce chapitre, nous avons présenté une procédure d'utilisation du logiciel CADNA. Cette procédure repose sur nos récentes expériences et particulièrement sur le travail effectué sur le code Telemac-2D(chapitre 6). Quelques idées pour l'amélioration de CADNA ont également été présentées. Nous pensons que ces améliorations feront de CADNA un outil de validation numérique complet et extrêmement efficace.

Chapitre 8

Conclusion générale

En conclusion, nous présentons ici un bilan de nos travaux avant de proposer quelques perspectives.

Le travail présenté dans ce mémoire s'articule au tour de la problématique de la validation numérique des codes de calcul industriels utilisés dans le cadre des simulations numériques. *Pourquoi une validation numérique des codes de simulation numérique industriels ?* Telle est l'interrogation que l'on pourrait avoir à la lecture du titre de ce manuscrit. Le chapitre 1 apporte quelques éléments de réponse. Ce chapitre souligne l'importance de la simulation numérique pour des acteurs industriels comme EDF. En effet, "simuler" permet d'orienter et/ou de valider des décisions stratégiques des industriels. De plus, l'avènement du calcul haute performance a permis à la simulation numérique de prendre une nouvelle envergure. De nos jours, on est en mesure de simuler des phénomènes de plus en plus complexes en des temps records. A titre d'exemple, les codes de neutronique peuvent actuellement traiter un problème neutronique de 10^{11} inconnues en 8 heures alors qu'en 2009 ce même type de calcul nécessitait 100000 heures pour traiter un cas de 10^{12} inconnues.

Dans le chapitre 1, nous attirons également l'attention du lecteur sur les différentes approximations faites durant le processus de simulation numérique. Il apparaît que plusieurs étapes séparent l'étude du phénomène physique de l'exécution du code sur ordinateur. Chacune de ces étapes peut être source d'erreurs. Dans cette thèse, nous nous intéressons particulièrement aux erreurs dues à la résolution numérique en arithmétique flottante IEEE 754, arithmétique que nous avons présentée dans le chapitre 2. En effet, le calcul sur les nombres flottants peut facilement engendrer des pertes de précision. Ces pertes sont encore plus importantes dès qu'il s'agit d'exécutions sur des architectures parallèles où l'on ne peut contrôler l'ordre des opérations. De fait, il est presque impossible d'avoir des résultats reproductibles dès lors qu'on

travaille sur des architectures parallèles. Vu l'importance des décisions prises grâce aux simulations numériques, il importe de s'assurer de la qualité des résultats produits. *Comment évaluer efficacement, dans un contexte industriel, la qualité des résultats d'un code parallèle de simulation numérique sans pour autant détruire les performances de ce dernier ?* Telle était notre interrogation au début de ce travail.

Nous définissons *validation numérique* d'un code par le processus visant à étudier sa fiabilité en termes de qualité numérique. Concrètement, cela consiste à considérer les effets de la propagation d'erreurs d'arrondi sur la qualité des résultats d'un calcul. Un état de l'art des outils et méthodes de validation numérique a été présenté dans le chapitre 2. Nous avons alors exposé les besoins spécifiques aux acteurs industriels en termes de vérification numérique. Des travaux ont montrés que le logiciel de validation CADNA était adapté aux études de qualité dans un contexte industriel. La bibliothèque CADNA, que nous avons présenté au chapitre 3, permet de localiser précisément les résultats entachés d'erreurs d'arrondi. CADNA est assez facile à implémenter sur un code séquentiel écrit dans un seul langage (Fortran90, C/C++). Toutefois, implémenter CADNA dans un code industriel peut s'avérer un brin complexe, ce type de code faisant généralement appel à des bibliothèques externes (MPI, BLACS, BLAS, LAPACK). Pour étudier complètement le comportement numérique d'un code avec CADNA, il est alors nécessaire de développer des extensions compatibles avec ces bibliothèques. Cela dit, l'implémentation de ces diverses extensions doit répondre à un critère de performance puisque la complexité algorithmique et la taille des logiciels de calcul numérique impliquent d'importants temps d'exécution. Il est donc important de chercher à minimiser l'impact de CADNA sur les performances de ces bibliothèques. Au cours ce de travail, nous nous sommes intéressé au standard de communication MPI et aux routines d'algèbre linéaire BLAS.

Le chapitre 4 a été consacré à l'implémentation d'une extension de CADNA pour le standard de communication MPI. Cette bibliothèque est utilisée dans les codes parallèles pour les échanges de données entre processus. Nous avons implémenté une extension CADNA_MPI qui permet l'échange et la réduction des données de types stochastiques. L'utilisation des *padding* a permis d'obtenir un rapport de 4 entre la taille des types stochastiques et ceux des types conventionnels (*float*, *double*). Ce rapport permet d'avoir les données stochastiques alignées en mémoire et donc de meilleures performances pour les temps de communication. CADNA_MPI permet également d'avoir un diagnostic complet des instabilités détectées sur chaque processus. CADNA_MPI est complètement fonctionnel et a été testé sur divers cas. Il a notamment été utilisé pour valider le schéma de communication du code Telemac-3D (voir chapitre 6). Il a également contribué à la mise en place de l'approche "xD+P" utilisée pour estimer la qualité numérique des résultats de Telemac-3D [Moulinec *et al.*, 2011a].

Nous nous sommes ensuite posé la question de l'implémentation de CADNA dans les bibliothèques de calcul scientifique. En ce sens, nous avons développé une routine DgemmCadna qui implémente un produit matriciel basé sur l'arithmétique stochastique discrète. Cette routine est présentée au chapitre 5. Nous montrons que l'implémentation directe (naïve) de CADNA dans la routine DGEMM des BLAS, introduit un important surcoût (supérieur à 1000 pour une matrice carré de taille 1024). Ce surcoût vient de l'utilisation de la méthode CESTAC, qui effectue chaque opération arithmétique trois fois avec à chaque opération un mode d'arrondi choisi aléatoirement. Il faut ajouter à cette première explication le fait que l'implémentation basique de la fonction DGEMM se fait avec trois boucles et sans aucune optimisation de l'utilisation de la mémoire. Nous présentons, à travers notre routine DgemmCADNA, une méthodologie pour réduire ce surcoût. Nous montrons qu'une meilleure utilisation des données avec un mode de stockage et un algorithme adapté à l'architecture matérielle, permet d'obtenir de biens meilleures performances. Outres ces apports d'ordre algorithmique, nous avons

également introduit une nouvelle implémentation de la méthode CESTAC. L'objectif de cette implémentation est de faire le minimum de changement d'arrondi tout en respectant les hypothèses stochastiques de la méthode CESTAC. Cette routine a permis de réduire ce surcoût d'un facteur 1100 à un facteur 35 par rapport aux versions de BLAS optimisées (GotoBlas [Goto et van de Geijn, 2008a]).

Enfin, nous avons travaillé sur la validation numérique des codes de calcul industriels. Notre interrogation était de savoir comment utiliser efficacement l'outil CADNA dans un contexte industriel, où l'aspect coût est un critère prépondérant. Nous nous sommes confronté au code Telemac-2D, code de mécanique des fluides dédié aux écoulements à surface libre (chapitre 6). Une dernière partie de notre travail à été consacrée à l'étude de la qualité numérique du code Telemac-2D. Ce code, développé par le département LNHE d'EDF R&D, permet de simuler les écoulements à surface libre en résolvant les équations de Saint-Venant discrétisées avec la méthode des éléments finis [Hervouet, 2007]. L'objectif de ce travail est de détecter les sources potentielles d'instabilités numériques dans le code Telemac-2D et de proposer des solutions pour minimiser l'impact de la propagation des erreurs arrondi. Le débogage numérique à l'aide de CADNA a montré que plus de 30% des instabilités détectées apparaissent dans les produits scalaires liés au calcul du résidu dans les méthodes itératives et dans les résolutions de systèmes linéaires creux. Nous montrons que l'utilisation des algorithmes de produit scalaire compensé [Ogita *et al.*, 2005] permet d'améliorer la précision des résultats sans toutefois dégrader les performances du code. Sur la base de cette expérience, nous avons mis en place une procédure de validation numérique des codes de calcul industriels dans le chapitre 7.

A travers les pages de ce mémoire, nous avons mis en exergue l'importance de la vérification numérique, en particulier pour les codes de simulation industriel. Nous faisons aussi remarquer qu'il est important de trouver l'outil adéquat qui réponde aux besoins de l'industriel pour une étude de qualité numérique efficace. Notre expérience de validation numérique chez EDF, nous a alors permis de proposer des idées pour la mise en place d'une plate-forme (efficace et performante) de vérification numérique industrielle basée sur le logiciel CADNA (chapitre 7). Intuitivement, le développement de cette plate-forme constitue la suite naturelle des travaux présentés dans ce mémoire. Nous dégageons ci-dessous trois principaux axes de travail pour y parvenir :

Sur les bibliothèques de communication. Il serait intéressant de travailler sur le second standard de communication : BLACS. Nous avons commencé à travailler sur cette problématique. Nous présentons une première implémentation en annexe de ce document. Outre les différents standard de communication, il serait pertinent d'étudier la possibilité de coupler CADNA et le paradigme de programmation parallèle OpenMP. L'objectif à terme serait d'avoir une version de CADNA compatible avec tous les outils de programmation parallèle.

Sur les bibliothèques de calcul scientifique. Notre travail a été consacré à la performance de la routine DgemmCADNA. La performance peut être améliorer avec l'utilisation des instructions vectoriels. Un premier travail dans ce sens a montré qu'il fallait modifier les opérations mathématiques directement dans CADNA. En effet, une utilisation directe des unités vectorielles est limitée par la non homogénéité des types stochastiques. Il faudrait également s'intéresser à la détection des instabilités numériques et particulièrement aux multiplications instables. Il faut travailler à trouver une méthode de détection qui limite le surcoût qui en

découlerait. L'objectif à terme serait d'étendre nos travaux et la méthodologie mise en œuvre aux autres routines BLAS et aux principales bibliothèques d'algèbre linéaire (LAPACK, ScaLAPACK).

Sur la validation des codes industriels. A partir de notre expérience, nous avons mis en place une procédure de validation numérique pour les codes industriels. Les travaux futurs seront consacrés à la mise en place des idées exposées au chapitre 7. On pourra ainsi développer un outil de développement intégré complet (IDE) basé sur l'outil CADNA. Il serait alors possible d'automatiser l'intégration de CADNA dans les codes industriels et et l'analyse de la qualité numérique de ces derniers.

Liste des publications

Revues internationales avec comité de lecture

[Montan *et al.*, 2013a] Séthy MONTAN, Jean-Marie CHESNEAUX, Christophe DENIS et Jean-Luc LAMOTTE. « Efficient matrix multiplication based on discrete stochastic arithmetic ». *Reliable computing journal*, Soumis le 10 fevrier 2013.

[Denis et Montan, 2012] Christophe DENIS et Séthy MONTAN. « Numerical verification of industrial numerical codes ». *ESAIM : Proc.*, 35 :107–113, 2012.

Communications dans des conférences internationales avec comité de lecture

[Montan *et al.*, 2013c] S. MONTAN, C. DENIS, D. R. EMERSON et C. MOULINEC. « Using compensated algorithms to improve the accuracy of a parallel hydrodynamic software suite ». *In B.H.V. TOPPING et P. IVÁNYI, éditeurs : Proceedings of the Third International Conference on Parallel, Distributed, Grid and Cloud Computing for Engineering*, Stirlingshire, United Kingdom, 2013.

[Montan *et al.*, 2012a] S. MONTAN, J-M. CHESNEAUX, C. DENIS et J-L. LAMOTTE. « Towards an efficient implementation of CADNA in the blas : Example of DgemmCADNA routine ». *Proceedings of the 15th GAMM - IMACS International Symposium on Scientific Computing, Computer Arithmetic, and Validated Numerics (SCAN)*, Novosibirsk, Russia, September 2012.

Communications dans des conférences francophones

[Montan *et al.*, 2012b] S. MONTAN, J-M. CHESNEAUX, C. DENIS et J-L. LAMOTTE. « Une implémentation efficace de cadna dans les blas : exemple de la routine dgemmcadna ». 5 ièmes Rencontres Arithmétique de l'Informatique Mathématique, RAIM 2012, 20-22 juin 2012, Dijon, France.

[Montan *et al.*, 2011] S. MONTAN, J-M. CHESNEAUX, C. DENIS et J-L. LAMOTTE. « Implémentation efficace de la bibliothèque cadna dans les bibliothèques de calcul et de communication scientifiques ». 5e Biennale Française des Mathématiques Appliquées, SMAI 2011, 23-27 mai 2011, Guidel, France.

Poster

[Montan *et al.*, 2013b] S. MONTAN, J-M. CHESNEAUX, C. DENIS et J-L. LAMOTTE. « Etude de la propagation des erreurs d'arrondi dans un code d'hydrodynamique parallèle ». Poster présenté à Compas'2013, Conférence d'informatique en Parallélisme, Architecture et Système. 15-18 janvier 2013, Grenoble, France.

Annexe A

Les plateformes de test : descriptif technique

A.1 Protocole de mesure des temps de calcul

Les temps d'exécution présentés dans tout ce chapitre sont calculés à base d'une moyenne de temps. On exécute chaque routine n fois ($n = 5$), on retire le maximum et le minimum et on calcule la moyenne du reste.

A.2 Le poste scientifique standard à EDF : *HP Z600 workstation*

La Z600 est une machine quadcore (quatre cœurs sur une même puce) muni d'un processeur Intel Xeon E5620[57] architecture x86 64 bits de type Westmere-EP[58] [Intel, 2012]. Nous recensons les principales caractéristiques de cette machine dans le *tableau* A.1. Notons que ce type d'architecture est muni de la technologie Hyper-Threading[59] ou Simultaneous multithreading (SMT) d'Intel. Cette technologie permet une meilleure gestion du multithreading et offre surtout la possibilité d'exécuter plusieurs threads parallèlement sur le même cœur. C'est la raison pour laquelle nous pouvons exécuter jusqu'à 8 threads sur notre machine. La puissance crête théorique est estimée à 153.6 Gflops en simple précision.

La Z600 est muni d'un système d'exploitation Calibre 7, OS maison d'EDF. C'est une version de Linux dont la souche est basée sur Debian 6.0 (squeeze).

57. http://ark.intel.com/products/47925
58. architecture de type Nehalem munie de l'Hyper-threading pouvant disposer de 6 cœurs (12 threads)
59. http://www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html

Tableau A.1 – Caractéristiques matérielles de la machine de test.

Nom	Processeur	Cœur	SIMD	GFlops	
			Date	théo.	obs.
$^{32}_{1\times4}$Nehalem	Xeon E5620 2.4 GHz	2×4	SSE 4.2 03/2010	153.6	150.06

Memory	Cache		
	L1	L2	L3
12 Go 3×DDR3-1066 Mhz	4×64 Kio	4×256 Kio	12 Mio

A.3 Le Cluster Ivanoe

Le cluster Ivanoe est la deuxième machine la plus performante d'EDF R&D. Il a été conçu par IBM pour atteindre un puissance crête théorique de $191.3TFlops$. Il est classé 205^e dans le classement de juin 2013 du Top500 avec une performance crête observée de $168.8TFlops$.

La machine dispose de 2 frontales accessibles via SSH (voir *tableau* A.2), de 1382 nœuds de calculs standards (voir *tableau* A.3), de 29 nœuds de calcul grande mémoire (voir *tableau* A.4) et 34 nœuds de calcul graphique (voir *tableau* A.5). Les frontales sont installées avec la souche Calibre7 en mode diskless (environnement chargé en mémoire RAM). Pour la gestion espaces de stockage, le cluster dispose également d'un ensemble de serveurs GPFS proposant 2 systèmes de fichiers distribués montés sur tous les nœuds de calcul et les frontales. Enfin, les nœuds sont interconnectés par un réseau InfiniBand 1 x 40Gb QDR.

Tableau A.2 – Caractéristiques matérielles des Frontales.

Nombre	Processeur			Mémoire
	CPU	Nombre CPU	Cœurs/CPU	
2	Intel Xeon X7560 Nehalem-EX	4	8	256Go

Tableau A.3 – Caractéristiques matérielles des nœuds de calculs standards.

Nombre	Processeur			Mémoire
	CPU	Nombre CPU	Coeurs/CPU	
1382	Intel Xeon X5670 Westmere	2	6	24Go

Tableau A.4 – Caractéristiques matérielles des nœuds de calculs grande mémoire.

Nombre	Processeur			Mémoire	Disque Dur
	CPU	Nombre CPU	Coeurs/CPU		
16	Intel Xeon X5660 Westmere	2	6	128Go	73Go
8	Intel Xeon X7560 Nehalem-EX	4	8	256Go	
4	Intel Xeon X7560 Nehalem-EX	4	8	512Go	SAS 15K RPM
1	Intel Xeon X7560 Nehalem-EX	4	8	1To	

Tableau A.5 – Caractéristiques matérielles des nœuds de calculs graphiques.

Nombre	Processeur			Mémoire	Disque Dur
	CPU	Nombre CPU	Coeurs/CPU		
24	Intel Xeon X5550 Nehalem-EP	2	4	32Go	146Go
6	Intel Xeon X5550 Nehalem-EP	2	4	64Go	
2	Intel Xeon X5550 Nehalem-EP	2	4	128Go	SAS 15K RPM
1	Intel Xeon X7560 Nehalem-EX	4	8	512Go	
1	Intel Xeon X7560 Nehalem-EX	4	8	1To	

A.4 Le Cluster IBM iDataplex de Daresbury

Le Cluster IBM iDataplex [60] appartient au STFC Daresbury Laboratory (Royaume-Uni). C'est une machine de 40 nœuds munis chacun d'un processeur Intel Westmere X56 processors de 2×6 cœurs, 2.67GHz et 24GB de mémoire. Les nœuds sont interconnectés par un réseau infiniband QDR. Deux nœuds sont munis de cartes nVidia Fermi GPU.

60. http://www.stfc.ac.uk/cse/37231.aspx

Annexe B

Implémentation efficace de CADNA dans les routines BLACS

B.1 BLACS : Basic Linear Algebra Communication Subprograms

BLACS (Basic Linear Algebra Communication Subprograms) [61] [Dongarra et Whaley, 1997] est une bibliothèque de communication écrite en C permettant, sur une grille 2D prédéfinie de processus, d'échanger des blocs de matrices (rectangulaires ou triangulaires) entre ces processus, de les diffuser globalement et de calculer sur eux des réductions. Les routines BLACS, offrent pratiquement les mêmes fonctionnalités qu'offre le standard MPI. Elles permettent de faire des communications point à point, les communications collectives et des opérations de réductions (min, max et la somme).

Il existe quatre différentes implémentations des BLACS, chacune de ses implémentations se basant sur un standard de communication (MPI , MPL [62], PVM [63]). Par contre, la syntaxe d'appel au BLACS reste identique d'un système à l'autre, d'où son intérêt. Bien qu'écrite en C, la bibliothèque dispose aussi d'une interface Fortran.

Contrairement à la majorité des standards de communication, les BLACS fonctionnent avec des tableaux 2D (en algèbre linéaire, les matrices sont en réalité en 2D). Il définit deux types de matrices qui sont stockés en mémoire en «Column-major».

- Matrice rectangulaire définie par M (nombre de lignes) et N (nombre de colonnes) et LDA (la distance, en mémoire, entre deux éléments successifs d'une ligne de la matrice).
- Matrice trapézoïdale définie par M,N,LDA et deux autres paramètres : UPLO pour indiquer si la matrice est montante ou descendante et DIAG pour indiquer si la matrice est diagonale.

61. http://www.netlib.org/blacs/
62. IBM Message Passing Layer
63. PVM (Parallel Virtual Machine) est une bibliothèque de communication (langages C et Fortran) pour machines parallèles et réseau d'ordinateurs (locaux ou distants, éventuellement hétérogènes). Il permet à un réseau d'ordinateurs d'apparaître comme un seul ordinateur.

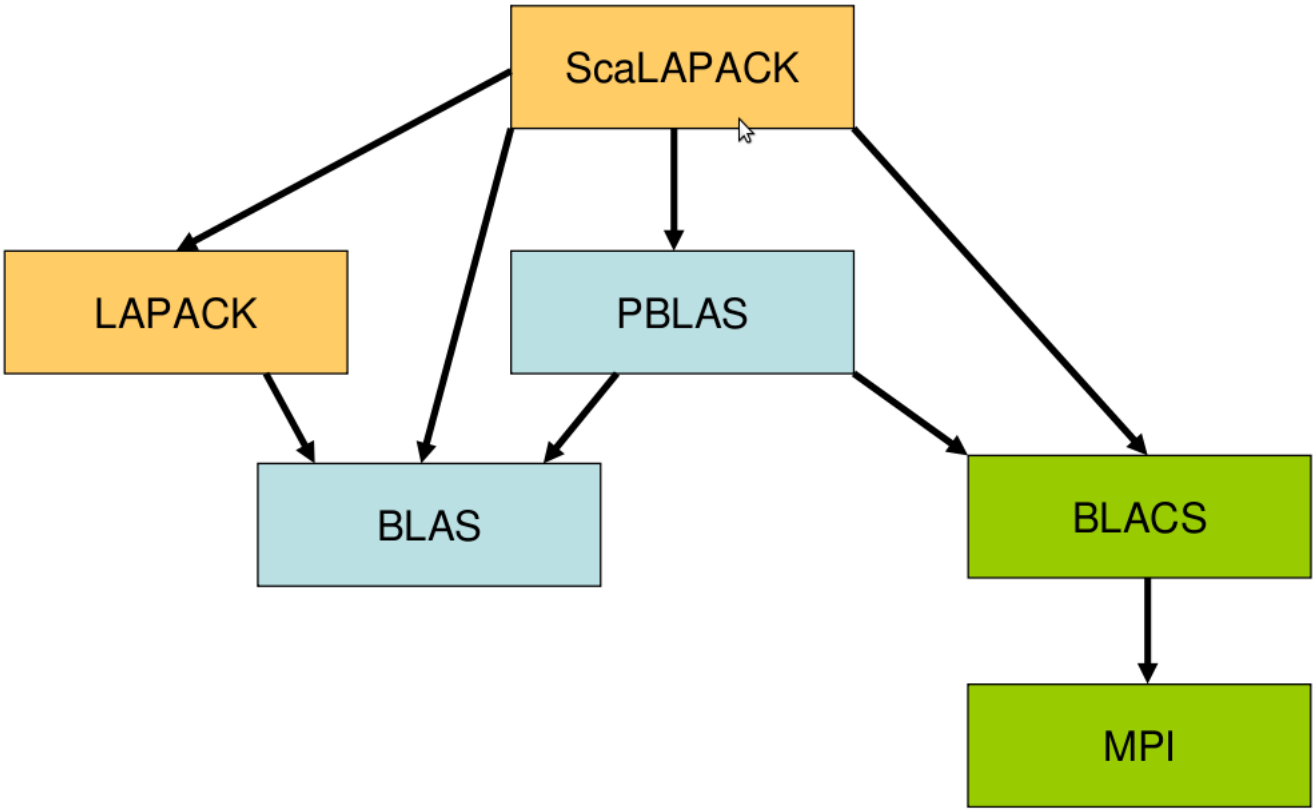


Figure B.1 – Organisation des bibliothèques de base d'algèbre linéaire

Les routines BLACS ont été écrit principalement pour faciliter l'implémentation des versions parallèles des bibliothèques d'algèbre linéaire (voir*figure* B.1). Ainsi, la bibliothèque PBLAS (Parallel BLAS) est basée sur des noyaux BLAS (Basic Linear Algebra Subprograms, voir chapitre 5) et des routines de communication BLACS. ScaLAPACK (Scalable Linear Algebra PACKage) réutilise PBLAS et implémente les fonctionnalités avancées de LAPACK (Linear Algebra PACKage). On peut trouver un large aperçu de ces bibliothèques à l'adresse `http://www.netlib.org/`.

La bibliothèque est composée essentiellement de deux sortes de sous-programmes des routines pour l'initialisation et la terminaison de l'environnement BLACS et des routines pour l'échange de données.

1. Initialisation et terminaison :
   - BLACS_PINFO( ) : initialise l'environnement BLACS
   - BLACS_GET( ) : définit un contexte unique de communication
   - BLACS_GRIDINIT( ) : initialise une grille logique 2D de processus
   - BLACS_GRIDEXIT( ) : libère la grille logique de processus
   - BLACS_EXIT( ) : libère toutes les grilles de processus

   Une fois la grille définie, les communications se font dans ce contexte en utilisant les primitives BLACS de communication.
2. Primitives de communication : Ces routines suivent une convention pour leurs noms :
   - communication point à point et collective (broadcast) : vXXYY2D : la lettre «v» indique le type de donnée, «XX» la forme de la matrice, et les lettres «YY» pour indiquer le type de communication (voir Tableaux B.1 ,B.2 ,B.3).
   - routines de réductions : vGZZZ2D , la lettre «v» pour indiquer le type de donnée et «ZZZ» l'opération (AMX pour max, AMN pour le min et SUM pour la somme).

Tableau B.1 – Types de données

v	signification
I	entier (integer)
S	réel simple précision
D	réel double précision
C	complexe simple précision
Z	complexe double précision

Tableau B.2 – Types de matrices

XX	signification
GE	matrice rectangulaire
TR	matrice triangulaire

Tableau B.3 – Types d'envoi/réception

YY	signification
SD	envoi point à point
RV	réception point à point
BS	envoi collectif
BR	réception collective

Tableau B.4 – Exemples de routines BLACS

IGESD2D( )	envoi point à point de matrice rectangulaire d'entier
DGERV2D( )	réception point à point de matrice rectangulaire de réel en double précision
ZTRBS2D( )	envoi broadcast de matrice triangulaire de complexe double précision
ZTRBR2D( )	réception broadcast de matrice triangulaire de complexe double précision

B.2 Comment associer les BLACS et CADNA ?

Pour associer CADNA aux BLACS, il faut implémenter des routines pour pouvoir échanger des matrices de types *double_st* ou single_st. Dans cette optique, nous nous sommes alors intéressé au fonctionnement interne des BLACS.

B.2.1 *Comment fonctionnent les routines BLACS ?*

Pour chaque type de donnée (integer, *double*, *float*, complex) et suivant le type de matrice à transférer il existe une routine spécifique d'envoi ou de réception. Chaque routine suit le principe suivant :

1. création du type dérivé de la matrice (à la manière MPI)
2. soit on met la matrice dans un buffer/soit on la met dans un pack
3. envoi/réception du buffer ou du pack

B.2.2 *Les types stochastiques et les BLACS ?*

Pour nos routines CADNA_BLACS, nous avons choisi de suivre le même principe et d'utiliser les types dérivés et les opérateurs de l'extension CADNA_MPI. Il aurait fallu alors partir des codes source des BLACS et travailler sur le point 1. Cependant, nous nous sommes confronté à des soucis de compatibilités notamment pour le Fortran. CADNA étant écrit en C++ et Fortran 90, de nombreux problèmes se sont posés au niveau des éditions de lien.

Finalement, nous avons utilisé des void* pour le type du buffer d'envoi et les paramètres MPI_Datatype et MPI_Op dans les prototypes de nos fonctions. Cette petite astuce, nous permet d'avoir des routines que nous pouvons qualifier de *génériques*. Il faut à chaque envoi ou réception, signifier le type dérivé sur lequel on travaille. Dans le cas d'une routine de réduction, il faut signifier aussi l'opération de réduction qu'on souhaite faire. Nos routines ont des prototypes comme ci-dessous :

Routine d'envoi : "sc" pour les types stochastiques

```
void Cscgesd2d(int ctxt,int m,int n,void* A,int lda,MPI_Datatype type,int ↩
    rdest,int cdest)
```

Routine de réception :

```
void Csctrsd2d(int ctxt,char *uplo,char* diag,int m,int n,void* A,int ↩
    lda,MPI_Datatype type,int rdest,int cdest)
```

Routine de réduction :

```
void Cscgop2d  (int ConTxt, char *scope, char *top, int m, int n, void ↩
    *A,void *B, int lda, MPI_Datatype type, MPI_Op op, int rdest, int ↩
    cdest);
```

CADNA_BLACS est encore en phase de développement. Les principales fonctions de base fonctionnent correctement mais il subsiste quelques petits soucis au niveau de l'interface Fortran, notamment pour l'implémentation des opérations MAX_LOC et MIN_LOC.

Annexe C

Equations d'hydrodynamique à surface libre

Dans cette partie, un aperçu des équations de Navier-Stokes à surface libre est présenté. Nous expliquons brièvement comment les équations de Saint-Venant sont obtenues à partir des équations de Navier-Stokes. Nous nous sommes largement inspiré du chapitre 2 de l'ouvrage de Jean-Michel Hervouet [Hervouet, 2003].

C.1 Équations de Navier-Stokes non-hydrostatiques

Les équations de Navier-Stokes constituent l'élément essentiel de la mécanique des fluides à surface libre. D'après [Hervouet, 2003], elles sont une forme adaptée de la relation fondamentale de la dynamique. Ces équations résultent des travaux de Newton et de Gabriel Stokes sur les équations fondamentales de l'hydrodynamique.

Les équations de Navier-Stokes sont établies à partir de deux équations :

i) l'équation de continuité pour la conservation de la masse de fluide ;

ii) l'équation de quantité de mouvement pour la relation fondamentale de la dynamique que l'on doit à Newton.

Soit ρ la masse volumique du fluide et $\overrightarrow{U}$ le vecteur vitesse ayant pour composantes (U, V, W). La conservation de la masse de fluide contenue dans un domaine Ω s'exprime de la façon suivante :

$$\frac{d}{dt}\left(\int_{\Omega} \rho \, d\Omega\right) = 0 \tag{C.1}$$

L'équation de continuité est obtenue à partir de l'équation C.1. D'abord le théorème de Leibnitz permet de décomposer la conservation de la masse en variations observées à l'intérieur du domaine et en flux aux frontières :

$$\frac{d}{dt}\left(\int_{\Omega} \rho \, d\Omega\right) = \int_{\Omega} \frac{\partial \rho}{\partial t} d\Omega + \int_{\Gamma} \rho \overrightarrow{U} \cdot \overrightarrow{n} \, d\Gamma \tag{C.2}$$

Grâce au théorème de Gauss, le flux aux frontières $\int_\Gamma \rho\overrightarrow{U} \cdot \overrightarrow{n} d\Gamma$ est ramené à $\int_\Omega div(\rho\overrightarrow{U})d\Omega$. Puis, comme la conservation de la masse doit être assurée quelque soit le domaine Ω, on obtient une expression locale de l'équation de continuité :

$$\frac{\partial \rho}{\partial t} + div(\rho\overrightarrow{U}) = 0 \tag{C.3}$$

On considère ensuite l'équation de la quantité de mouvement (ou deuxième loi de Newton) qui s'écrit $\overrightarrow{f} = \frac{md\overrightarrow{U}}{dt}$ pour un fluide (car sa masse volumique est considérée variable). Il a été montré que la quantité $\frac{md\overrightarrow{U}}{dt}$ est équivalente à $\frac{d}{dt}(\int_\Omega \rho\overrightarrow{U} d\Omega)$ qui peut être écrit comme suit :

$$\int_\Omega \frac{\partial(\rho\overrightarrow{U})}{\partial t} d\Omega + \int_\Gamma \nabla(\rho\overrightarrow{U} \otimes \overrightarrow{U}) d\Gamma \tag{C.4}$$

où $\otimes$ désigne le produit extérieur entre deux tenseurs d'ordre et ∇ l'opérateur tensoriel "nabla".

La force $\overrightarrow{f}$ est la somme des forces extérieures (qui s'appliquent dans la masse du fluide) et des forces dites de contact (qui s'appliquent sur sa surface). Ici, les forces extérieures sont notées $\overrightarrow{g} + \overrightarrow{F}$ et les forces de contact $d\overrightarrow{F} = \underline{\underline{\sigma}}\overrightarrow{n} d\Gamma$. Le vecteur $\overrightarrow{g}$ représente la pesanteur, $d\Gamma$ est un élément de surface du domaine Ω dont la normale extérieure est $\overrightarrow{n}$ et $\underline{\underline{\sigma}}$ est le tenseur des contraintes.

Les forces appliquées au domaine Ω peuvent alors être explicitées de la façon suivante :

$$\int_\Omega \rho(\overrightarrow{g} + \overrightarrow{F}) d\Omega + \int_\Omega div(\underline{\underline{\sigma}}) d\Omega \tag{C.5}$$

L'équation de la quantité de mouvement est déduite de l'équation C.5 :

$$\frac{\partial(\rho\overrightarrow{U})}{\partial t} + \nabla(\rho\overrightarrow{U} \otimes \overrightarrow{U}) = div(\underline{\underline{\sigma}}) + \rho\overrightarrow{g} + \rho\overrightarrow{F} \tag{C.6}$$

Les équations C.3 et C.6 permettent alors d'établir les équations de Navier-Stokes sous forme dite "conservative" :

Continuité :

$$\frac{\partial \rho}{\partial t} + div(\rho\overrightarrow{U}) = 0 \tag{C.7}$$

Quantité de mouvement :

$$\frac{\partial(\rho\overrightarrow{U})}{\partial t} + \nabla(\rho\overrightarrow{U} \otimes \overrightarrow{U}) = div(\underline{\underline{\sigma}}) + \rho\overrightarrow{g} + \rho\overrightarrow{F} \tag{C.8}$$

A partir de l'équation de continuité (C.7), on obtient une forme dite "non-conservative" pour l'équation de la quantité de mouvement :

$$\frac{\partial(\overrightarrow{U})}{\partial t} + \overrightarrow{U}\nabla\overrightarrow{U} = \frac{1}{\rho} div(\underline{\underline{\sigma}}) + \overrightarrow{g} + \overrightarrow{F} \tag{C.9}$$

Signalons que les deux équations (conservative et non conservative) ne sont pas équivalentes pour les fluides compressibles [64]. Pour les fluides newtoniens [65], dans le cas des fluides incompressibles, le tenseur des contraintes est exprimé comme ci-dessous :

$$\underline{\underline{\sigma}} = -p\underline{\underline{\delta}} + 2\mu\underline{\underline{D}} \tag{C.10}$$

64. Compressibilité : propriété que possèdent tous les corps de pouvoir être comprimés, de pouvoir diminuer de volume sous l'action d'une pression.

65. On appelle fluide newtonien (en hommage à Isaac Newton) un fluide dont la loi contrainte – vitesse de déformation est linéaire http://fr.wikipedia.org/wiki/Fluide_newtonien

où p est la pression, δ est le tenseur d'intensité, μ le coefficient de viscosité dynamique et $\underline{\underline{D}}$ le tenseur des taux de déformation définie comme suit :

$$\underline{\underline{D}} = \frac{1}{2}(\nabla \overrightarrow{U} + {}^t \nabla \overrightarrow{U}) \tag{C.11}$$

μ est une constante dépendant du fluide qui peut être également écrite sous la forme $\rho\nu$ où ν est le coefficient de viscosité cinématique.

Par la suite, les variations de la masse volumique sont considérées suffisamment faibles pour que l'équation de continuité sous forme incompressible puisse être utilisée. L'équation de continuité et le fait que μ soit constant en espace permet de simplifier $-\frac{1}{\rho} div(2\mu\underline{\underline{D}})$ en $\nu\Delta(\overrightarrow{U})$. A partir de ces formulations, on obtient les équations de Navier-Stokes en coordonnées cartésiennes :

Equation de continuité :

$$\frac{\partial U}{\partial x} + \frac{\partial U}{\partial y} + \frac{\partial U}{\partial z} = 0 \tag{C.12}$$

Quantité de mouvement :

$$\frac{\partial U}{\partial t} + U\frac{\partial U}{\partial x} + V\frac{\partial U}{\partial y} + W\frac{\partial U}{\partial z} = -\frac{1}{\rho}\frac{\partial p}{\partial x} + \nu\Delta(U) + F_x \tag{C.13}$$

$$\frac{\partial V}{\partial t} + U\frac{\partial V}{\partial x} + V\frac{\partial V}{\partial y} + W\frac{\partial V}{\partial z} = -\frac{1}{\rho}\frac{\partial p}{\partial y} + \nu\Delta(V) + F_y \tag{C.14}$$

$$\frac{\partial W}{\partial t} + U\frac{\partial W}{\partial x} + V\frac{\partial W}{\partial y} + W\frac{\partial W}{\partial z} = -\frac{1}{\rho}\frac{\partial p}{\partial z} + g + \nu\Delta(W) + F_z \tag{C.15}$$

Cette forme des équations de Navier-Stokes est dite non-hydrostatique puisqu'aucune hypothèse n'a été faite sur la pression. On notera également 4 inconnues dans les équations U, V, W et p. Le Logiciel Telemac-3D résout ces équations.

C.2 Conditions aux limites

Pour les écoulements à surface libre, le domaine peut avoir des formes très complexes. Il est limité par :

- le fond (en général imperméable) ;
- la surface de l'eau (elle n'est pas franchie par le fluide mais évolue au cours du temps) ;
- les frontières physiques : structures verticales infranchissables (mur, quai), digues, rives, plages ;
- les frontières imaginaires pour la modélisation numérique.

Ces conditions limites influenceront directement l'évolution du domaine occupé par le fluide. Dans le cadre des simulations numériques d'écoulements à surface libre, il convient alors de rajouter aux équations de Navier-Stokes établies plus haut, les équations aux conditions limites. Ces équations, que nous ne présentons ici, sont expliquées de façon très détaillée dans les ouvrages de Jean-Michel Hervouet [Hervouet, 2007, Hervouet, 2003]. Le lecteur intéressé y trouvera largement son compte.

C.3 Pression hydrostatique

La pression notée p se décompose en deux parties [Razafindrakoto, 2004] : sa partie hydrostatique ϕ et sa partie dynamique ici π telles que :

$$p = \phi + \pi \tag{C.16}$$

et

$$\frac{\partial \phi}{\partial z} = -\rho g \quad \text{et} \quad \phi(Z_s) = 0 \tag{C.17}$$

soit en intégrant sur la verticale :

$$\phi(x,z,t) = \rho g(Z_s - z) \tag{C.18}$$

Dans le cas du modèle non hydrostatique, la pression telle que nous venons de la définir est utilisée. Le modèle hydrostatique revient à considérer que l'accélération due à la pression équilibre la gravité. Cette hypothèse implique également de faibles mouvements dans le sens vertical (vitesse verticale négligeable). La pression $p(x,y,z,t)$ est donc due uniquement au poids de la colonne d'eau au-dessus du point de coordonnées(x,y,z). La seconde composante de la pression est alors négligée :

$$\frac{\partial \pi}{\partial z} = 0 \tag{C.19}$$

On a alors :

$$-\frac{1}{\rho}\frac{\partial p}{\partial z} - g = 0 \tag{C.20}$$

Pour simplifier, on écrit :

$$p(x,y,z,t) = \rho g(Z_s - z) \tag{C.21}$$

En particulier, au fond du domaine, on a $p = \rho g h$ où $h = Z_s - Z_f$ est la hauteur d'eau, Z_s la cote de la surface libre et Z_f la cote au fond.

C.4 Équations de Saint-Venant hydrostatiques

On doit ces équations aux Comte de Saint-Venant [66] et à ses travaux sur la mécanique théorique et la dynamique des fluides. Les équations de Saint-Venant, encore appelées "de Barré de Saint-Venant", permettent de modéliser les écoulements à surface libre en eaux peu profondes, d'où leur appellation anglaise "Shallow water equations".

Les équations sont obtenues à partir des équations de Navier-Stokes et de l'hypothèse de la pression hydrostatique. Les équations de Navier-Stokes avec masse volumique constante et pression hydrostatique sont moyennées sur la verticale par intégration depuis le fond jusqu'à la surface. Deux nouvelles variables sont définies :

$$u = \frac{1}{h}\int_{Z_f}^{Z_s} U dz \quad \text{et} \quad v = \frac{1}{h}\int_{Z_f}^{Z_s} V dz \tag{C.22}$$

L'équation de continuité C.12 devient alors :

$$\int_{Z_f}^{Z_s} (\frac{\partial U}{\partial x} + \frac{\partial U}{\partial y} + \frac{\partial U}{\partial z}) dz = 0 \tag{C.23}$$

La règle de Leibnitz et les conditions d'imperméabilité (pas de transfert de masse à travers le fond et la surface) permettent de reformuler cette équation comme ci-dessous :

$$\frac{\partial h}{\partial t} + div(h\overrightarrow{u}) = 0 \tag{C.24}$$

On effectue aussi la moyenne des équations de quantité de mouvement en considérant la pression hydrostatique et les conditions d'imperméabilité. On obtient les équations suivantes en U et en V :

Quantité de mouvement :

$$\frac{\partial (hu)}{\partial t} + \frac{\partial (huu)}{\partial x} + \frac{\partial (huv)}{\partial y} = -gh\frac{\partial Z_s}{\partial x} + hF_x + div(h\nu_e \overrightarrow{grad}(u)) \tag{C.25}$$

66. Admis à l'école Polytechnique à 15 ans, il entra ensuite à l'école des Ponts et Chaussées dont il sortit premier en 1825.

$$\frac{\partial(hv)}{\partial t} + \frac{\partial(huv)}{\partial x} + \frac{\partial(hvv)}{\partial y} = -gh\frac{\partial Z_s}{\partial y} + hF_y + div(h\nu_e\overrightarrow{grad}(v)) \tag{C.26}$$

où ν_e est une diffusion effective qui doit prendre en compte la viscosité turbulence et la dispersion. L'équation en W (C.15) n'est pas utilisée ici car elle a déjà été mise en contribution dans l'hypothèse d'hydrostaticité.

Les équations C.24, C.25, C.26 constituent les équations "de Barré de Saint-Venant" bidimentionnelles, sous forme dite "conservative".

La forme dite "non conservative" de ces équations est obtenue à partir de la forme conservative en développant les dérivées de produits de fonction. Aussi, en présence d'une prise ou d'un rejet d'eau sur le fond, l'équation de continuité est égale à un terme source notée Sce. On obtient finalement :

Équation de continuité :

$$\frac{\partial h}{\partial t} + \overrightarrow{u} \cdot \overrightarrow{grad}(h) + hdiv(\overrightarrow{u}) = S_{ce} \tag{C.27}$$

Équations de mouvement (selon x puis selon y) :

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} = -g\frac{\partial Z_s}{\partial x} + F_x + \frac{1}{h}div(hv_e\overrightarrow{grad}(u)) \tag{C.28}$$

$$\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} = -g\frac{\partial Z_s}{\partial y} + F_y + \frac{1}{h}div(hv_e\overrightarrow{grad}(v)) \tag{C.29}$$

Table des figures

Liste des tableaux

Bibliographie

[lar, 2012] (2012). Les Supercalculateurs relèvent le défi. *La Recherche*, 469. Cité page 10.

[Advanced Micro Devices, 2012] ADVANCED MICRO DEVICES, I. N. A. G. L. (2012). AMD Core Math Library (ACML) version 5.2.0, User Guide. Rapport technique, AMD. Cité page 76.

[Agullo *et al.*, 2009] AGULLO, E., DEMMEL, J., DONGARRA, J., HADRI, B., KURZAK, J., LANGOU, J., LTAIEF, H., LUSZCZEK, P. et TOMOV, S. (2009). Numerical linear algebra on emerging architectures : The PLASMA and MAGMA projects. *In Journal of Physics : Conference Series*, volume 180, page 012037. IOP Publishing. Cité pages 76, 77.

[Allaire, 2005] ALLAIRE, G. (2005). *Analyse numérique et optimisation : une introduction à la modélisation mathématique et à la simulation numérique*. Editions Ecole Polytechnique. Cité page 4.

[Anderson *et al.*, 1999] ANDERSON, E., BAI, Z. et BISCHOF, C. (1999). *LAPACK Users' guide*, volume 9. Society for Industrial Mathematics. Cité pages 76, 91.

[Anderson *et al.*, 1992] ANDERSON, E., BAI, Z., BISCHOF, C., DEMMEL, J., DONGARRA, J., DU CROZ, J., GREENBAUM, A., HAMMARLING, S., MCKENNEY, A., OSTROUCHOV, S. *et al.* (1992). LAPACK users' guide : Release 1.0. Rapport technique, Argonne National Lab., IL (United States). Cité page 91.

[Anderson *et al.*, 1993] ANDERSON, E., DONGARRA, J. et OSTROUCHOV, S. (1993). LAPACK Working Note 41 Installation Guide for LAPACK. Rapport technique, University of Tennessee, Knoxville, TN. Cité page 28.

[ANSI/IEEE, 1985] ANSI/IEEE (1985). *IEEE Standard for Binary Floating-Point Arithmetic*. ANSI/IEEE Standard 754–1985. Cité page 21.

[Antony *et al.*, 2006] ANTONY, J., JANES, P. et RENDELL, A. (2006). Exploring thread and memory placement on NUMA architectures : Solaris and Linux, UltraSPARC/FirePlane and Opteron/HyperTransport. *High Performance Computing-HiPC 2006*, pages 338–352. Cité page 93.

[Apple Inc., 2011] APPLE INC. (2011). Accelerate Framework Reference. Rapport technique. Cité page 76.

[Archambeau *et al.*, 2004] ARCHAMBEAU, F., MEHITOUA, N. et SAKIZ, M. (2004). Code_Saturne : a Finite Volume Code for the Computation of Turbulent Incompressible Flows - Industrial Applications. *International Journal on Finite Volumes*, 1. Cité page 5.

[Baboulin *et al.*, 2012] BABOULIN, M., GRATTON, S., LACROIX, R., LAUB, A. J. *et al.* (2012). Efficient computation of condition estimates for linear least squares problems. Également publié en tant que LAPACK Working Note 273, http://hal.archives-ouvertes.fr/docs/00/73/11/36/PDF/RR-8065.pdf. Cité page 32.

[Bader, 2007] BADER, D. A. (2007). *Petascale computing : algorithms and applications*, volume 1. Chapman and Hall/CRC. Cité page 14.

[Bailey *et al.*, 2010] BAILEY, D., HIDA, Y. et LI, X. (2010). *QD (C++/Fortran-90 double-double and quad-double package)*. http://crd-legacy.lbl.gov/~dhbailey/mpdist/. Cité page 30.

[Bailey *et al.*, 2002] BAILEY, D., YOZO, H., LI, X. et THOMPSON, B. (2002). ARPREC : An arbitrary precision computation package. Cité page 30.

[Barrault *et al.*, 2011] BARRAULT, M., LATHUILIÈRE, B., RAMET, P. et ROMAN, J. (2011). Efficient parallel resolution of the simplified transport equations in mixed-dual formulation. *Journal of Computational Physics*, 230(5):2004–2020. Cité page 12.

[Basic Linear Algebra Subprograms Technical (Blast) Forum, 2001] BASIC LINEAR ALGEBRA SUBPROGRAMS TECHNICAL (BLAST) FORUM (2001). Basic Linear Algebra Subprograms Technical (Blast) Forum Standard. Rapport technique, University of Tennessee, Knoxville, Tennessee. Cité page 75.

[Bischof *et al.*, 1997] BISCHOF, C., ROH, L. et MAUER-OATSY, A. (1997). ADIC : an extensible automatic differentiation tool for ANSI-C. *Urbana*, 51:61802. Cité page 31.

[Bischof *et al.*, 2008] BISCHOF, C. H., HOVLAND, P. D. et NORRIS, B. (2008). On the implementation of automatic differentiation tools. *Higher-Order and Symbolic Computation*, 21(3):311–331. Cité page 31.

[Blackford *et al.*, 1997] BLACKFORD, L., CLEARY, A., CHOI, J., D'AZEVEDO, E., DEMMEL, J., DHILLON, I., DONGARRA, J., HAMMARLING, S., HENRY, G., PETITET, A. *et al.* (1997). *ScaLAPACK users' guide*, volume 4. Society for Industrial Mathematics. Cité pages 76, 91.

[Bouissou *et al.*, 2009] BOUISSOU, O., CONQUET, E., COUSOT, P., COUSOT, R., FERET, J., GHORBAL, K., GOUBAULT, E., LESENS, D., MAUBORGNE, L., MINÉ, A. *et al.* (2009). Space software validation using abstract interpretation. *In Proc. of the Int. Space System Engineering Conf., Data Systems in Aerospace (DASIA 2009)*, volume 669, pages 1–7. Citeseer. Cité page 33.

[Bourgerie *et al.*, 2010] BOURGERIE, Q., FORTIN, P. et LAMOTTE, J. (2010). Efficient complex matrix multiplication on the Synergistic Processing Element of the Cell processor. *In Cluster Computing Workshops and Posters (CLUSTER WORKSHOPS), 2010 IEEE International Conference on*, pages 1–8. IEEE. Cité page 91.

[Braconnier et Langlois, 2002] BRACONNIER, T. et LANGLOIS, P. (2002). From rouding error estimation to automatic correction with automatic differentiation. *In Automatic differentiation of algorithms : from simulation to optimization*, pages 351–357. Springer-Verlag New York, Inc. Cité page 31.

[Brent, 1978] BRENT, R. (1978). A Fortran multiple-precision arithmetic package. *ACM Transactions on Mathematical Software (TOMS)*, 4(1):57–70. Cité page 30.

[CADNA Team, 2010] CADNA TEAM (2010). CADNA for C/C++ source codes : user guide. http://www-pequan.lip6.fr/cadna/. Cité page 122.

[CEA, 2007] CEA (2007). La simulation numérique : simuler pour comprendre et anticiper. Cité page 4.

[Chapoutot *et al.*, 2009] CHAPOUTOT, A., MARTEL, M. *et al.* (2009). Différentiation automatique et formes de Taylor en analyse statique de programmes numériques. *Technique et Science Informatiques*, 28(4):503–531. Cité page 31.

[Chesneaux, 1995] CHESNEAUX, J. (1995). *L'arithmétique stochastique et le logiciel CADNA*. Thèse de doctorat, Université Pierre et Marie Curie (UPMC). Habilitation à Diriger des Recherches. Cité pages 40, 46.

[Chesneaux *et al.*, 2009] CHESNEAUX, J., GRAILLAT, S. et JÉZÉQUEL, F. (2009). Numerical validation and assessment of numerical accuracy. Extended version of [Chesneaux *et al.*, 2008], Oxford e-Research Centre, overview article, http://cpc.cs.qub.ac.uk/oerc_numerical_accuracy.pdf. Cité page 37.

[Chesneaux, 1988] CHESNEAUX, J.-M. (1988). *Etude théorique et implémentation en ADA de la méthode CESTAC*. Thèse de doctorat, Université Paris VI, Paris, France. Cité pages 29, 37, 39, 40 et 103.

[Chesneaux *et al.*, 2008] CHESNEAUX, J.-M., GRAILLAT, S. et JÉZÉQUEL, F. (2008). Rounding errors. *Wiley Encyclopedia of Computer Science and Engineering*. Cité page 170.

[Chesneaux et Vignes, 1992] CHESNEAUX, J.-M. et VIGNES, J. (1992). Les fondements de l'arithmétique stochastique. *Comptes rendus de l'Académie des sciences. Série 1, Mathématique*, 315(13):1435–1440. Cité page 29.

[Clint Whaley *et al.*, 2001] CLINT WHALEY, R., PETITET, A. et DONGARRA, J. (2001). Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1-2):3–35. Cité pages 77, 78, 82 et 92.

[Courau *et al.*, 2013] COURAU, T., MOUSTAFA, S., PLAGNE, L. et PONÇOT, A. (2013). DOMINO : A Fast 3D Discrete Ordinates Solver for Reference PWR Simulations and SPN Validation. *In International Conference on Mathematics and Computational Methods Applied to Nuclear Science & Engineering (M&C 2013)*, USA. Cité pages 11, 12.

[Dautray *et al.*, 1988] DAUTRAY, R., ARTOLA, M. et LIONS, J. L. (1988). *Analyse mathématique et calcul numérique pour les sciences et les techniques*, volume 4. Masson. Cité page 4.

[de Dinechin, 2007] de DINECHIN, F. (2007). *Matériel et logiciel pour l'évaluation de fonctions numériques*. Thèse de doctorat, Université Claude Bernard-Lyon. Habilitation à diriger des recherches. Précision, performance et validation. Cité page 23.

[De Dinechin, 2013] DE DINECHIN, F. (2013). De calculer juste à calculer au plus juste. Cours donné à Ecole "précision et reproductibilité en calcul numérique",Fréjus,France. Cité page 25.

[Dekker, 1971] DEKKER, T. J. (1971). A floating-point technique for extending the available precision. *Numerische Mathematik*, 18(3):224–242. Cité pages 125, 126.

[Delmas *et al.*, 2009] DELMAS, D., GOUBAULT, E., PUTOT, S., SOUYRIS, J., TEKKAL, K. et VÉDRINE, F. (2009). Towards an industrial use of FLUCTUAT on safety-critical avionics software. *In Formal Methods for Industrial Critical Systems*, pages 53–69. Springer. Cité page 33.

[Demmel et Hida, 2004] DEMMEL, J. et HIDA, Y. (2004). Accurate and efficient floating point summation. *SIAM Journal on Scientific Computing*, 25(4):1214–1248. Cité pages 26, 30 et 123.

[Demmel et Nguyen, 2013] DEMMEL, J. et NGUYEN, H. D. (2013). Fast Reproducible Floating-Point Summation. *In 21st IEEE Symposium on Computer Arithmetic, Austin, Texas, USA*. Cité pages 26, 123.

[Denis, 2012] DENIS, C. (2012). Évaluation de la qualité numérique de code de simulation industriels : premiers résultats et perceptives. Note Interne H-I23-2010-02071-FR, EDF R&D. Cité pages 113, 139.

[Denis et Montan, 2012] DENIS, C. et MONTAN, S. (2012). Numerical Verification of Industrial Numerical Codes. *ESAIM : Proc.*, 35:107–113. Cité pages 15, 113, 135, 139 et 149.

[Denis *et al.*, 2011] DENIS, C., MOULINEC, C., BARBER, R., EMERSON, D., RAZAFINDRAKOTO, E. et HERVOUET), J.-M. (2011). Simulate Accurately and Efficiently Large Scale Hydrodynamics Events. *In 33rd National Hydrology and Water Resources Symposium (IAHR2011), 26 June-1 July 2011*, Brisbane-Australia. Cité pages 118, 120 et 123.

[Didier *et al.*, 2004] DIDIER, L.-S., CHESNEAUX, J.-M. et RICO, F. (2004). The Fixed CADNA Library. *Numbers and Computers*. Cité page 46.

[Dodson *et al.*, 1991] DODSON, D. S., GRIMES, R. G. et LEWIS, J. G. (1991). Sparse extensions to the Fortran basic linear algebra subprograms. *ACM Transactions on Mathematical Software (TOMS)*, 17(2): 253–263. Cité page 75.

[Dongarra, 2011] DONGARRA, J. (2011). Linear Algebra Libraries for High- Performance Computing : Scientific Computing with Multicore and Accelerators. Part 1, SC2011 Tutorial. Lecture given at International Conference for High Performance Computing, Networking, Storage and Analysis in Seattle, Washington (SC2011). Cité page 76.

[Dongarra, 2013] DONGARRA, J. (2013). Algorithmic and Software Challenges when Moving Towards Exascale. Lecture given at 4th International Supercomputing Conference in Mexico (ISUM 2013) Manzanillo, Mexico. Cité page 8.

[Dongarra *et al.*, 2011] DONGARRA, J., BECKMAN, P., MOORE, T., AERTS, P., ALOISIO, G., ANDRE, J., BARKAI, D., BERTHOU, J., BOKU, T., BRAUNSCHWEIG, B. *et al.* (2011). The international exascale software project roadmap. *International Journal of High Performance Computing Applications*, 25(1):3–60. Cité page 8.

[Dongarra et van der Steen, 2012] DONGARRA, J. et van der STEEN, A. (2012). High-performance computing systems : Status and outlook. *Acta Numerica*, 21(1):379–474. Cité page 8.

[Dongarra et Whaley, 1997] DONGARRA, J. et WHALEY, R. (1997). LAPACK Working Note 94 A User's Guide to the BLACS v1. Rapport technique. Cité page 155.

[Dongarra, 1979] DONGARRA, J. J. (1979). *LINPACK users' guide*. Numéro 8. Siam. Cité pages 75, 76.

[Dongarra *et al.*, 1990a] DONGARRA, J. J., CRUZ, J. D., HAMMERLING, S. et DUFF, I. S. (1990a). Algorithm 679 : A set of level 3 basic linear algebra subprograms : model implementation and test programs. *ACM Trans. Math. Softw.*, 16(1):18–28. Cité page 75.

[Dongarra *et al.*, 1990b] DONGARRA, J. J., DU CROZ, J., HAMMARLING, S. et DUFF, I. S. (1990b). A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 16(1):1–17. Cité page 75.

[Dongarra *et al.*, 1988a] DONGARRA, J. J., DU CROZ, J., HAMMARLING, S. et HANSON, R. J. (1988a). Algorithm 656 : an extended set of basic linear algebra subprograms : model implementation and test programs. *ACM Trans. Math. Softw.*, 14(1):18–32. Cité page 75.

[Dongarra *et al.*, 1988b] DONGARRA, J. J., DU CROZ, J., HAMMARLING, S. et HANSON, R. J. (1988b). An extended set of FORTRAN basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 14(1):1–17. Cité page 75.

[Dongarra *et al.*, 2003] DONGARRA, J. J., LUSZCZEK, P. et PETITET, A. (2003). The LINPACK benchmark : past, present and future. *Concurrency and Computation : practice and experience*, 15(9):803–820. Cité page 76.

[Drepper, 2007] DREPPER, U. (2007). What every programmer should know about memory. http://people.redhat.com/drepper/cpumemory.pdf. Cité pages 91, 93 et 94.

[Duff *et al.*, 1997] DUFF, I. S., MARRONE, M., RADICATI, G. et VITTOLI, C. (1997). Level 3 basic linear algebra subprograms for sparse matrices : a user-level interface. *ACM Transactions on Mathematical Software (TOMS)*, 23(3):379–401. Cité page 75.

[Durand, 2011] DURAND, N. (2011). Synthèses et conclusions des études sur la modélisation numérique TELEMAC-3D de l'évolution des courants, de la salinité et de la température dans l'étang de Berre. Note Interne H-P74-2011-02289-FR, EDF R&D. Cité page 113.

[EDF R&D, 2012] EDF R&D (2012). EDF R&D : Relever les défis énergétiques. Cité page 5.

[EDF R&D, 2013] EDF R&D (2013). Les Logiciels Open Source. http://innovation.edf.com/recherche-et-communaute-scientifique/logiciels/tous-les-logiciels-41436.html. Cité page 7.

[Ern et Guermond, 2004] ERN, A. et GUERMOND, J.-L. (2004). *Theory and practice of finite elements*, volume 159. Springer. Cité page 4.

[Faverge, 2009] FAVERGE, M. (2009). *Ordonnancement hybride statique-dynamique en algèbre linéaire creuse pour de grands clusters de machines NUMA et multi-cœurs*. Thèse de doctorat, Université Sciences et Technologies-Bordeaux I. Cité pages 91, 93.

[Fournier *et al.*, 2011] FOURNIER, Y., BONELLE, J., MOULINEC, C., SHANG, Z., SUNDERLAND, A. et URIBE, J. (2011). Optimizing Code_Saturne computations on Petascale systems. *Computers & Fluids*, 45(1):103–108. Cité page 12.

[Fournier *et al.*, 2012] FOURNIER, Y., BONELLE, J., VEZOLLE, P., HEYMAN, J., D'AMORA, B., MAGERLEIN, K., MAGERLEIN, J., BRAUDAWAY, G., MOULINEC, C. et SUNDERLAND, A. (2012). Multiple threads and parallel challenges for large simulations to accelerate a general Navier–Stokes CFD code on massively parallel systems. *Concurrency and Computation : Practice and Experience*. Cité page 12.

[Fournier *et al.*, 2013] FOURNIER, Y., MOULINEC, C. et VEZOLLE, P. (2013). Testing, Deploying, and Evolving the Code_Saturne CFD Toolchain for Billion-Cell Calculations. *In* TOPPING, B. H. V. et IVÁNYI, P., éditeurs : *Proceedings of the Third International Conference on Parallel, Distributed, Grid and Cloud Computing for Engineering*, Stirlingshire, United Kingdom. Civil-Comp Press. paper 12. Cité page 12.

[Fousse *et al.*, 2007] FOUSSE, L., HANROT, G., LEFÈVRE, V., PÉLISSIER, P. et ZIMMERMANN, P. (2007). MPFR : A multiple-precision binary floating-point library with correct rounding. *ACM Transactions on Mathematical Software (TOMS)*, 33(2):13. Cité pages 30, 124.

[Gabriel *et al.*, 2004] GABRIEL, E., FAGG, G. E., BOSILCA, G., ANGSKUN, T., DONGARRA, J. J., SQUYRES, J. M., SAHAY, V., KAMBADUR, P., BARRETT, B., LUMSDAINE, A., CASTAIN, R. H., DANIEL, D. J., GRAHAM, R. L. et WOODALL, T. S. (2004). Open MPI : Goals, Concept, and Design of a Next Generation MPI Implementation. *In Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary. Cité page 52.

[Garg et Hendren, 2013] GARG, R. et HENDREN, L. (2013). A portable and high-performance matrix operations library for CPUs, GPUs and beyond. Sable Technical Report No. sable-2013-1, McGill University, School of Computer Science. Cité page 92.

[Gepner *et al.*, 2012] GEPNER, P., GAMAYUNOV, V. et FRASER, D. L. (2012). Effective Implementation of DGEMM on Modern Multicore CPU. *Procedia Computer Science*, 9(0):126–135. Proceedings of the International Conference on Computational Science, ICCS 2012. Cité page 91.

[Ghidaglia et Rittaud, 2004] GHIDAGLIA, J.-M. et RITTAUD, B. (2004). La simulation numérique. *La Recherche*, 380:736. Cité page 4.

[Girardeau-Montaut, 2006] GIRARDEAU-MONTAUT, D. (2006). *Détection de changement sur des données géométriques tridimensionnelles*. Thèse de doctorat, Télécom ParisTech. Cité page 7.

[Goldberg, 1991] GOLDBERG, D. (1991). What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys (CSUR)*, 23(1):5–48. Cité page 20.

[Goto et van de Geijn, 2008a] GOTO, K. et VAN DE GEIJN, R. (2008a). Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software (TOMS)*, 34(3):12. Cité pages 77, 92, 98 et 147.

[Goto et van de Geijn, 2008b] GOTO, K. et VAN DE GEIJN, R. (2008b). High-performance implementation of the level-3 BLAS. *ACM Transactions on Mathematical Software (TOMS)*, 35(1):4. Cité pages 77, 91, 92 et 93.

[Gottschling *et al.*, 2009] GOTTSCHLING, P., WISE, D. et JOSHI, A. (2009). Generic support of algorithmic and structural recursion for scientific computing 1. *International Journal of Parallel, Emergent and Distributed Systems*, 24(6):479–503. Cité pages 92, 99.

[Goubault *et al.*, 2008] GOUBAULT, E., PUTOT, S., BAUFRETON, P. et GASSINO, J. (2008). Static analysis of the accuracy in control systems : Principles and experiments. *In Formal methods for industrial critical systems*, pages 3–20. Springer. Cité page 31.

[Graillat *et al.*, 2011] GRAILLAT, S., JÉZÉQUEL, F., WANG, S. et ZHU, Y. (2011). Stochastic arithmetic in multiprecision. *Mathematics in Computer Science*, 5(4):359–375. Cité page 46.

[Granlund, 1996] GRANLUND, T. (1996). *Gnu MP, The GNU Multiple Precision Arithmetic Library*. Cité page 30.

[Griewank et Walther, 2008] GRIEWANK, A. et WALTHER, A. (2008). *Evaluating derivatives : principles and techniques of algorithmic differentiation*. Society for Industrial and Applied Mathematics. Cité page 31.

[Guillo *et al.*, 2010a] GUILLO, M., COUYRAS, D., PORA, Y. et HOAREAU, F. (2010a). Note de vérification et de validation de la version 1.1 du code COCAGNE. Note Interne H-I27-2010-03511-FR, EDF R&D. Cité page 10.

[Guillo *et al.*, 2010b] GUILLO, M., COUYRAS, D., PORA, Y. et HOAREAU, F. (2010b). Notice utilisateur de la version 1.1 du code COCAGNE. Note Interne H-I27-2010-03509-FR, EDF R&D. Cité page 5.

[Hascoëet et Araya-Polo, 2006] HASCOËET, L. et ARAYA-POLO, M. (2006). The adjoint data-flow analyses : Formalization, properties, and applications. *In Automatic Differentiation : Applications, Theory, and Implementations*, pages 135–146. Springer. Cité page 31.

[Heroux et Dongarra, 2013] HEROUX, M. A. et DONGARRA, J. (2013). Toward a New Metric for Ranking High Performance Computing Systems. Rapport technique SAND2013-4744, Sandia National Labs. Cité page 76.

[Hervouet, 2000] HERVOUET, J. (2000). A high resolution 2-D dam-break model using parallelization. *Hydrological processes*, 14(13):2211–2230. Cité page 117.

[Hervouet, 2007] HERVOUET, J. (2007). *Hydrodynamics of Free Surface Flows : Modelling with the finite element method*. Wiley. Cité pages 5, 113, 147 et 161.

[Hervouet, 2003] HERVOUET, J.-M. (2003). *Hydrodynamique des écoulements à surface libre : modélisation numérique avec la méthode des elements finis*. Presses de l'école nationale des ponts et chaussées. Cité pages 114, 115, 118, 159 et 161.

[Higham, 2002] HIGHAM, N. J. (2002). *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second édition. Cité pages 19, 23, 26, 27, 28, 32 et 123.

[Hofschuster et Krämer, 2004] HOFSCHUSTER, W. et KRÄMER, W. (2004). C-XSC 2.0–A C++ library for extended scientific computing. *Numerical software with result verification*, pages 259–276. Cité page 29.

[Hypolite *et al.*, 2012] HYPOLITE, M., TEXERAUD, J., MARGUET, S., TETART, F. et CARTIER, Y. (2012). COCCINELLE v3.10 : Manuel d'utilisation. Note Interne H-I27-2010-02481-FR, EDF R&D. Cité page 5.

[IBM, 2013] IBM (2013). ESSL for AIX V5.2 ESSL for Linux on POWER V5.2 Guide and Reference. Rapport technique, IBM. Cité page 76.

[IEEE Computer Society, 2008] IEEE COMPUTER SOCIETY (2008). *IEEE Standard for Floating-Point Arithmetic*. IEEE Standard 754-2008. available at http://ieeexplore.ieee.org/servlet/opac?punumber=4610933. Cité page 21.

[Intel, 2012] INTEL (2012). Intel® 64 and IA-32 Architectures Optimization Reference Manual. Rapport technique, Intel. http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html. Cité pages 95, 151.

[Intel, 2013] INTEL (2013). Intel® Math Kernel Library Reference Manual,Intel® MKL 11.0 Update 3. Rapport technique, Intel. http://software.intel.com/sites/products/documentation/hpc/mkl/mklman/index.htm. Cité page 76.

[Ioualalen, 2013] IOUALALEN, A. (2013). *Transformation de programmes synchrones pour l'optimisation de la précision numérique*. Thèse de doctorat, Ph. D. dissertation, Université de Perpignan, Perpignan, France. Cité page 31.

[Jézéquel, 2005] JÉZÉQUEL, F. (2005). *Dynamical control of approximation methods*. Thèse de doctorat, Université Pierre et Marie Curie (UPMC). Habilitation à Diriger des Recherches. Cité page 46.

[Jézéquel *et al.*, 2010] JÉZÉQUEL, F., CHESNEAUX, J. et LAMOTTE, J. (2010). A new version of the CADNA library for estimating round-off error propagation in Fortran programs. *Computer Physics Communications*, 181(11):1927–1928. Cité pages 14, 29, 37 et 46.

[Jézéquel et Lamotte, 2010] JÉZÉQUEL, F. et LAMOTTE, J. (2010). Numerical validation of Slater integrals computation on GPU. *In Proceedings of the 14th GAMM - IMACS International Symposium on Scientific Computing, Computer Arithmetic, and Validated Numerics (SCAN), Lyon (France)*. Cité page 46.

[Jézéquel *et al.*, 2013a] JÉZÉQUEL, F., LAMOTTE, J. et SAID, I. (2013a). Validation numérique de codes scientifiques sur GPU. Exposé Congrès SMAI, Biennale Française des Mathématiques Appliquées et Industrielles, mini-symposium "Etapes vers la reproductibilité numérique des calculs", Seignosse (France). Cité page 46.

[Jézéquel *et al.*, 2013b] JÉZÉQUEL, F., LAMOTTE, J.-L. et CHUBACH, O. (2013b). Parallelization of Discrete Stochastic Arithmetic on Multicore Architectures. *In Proceedings of the 2013 10th International Conference on Information Technology : New Generations*, pages 160–166. IEEE Computer Society. Cité page 46.

[Kahan, 1987] KAHAN, W. (1987). Doubled-precision IEEE standard 754 floating-point arithmetic. Cité page 127.

[Kahan *et al.*, 1998] KAHAN, W., DARCY, J. D., ENG, E. et COMPUTING, H.-P. N. (1998). How Java's floating-point hurts everyone everywhere. *In Talk given at the ACM 1998 Workshop on Java for High-Performance Network Computing (http ://www. cs. uscb. edu/conferences/wkahan/JAVAhurt. pdf)*. Cité page 26.

[Kalos et Whitlock, 2008] KALOS, M. H. et WHITLOCK, P. A. (2008). *Monte carlo methods*. Wiley-VCH. Cité page 4.

[Karypis et Kumar, 1998] KARYPIS, G. et KUMAR, V. (1998). A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392. Cité page 117.

[Kirschenmann, 2012] KIRSCHENMANN, W. (2012). *Vers des noyaux de calcul intensif pérennes*. Thèse de doctorat, Université de Lorraine, Metz, France. Cité pages 10, 77 et 108.

[Kirschenmann *et al.*, 2011] KIRSCHENMANN, W., PLAGNE, L., PONÇOT, A. et VIALLE, S. (2011). Parallel S PN on Multi-Core CPUS and Many-Core GPUS. *Transport Theory and Statistical Physics*, 39(2-4):255–281. Cité page 12.

[Knuth, 1969] KNUTH, D. (1969). *The Art of Computer Programming, Vol. 2. Seminumerical Algorithms*. Addison-Wesley, Reading, Mass. Cité page 125.

[Kulisch et Ebrary, 2008] KULISCH, U. et EBRARY, I. (2008). *Computer arithmetic and validity : theory, implementation, and applications*. Walter de Gruyter. Cité page 19.

[Kurzak *et al.*, 2009] KURZAK, J., ALVARO, W. et DONGARRA, J. (2009). Optimizing matrix multiplication for a short-vector SIMD architecture-CELL processor. *Parallel Computing*, 35(3):138–150. Cité pages 91, 92 et 100.

[Lamotte *et al.*, 2012] LAMOTTE, J., CHESNEAUX, J., DIDIER, L., JÉZÉQUEL, F. et VIGNES, J. (2012). Les développements récents liés à l'arithmétique stochastique. Exposé fait au CNES-CCT/SIL : Architecture des Systèmes Iinformatiques & Genie Logiciel,Séminaire précision numérique dans le Calcul Haute Performance, Toulouse, 19 janvier 2012. Cité page 46.

[Lamotte, 2004] LAMOTTE, J.-L. (2004). *Vers une chaine de validation des logiciels numériques à l'aide de méthodes probabilistes*. Thèse de doctorat, Université Pierre et Marie Curie (UPMC). Habilitation à Diriger des Recherches. Cité page 46.

[Lamotte *et al.*, 2010] LAMOTTE, J.-L., CHESNEAUX, J.-M. et JÉZÉQUEL, F. (2010). CADNA_C : A version of CADNA for use with C or C++ programs. *Computer Physics Communications*, 181(11):1925–1926. Cité pages 14, 29, 37 et 46.

[Langlois, 2001a] LANGLOIS, P. (2001a). Analyse d'erreur en précision finie. *Outils d'analyse numérique pour l'Automatique, Traité IC2*, pages 19–52. Cité page 28.

[Langlois, 2001b] LANGLOIS, P. (2001b). Automatic linear correction of rounding errors. *BIT Numerical Mathematics*, 41(3):515–539. Cité page 30.

[Langlois *et al.*, 2012a] LANGLOIS, P., GOOSSENS, B., PARELLO, D. et PORADA, K. (2012a). Less Hazard and More Scientific Research for the Computing Time of Summation Algorithms. Exposé fait à Numerical Sofware : Design, Analysis and Verification. IFIP WG 2.5 meeting, Santander, Spain, July 4-6 2012. Cité page 123.

[Langlois *et al.*, 2012b] LANGLOIS, P., PARELLO, D., GOOSSENS, B. et PORADA, K. (2012b). Less Hazardous and More Scientific Research for Summation Algorithm Computing Times. Rapport technique, Equipe-Projet DALI/LIRMM, Université de Perpignan. Manuscrit soumis à Science of Computer Programming. Cité pages 128, 129 et 133.

[Lawson *et al.*, 1979] LAWSON, C., HANSON, R., KINCAID, D. et KROGH, F. (1979). Basic linear algebra subprograms for Fortran usage. *ACM Transactions on Mathematical Software (TOMS)*, 5(3):308–323. Cité page 75.

[Li *et al.*, 2002] LI, X., DEMMEL, J., BAILEY, D., HENRY, G., HIDA, Y., ISKANDAR, J., KAHAN, W., KANG, S., KAPUR, A., MARTIN, M. *et al.* (2002). Design, implementation and testing of extended and mixed precision BLAS. *ACM Transactions on Mathematical Software (TOMS)*, 28(2):152–205. Cité pages 30, 75, 92 et 123.

[Lipshitz, 2013] LIPSHITZ, B. (2013). Communication-Avoiding Parallel Recursive Algorithms for Matrix Multiplication. Technical Report No. UCB/EECS-2013-100, Electrical Engineering and Computer Sciences,University of California at Berkeley. Cité page 92.

[Louvet, 2007] LOUVET, N. (2007). *Algorithmes compensés en arithmétique flottante : Précision, validation, performances*. Thèse de doctorat, Ph. D. dissertation, Université de Perpignan, Perpignan, France. Cité pages 26, 27.

[Lucquin et Pironneau, 1996] LUCQUIN, B. et PIRONNEAU, O. (1996). *Introduction au calcul scientifique*. Masson. Cité page 4.

[Malcolm, 1971] MALCOLM, M. A. (1971). On accurate floating-point summation. *Communications of the ACM*, 14(11):731–736. Cité pages 26, 30 et 123.

[Marguet, 2011] MARGUET, S. (2011). *La physique des réacteurs nucléaires*. Lavoisier. Cité page 5.

[Martel, 2006] MARTEL, M. (2006). Semantics of roundoff error propagation in finite precision calculations. *Higher-order and symbolic computation*, 19(1):7–30. Cité page 31.

[Martel, 2007] MARTEL, M. (2007). Semantics-based transformation of arithmetic expressions. *In Static Analysis*, pages 298–314. Springer. Cité page 31.

[Martel, 2009] MARTEL, M. (2009). Program transformation for numerical precision. *In Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation*, pages 101–110. ACM. Cité page 31.

[Message Passing Interface Forum, 1994] MESSAGE PASSING INTERFACE FORUM (1994). MPI : A Message-Passing Interface standard. *The International Journal of Supercomputer Applications and High Performance Computing*, 8. Cité page 51.

[Message Passing Interface Forum, 1995] MESSAGE PASSING INTERFACE FORUM (1995). MPI : A Message-Passing Interface standard (version 1.1). Rapport technique. http://www.mpi-forum.org. Cité page 51.

[Message Passing Interface Forum, 2012] MESSAGE PASSING INTERFACE FORUM (2012). MPI : A Message-Passing Interface Standard. Rapport technique. http://www.mpi-forum.org/docs/. Cité pages 51, 63.

[Meuer *et al.*, 2012] MEUER, H. W., STROHMAIER, E., DONGARRA, J. et SIMON, H. D. (2012). TOP500 Supercomputer Sites. Cité page 8.

[Montan, 2010] MONTAN, S. (2010). Développement de CADNA// : une bibliothèque d'étude de la propagation des erreurs d'arrondi dans un code parallèle sur machines à mémoire distribuée. Mémoire de fin d'études, Polytech'Paris-UPMC. Cité page 52.

[Montan *et al.*, 2011] MONTAN, S., CHESNEAUX, J.-M., DENIS, C. et LAMOTTE, J.-L. (2011). Implémentation efficace de la bibliothèque CADNA dans les bibliothèques de calcul et de communica-tion scientifiques. 5e Biennale Française des Mathématiques Appliquées, SMAI 2011, 23-27 mai 2011,Guidel, France. Cité pages 51, 149.

[Montan *et al.*, 2012a] MONTAN, S., CHESNEAUX, J.-M., DENIS, C. et LAMOTTE, J.-L. (2012a). Towards an efficient implementation of CADNA in the BLAS : Example of DgemmCADNA routine. *In Proceedings of the 15th GAMM - IMACS International Symposium on Scientific Computing, Computer Arithmetic, and Validated Numerics (SCAN), Novosibirsk, Russia*. Cité pages 73, 149.

[Montan *et al.*, 2012b] MONTAN, S., CHESNEAUX, J.-M., DENIS, C. et LAMOTTE, J.-L. (2012b). Une implémentation efficace de CADNA dans les BLAS : exemple de la routine DgemmCADNA. 5 ièmes Rencontres Arithmétique de l'Informatique Mathématique, RAIM 2012, 20-22 juin 2012, Dijon, France. Cité page 149.

[Montan *et al.*, 2013a] MONTAN, S., CHESNEAUX, J.-M., DENIS, C. et LAMOTTE, J.-L. (2013a). Efficient matrix multiplication based on discrete stochastic arithmetic. *Reliable computing journal*. Soumis 10 fevrier 2013. Cité pages 73, 149.

[Montan *et al.*, 2013b] MONTAN, S., CHESNEAUX, J.-M., DENIS, C. et LAMOTTE, J.-L. (2013b). Etude de la propagation des erreurs d'arrondi dans un code d'hydrodynamique parallèle. Poster présenté à Compas'2013, ComPAS : la Conférence d'informatique en Parallélisme, Architecture et Système.15-18 janvier 2013, Grenoble, France. Cité page 149.

[Montan *et al.*, 2013c] MONTAN, S., DENIS, C., EMERSON, D. R. et MOULINEC, C. (2013c). Using Compensated Algorithms to Improve the Accuracy of a Parallel Hydrodynamic Software Suite. *In* TOPPING, B. H. V. et IVÁNYI, P., éditeurs : *Proceedings of the Third International Conference on Parallel, Distributed, Grid and Cloud Computing for Engineering*, Stirlingshire, United Kingdom. Civil-Comp Press. paper 23. Cité pages 111, 149.

[Moore *et al.*, 2009] MOORE, R., KEARFOTT, R. et CLOUD, M. (2009). *Introduction to interval analysis*. Society for Industrial Mathematics. Cité page 28.

[Moreau *et al.*, 2009] MOREAU, O., BUVAT, F., GILLES-PASCAUD, C. et REBOUD, C. (2009). An approach for validating NDT eddy current simulation codes. *In 7 th International Conference on NDE in relation to structural integrity for nuclear and pressurized components, Yokohama*. Cité page 7.

[Moulinec *et al.*, 2011a] MOULINEC, C., DENIS, C., DURAND, N., BARBER, R., EMERSON, D., GU, X., RAZAFINDRAKOTO, E., ISSA, R. et HERVOUET, J.-M. (2011a). Coupling HPC and Numerical Validation : Accurate and Efficient Simulation of Large-scale Hydrodynamic Events. *In Proceedings of the Second International Conference on Parallel, Distributed, Grid and Cloud Computing for Engineering*, Stirlingshire, UK. Cité pages 14, 29, 67, 118, 135 et 146.

[Moulinec *et al.*, 2011b] MOULINEC, C., DENIS, C., PHAM, C.-T., ROUGÉ, D., HERVOUET, J.-M., RAZAFINDRAKOTO, E., BARBER, R., EMERSON, D. et GU, X.-J. (2011b). TELEMAC : An efficient hydrodynamics suite for massively parallel architectures. *Computers & Fluids*, 51(1):30–34. Cité page 115.

[Muller *et al.*, 2010] MULLER, J.-M., BRISEBARRE, N., DE DINECHIN, F., JEANNEROD, C.-P., LEFÈVRE, V., MELQUIOND, G., REVOL, N., STEHLÉ, D. et TORRES, S. (2010). *Handbook of Floating-Point Arithmetic*. Birkhauser Boston. Cité pages 19, 24, 25 et 26.

[Nicolas et Fouquet, 2013] NICOLAS, G. et FOUQUET, T. (2013). Adaptive mesh refinement for conformal hexahedralmeshes. *Finite Elements in Analysis and Design*, 67(0):1–12. Cité page 8.

[NVIDIA, 2013] NVIDIA (2013). cuBLAS User Guide. Rapport technique, NVIDIA. http://docs.nvidia.com/cuda/cublas/index.html. Cité page 77.

[Ogita *et al.*, 2005] OGITA, T., RUMP, S. et OISHI, S. (2005). Accurate sum and dot product. *SIAM Journal on Scientific Computing*, 26(6):1955–1988. Cité pages 26, 28, 30, 123, 124, 125, 126, 127, 128, 129 et 147.

[Ozaki *et al.*, 2012a] OZAKI, K., OGITA, T., OISHI, S. et RUMP, S. M. (2012a). Error-free transformations of matrix multiplication by using fast routines of matrix multiplication and its applications. *Numerical Algorithms*, 59(1):95–118. Cité page 92.

[Ozaki *et al.*, 2012b] OZAKI, K., OGITA, T., RUMP, S. M. et OISHI, S. (2012b). Fast algorithms for floating-point interval matrix multiplication. *Journal of Computational and Applied Mathematics*, 236(7):1795–1814. Cité page 92.

[Park *et al.*, 2003] PARK, N., HONG, B. et PRASANNA, V. K. (2003). Tiling, Block Data Layout, and Memory Hierarchy Performance. *IEEE Transactions on Parallel and Distributed Systems*, 14:640–654. Cité pages 91, 98 et 99.

[Razafindrakoto, 2004] RAZAFINDRAKOTO, E. (2004). Modélisation 3D d'un écoulement avec surface libre. Comparaison des modèles TRIM et Telemac-3D. Note Interne HP-75/04/043/A, EDF R&D. Cité page 161.

[Revol, 2004] REVOL, N. (2004). Introduction à l'arithmétique par intervalles. Cours donné à l'École Jeunes Chercheurs en Algorithmique et Calcul Formel, Grenoble,2004. Cité page 28.

[Revol et Rouillier, 2005] REVOL, N. et ROUILLIER, F. (2005). Motivations for an arbitrary precision interval arithmetic and the MPFI library. *Reliable Computing*, 11(4):275–290. Cité pages 30, 124.

[Revol et Théveny, 2013] REVOL, N. et THÉVENY, P. (2013). Parallel Implementation of Interval Matrix Multiplication. submitted. Cité page 92.

[Rump, 1988] RUMP, S. (1988). Algorithms for verified inclusions–theory and practice. *Reliability in computing*, 109. Cité page 20.

[Rump, 1999] RUMP, S. (1999). INTLAB—INTerval LABoratory. *Developments in reliable computing*, page 77. Cité page 29.

[Rump, 2005] RUMP, S. (2005). Computer-assisted proofs and self-validating methods. *Handbook on Accuracy and Reliability in Scientific Computation*, pages 195–240. Cité page 28.

[Rump, 2009] RUMP, S. M. (2009). Ultimately fast accurate summation. *SIAM Journal on Scientific Computing*, 31(5):3466–3502. Cité pages 26, 30, 123, 124 et 125.

[Rump *et al.*, 2008] RUMP, S. M., OGITA, T. et OISHI, S. (2008). Accurate floating-point summation part I : Faithful rounding. *SIAM Journal on Scientific Computing*, 31(1):189–224. Cité pages 123, 124.

[Rupp et Peniguel, 1999] RUPP, I. et PENIGUEL, C. (1999). Coupling heat conduction, radiation and convection in complex geometries. *International Journal of Numerical Methods for Heat & Fluid Flow*, 9(3):240–264. Cité page 7.

[Scott *et al.*, 2007] SCOTT, N., JÉZÉQUEL, F., DENIS, C. et CHESNEAUX, J. (2007). Numerical 'health check' for scientific codes : the CADNA approach. *Computer physics communications*, 176(8):507–521. Cité pages 14, 29.

[Souffez et Domain, 1997] SOUFFEZ, Y. et DOMAIN, C. (1997). Dynamique moléculaire : Bilan du CSN au CSCS. Note Interne HI-76/97/008/0, EDF R&D. Cité page 10.

[Stewart, 1998] STEWART, G. W. (1998). *Matrix Algorithms*. Society for Industrial and Applied Mathematics, Philadephia, PA. Cité page 92.

[Trebuchet, 2010] TREBUCHET, P. (2010). The linalg Library (Lapack made Generic). http://www-apr.lip6.fr/~trebuche/linalg.html. Cité page 77.

[Van Zee *et al.*, 2013] Van ZEE, F. G., SMITH, T., IGUAL, F. D., SMELYANSKIY, M., ZHANG, X., KISTLER, M., AUSTEL, V., GUNNELS, J., LOW, T. M. *et al.* (2013). Implementing Level-3 BLAS with BLIS : Early Experience. FLAME Working Note #69. Technical Report TR-13-03, The University of Texas at Austin, Department of Computer Science. submitted to SC13. Cité page 92.

[Van Zee et van de Geijn, 2012] Van ZEE, F. G. et van de GEIJN, R. A. (2012). BLIS : A Framework for Generating BLAS-like Libraries. FLAME Working Note #66. Technical Report TR-12-30, The University of Texas at Austin, Department of Computer Science. submitted to ACM Transactions on Mathematical Software. Cité page 92.

[Vignes, 1986] VIGNES, J. (1986). Zéro mathématique et zéro informatique. *Comptes rendus de l'Académie des sciences. Série 1, Mathématique*, 303(20):997–1000. Cité page 41.

[Vignes, 1993] VIGNES, J. (1993). A stochastic arithmetic for reliable scientific computation. *Mathematics and computers in simulation*, 35(3):233–261. Cité page 40.

[Vignes, 2004] VIGNES, J. (2004). Discrete stochastic arithmetic for validating results of numerical software. *Numerical Algorithms*, 37(1):377–390. Cité page 29.

[Vignes et La Porte, 1974] VIGNES, J. et LA PORTE, M. (1974). Error analysis in computing. *Information Processing*, 74:610–614. Cité pages 29, 37 et 39.

[Walther et Griewank, 2012] WALTHER, A. et GRIEWANK, A. (2012). Getting started with ADOL-C. *Combinatorial Scientific Computing*, 12:181. Cité page 31.

[Wilkinson, 1963] WILKINSON, J. (1963). Rounding errors in algebraic processes. *Notes on applied science*, (32). Réédité par Dover [Wilkinson, 1994]. Cité page 27.

[Wilkinson, 1994] WILKINSON, J. (1994). *Rounding errors in algebraic processes*. Dover Publications. Cité pages 28, 178.

[Xianyi *et al.*, 2012a] XIANYI, Z. *et al.* (2012a). OpenBLAS. http://xianyi.github.io/OpenBLAS/. Cité page 92.

[Xianyi *et al.*, 2012b] XIANYI, Z., QIAN, W. et YUNQUAN, Z. (2012b). Model-driven level 3 BLAS performance optimization on Loongson 3A processor. *In Parallel and Distributed Systems (ICPADS), 2012 IEEE 18th International Conference on*, pages 684–691. IEEE. Cité page 92.

[Yotov *et al.*, 2005] YOTOV, K., LI, X., REN, G., GARZARAN, M., PADUA, D., PINGALI, K. et STODGHILL, P. (2005). Is search really necessary to generate high-performance BLAS ? *Proceedings of the IEEE*, 93(2):358–386. Cité page 108.

[Zhu et Hayes, 2009] ZHU, Y.-K. et HAYES, W. B. (2009). Correct rounding and a hybrid approach to exact floating-point summation. *SIAM Journal on Scientific Computing*, 31(4):2981–3001. Cité pages 123, 124.

[Zhu et Hayes, 2010] ZHU, Y.-K. et HAYES, W. B. (2010). Algorithm 908 : Online exact summation of floating-point streams. *ACM Transactions on Mathematical Software (TOMS)*, 37(3):37. Cité pages 26, 30, 123 et 124.

UNIVERSITÉ PIERRE ET MARIE CURIE - PARIS 6

Sur la validation numérique des codes de calcul industriels

Séthy A. MONTAN

Résumé

L'étude de la qualité numérique est cruciale pour les codes industriels tels que ceux développés à EDF R&D. C'est d'autant plus important dans le contexte actuel où les simulations numériques sont faites sur des architectures pouvant exécuter des milliards d'opérations flottantes par seconde. Des études ont montré que la bibliothèque CADNA est un outil adapté à la validation numérique des codes industriels. Toutefois, CADNA ne peut être utilisée simplement sur des grands codes industriels, ces derniers faisant appel à des bibliothèques externes (MPI, BLACS, BLAS, LAPACK). Il est donc nécessaire de développer des extensions compatibles avec l'outil CADNA. L'implémentation de ces diverses extensions pose un problème de performance, la complexité algorithmique et la taille des logiciels de calcul numérique impliquant d'importants temps d'exécution. A titre d'exemple, l'implémentation directe de CADNA dans la routine de produit matriciel DGEMM des BLAS, introduit un surcoût supérieur à 1000 pour une matrice carrée de taille 1024. Les raisons de ce surcoût sont expliquées dans ce mémoire. Nous présentons également, à travers notre routine DgemmCADNA, la méthodologie pour réduire ce surcoût. Cette routine a permis de réduire ce surcoût d'un facteur 1100 à un facteur 35 par rapport à la version GotoBLAS.

Une deuxième partie de notre travail a été consacrée à l'étude de la qualité numérique du code Telemac-2D. Pour valider entièrement le code, nous avons implémenté une extension de CADNA pour le standard MPI. Le débogage numérique à l'aide de CADNA a montré que plus de 30% des instabilités détectées apparaissent dans les produits scalaires. L'utilisation des algorithmes de produit scalaire compensé permet d'améliorer la précision des résultats sans dégrader les performances du code.

Mots-clés : Simulation numérique, Codes industriels, Validation numérique, CADNA, Algorithmes compensés, Bibliothèques scientifiques, BLAS, MPI.

Abstract

Numerical verification of industrial codes, such as those developed at EDF R&D, is required to estimate the precision and the quality of computed results, even more for code running in HPC environments where millions of instructions are performed each second. These programs usually use external libraries (MPI, BLACS, BLAS, LAPACK). In this context, it is required to have a tool as nonintrusive as possible to avoid rewriting the original code. In this regard, the CADNA library, which implements the Discrete Stochastic Arithmetic, appears to be one of a promising approach for industrial applications. In the first part of this work, we are interested in an efficient implementation of the BLAS routine DGEMM (General Matrix Multiply) implementing Discrete Stochastic Arithmetic. The implementation of a basic algorithm for matrix product using stochastic types leads to an overhead greater than 1000 for a matrix of 1024×1024 compared to the standard version and commercial versions of xGEMM. Here, we detail different solutions to reduce this overhead and the results we have obtained. A new routine *Dgemm-CADNA* have been designed. This routine has allowed to reduce the overhead from 1100 to 35 compare to optimized BLAS implementations (GotoBLAS).

Then, we focus on the numerical verification of Telemac-2D computed results. Performing a numerical validation with the CADNA library shows that more than 30% of the numerical instabilities occurring during an execution come from the dot product function. A more accurate implementation of the dot product with compensated algorithms is presented in this work. We show that implementing these kinds of algorithms, in order to improve the accuracy of computed results does not alter the code performance.

Keywords : Numerical simulation, Industrial codes, Numerical validation, CADNA, Compensated algorithms, Scientific libraries, BLAS, MPI.