Calcul à haute performance et simulations stochastiques : Etude de la reproductibiité numérique sur architectures multicore et manycore

van Toan Dao

Numéro de DU: 2797

Université Clermont Auvergne

École Doctorale des Sciences pour l'Ingénieur

N° d'ordre : 792

Thèse

pour l'obtention du grade de

DOCTEUR D'UNIVERSITÉ

Discipline : Informatique

Soutenue par

Van Toan DAO

le 2/3/2017

Calcul à haute performance et simulations stochastiques : Etude de la reproductibilité numérique sur architectures multicore et manycore

---

Composition du jury :

Rapporteur :	Jean Pierre Müller, Directeur de Recherche, CIRAD, Montpellier.
	Jean Daniel Zücker, Directeur de Recherche 1, IRD, Bondy.
Examinateur:	Alexandre Muzy, Chargé de Recherche 1, CNRS-I3S, Nice.
	Paul-Antoine BISGAMBIGLIA, Maître de conférences, Université de Corse.
Directeurs :	David R.C. HILL, Professeur, LIMOS CNRS-UCA. (Directeur de thèse)
	Hong Quang NGUYEN, Professeur, IFI, Hanoi, Vietnam.
	Vincent BRETON, Directeur de recherche 1, IN2P3 CNRS-UCA.
Invité :	Claude MAZEL, Maître de conférences, LIMOS CNRS-UCA.

Numéro de DU: 2797

Résumé

La reproductibilité des expériences numériques sur les systèmes de calcul à haute performance est parfois négligée. De plus, les méthodes numériques employées pour une parallélisation rigoureuse des simulations stochastiques sont souvent méconnues. En effet, les résultats obtenus pour une simulation stochastique utilisant des systèmes de calcul à hautes performances peuvent être différents d'une exécution à l'autre, et ce pour les mêmes paramètres et les même contextes d'exécution du fait de l'impact des nouvelles architectures, des accélérateurs, des compilateurs, des systèmes d'exploitation ou du changement de l'ordre d'exécution en parallèle des opérations en arithmétique flottantes au sein des micro-processeurs. En cas de non répétabilité des expériences numériques, comment mettre au point les applications ? Quel crédit peut-on apporter au logiciel parallèle ainsi développé ?

Dans cette thèse, nous faisons une synthèse des causes de non-reproductibilité pour une simulation stochastique parallèle utilisant des systèmes de calcul à haute performance. Contrairement aux travaux habituels du parallélisme, nous ne nous consacrons pas à l'amélioration des performances, mais à l'obtention de résultats numériquement répétables d'une expérience à l'autre. Nous présentons la reproductibilité et ses apports dans la science numérique expérimentale. Nous proposons dans cette thèse quelques contributions, notamment : pour vérifier la reproductibilité et la portabilité des générateurs modernes de nombres pseudo-aléatoires ; pour détecter la corrélation entre flux parallèles issus de générateurs de nombres pseudo-aléatoires ; pour répéter et reproduire les résultats numériques de simulations stochastiques parallèles indépendantes.

Mots-clés : reproducibilité numérique, simulation stochastique parallèle, calcul à haute performance, architectures manycore et multicore.

Abstract

The reproducibility of numerical experiments on high performance computing systems is sometimes overlooked. Moreover, the numerical methods used for rigorous parallelization of stochastic simulations are often unknown. Indeed, the results obtained for a stochastic simulation using high performance computing systems can be different from run to run with the same parameters and the same execution contexts due to the impact of new architectures, accelerators, compilers, operating systems or a changing of the order of execution of the floating arithmetic operations within the micro-processors for parallelizing optimizations. In the case of non-repeatability of numerical experiments, how can we seriously develop a scientific application? What credit can be given to the parallel software thus developed?

In this thesis, we synthesize the main causes of non-reproducibility for a parallel stochastic simulation using high performance computing systems. Unlike the usual parallelism works, we do not focus on improving performance, but on obtaining numerically repeatable results from one experiment to another. We present the reproducibility and its contributions to the science of experimental and numerical computing. Furthermore, we propose some contributions, in particular: to verify the reproducibility and portability of top modern pseudo-random number generators, to detect the correlation between parallel streams issued from such generators, to repeat and reproduce the numerical results of independent parallel stochastic simulations.

Keywords: numerical reproducibility, parallel stochastic simulation, high performance computing, manycore and multicore architectures.

Remerciements

Tout d'abord, je voudrais remercier tous les membres du jury et en particulier les membres extérieurs : Jean-Daniel Zücker, Jean-Pierre Müller, Alexandre Muzy, Paul-Antoine Bisgambiglia pour avoir accepté d'évaluer mon travail, de lire cette thèse et d'assister à ma soutenance de thèse.

Merci à mes directeurs de thèse David (Benny) Hill, Nguyen Hong Quang et Vincent Breton pour leur écoute, leur encadrement, leur soutien, leur disponibilité et leurs conseils tout au long de mes travaux de thèse. Benny m'aidé à développer une autonomie pour la recherche avec sa méthode d'encadrement, ma reconnaissance pour lui est sans limite.

Je tiens à remercier Claude Mazel pour son suivi de mes travaux, ses conseils précieux lors de la rédaction de mon manuscrit, la relecture de ce manuscrit et son travail en co-encadrement sur l'évaluation de la méthode de test statistique des générateurs de nombres pseudo-aléatoires en parallèle.

Dans le même esprit, je tiens à remercier Paul-Antoine Bisgambiglia pour la relecture de ce manuscrit.

Cette thèse ne serait pas là sans Pascale Gouinaud, Elodie Philipponneau, Nicolas Champeil et Guillaume Avez, ingénieurs en informatique au LIMOS et ISIMA pour leur aides techniques et en particulier Guillaume pour son aide pour l'accès aux serveurs Xeon Phi.

Je me tourne à présent vers les personnes qui mon aidé pour les procédures administratives et financières Martine Caccioppoli, Séverine Miginiac et Béatrice Bourdieu au LIMOS et à l'ISIMA, Géraldine Fettahi pour l'IDGC, Catinca Birna et Nguyen Thuy Nga pour l'AUF.

Je suis reconnaissant au conseil de l'IFI pour m'avoir choisi pour une bourse de thèse et à l'Agence Universitaire de la Francophonie pour le financement de cette thèse. Je remercie le LIMOS (Laboratoire d'Informatique de Modélisation et d'Optimisation des Systèmes) – UMR-CNRS 6158 – Université Clermont Auvergne et l'équipe MSI (Modélisation et Simulation Informatique de systèmes complexes) – IFI (Institut Francophonie International) – Université national de Hanoï pour m'avoir accueilli pendant la durée de mes travaux de Doctorat.

Je tiens à remercie mes collègues et mes amis de travail à l'ISIMA et au LIMOS, avec une pensée particulière pour Jean Cornier, Antoine Dufaure et Pierre Schweitzer qui ont discuté avec moi sur bien des problèmes techniques. De même, cette thèse ne serait pas là sans mes amis présents au quotidien, en particulier mes amis vietnamiens.

En dernier lieu, je veux dire un grand merci à ma famille pour leur stimulation, leur encouragement et leur intérêt tout au long de mes travaux de thèse, tout ce que ma famille a fait pour moi n'a pas de prix. Mon père croit tellement en moi et m'aide à avoir un esprit de découverte, à aller de l'avant. Ma mère me dit des mots d'encouragement, et je la remercie pour tous ses soins et son amour. Mon épouse est toujours à mes côtés, pour partager mes problèmes et prendre soin de moi, ses parents s'ocuppent aussi de nous et nous encouragent. Mon petit frère, m'aidé à résoudre bien des problèmes et il a pris soin de ma famille quand je n'étais pas chez nous.

Table des matières

Table des figures

Table des tableaux

Table des codes, algorithmes et script

Table des équations

Introduction

1 - Contexte

Le calcul à haute performance prend une place de plus en plus significative dans les organismes de recherche publics et privés, notamment dans les domaines innovants et les nouvelles technologies liées aux Nanotechnologies, à la Biologie, à l'Informatique et à la Cognition (NBIC). Les systèmes de calcul à haute performance utilisent principalement des architectures de processeurs multi-cœurs et plus récemment, des nouvelles architectures dites accélérateurs de calcul de type manycores à base par exemple, de processeurs Intel Xeon Phi ou à base de processeurs graphiques généralisés de type GP-GPU (*General Purpose – Graphical Processing Unit*). Sur ces systèmes, les études les plus nombreuses concernent la recherche de performances, d'économies d'énergie et de techniques innovantes pour le développement d'applications parallèles. Même si ces centres d'intérêt sont justifiés, la précision mais surtout la reproductibilité des expériences numériques, devraient rester des préoccupations majeures dans le processus d'accroissement et de diffusion de la connaissance scientifique.

Le contexte technique retenu pour cette thèse est celui de la simulation parallèle stochastique. Il s'agit d'une méthode hautement parallélisable et importante dans de nombreux domaines de recherche appliquée. Lorsque des méthodes analytiques ne sont pas disponibles, les simulations stochastiques qui décrivent les phénomènes observés et qui se basent sur de l'échantillonnage, permettent d'améliorer nos connaissances sur des systèmes existants. Elles nous permettent de plus d'optimiser, de tester et de valider des modèles proposés pour des systèmes complexes. Parmi les simulations stochastiques, l'approche la plus connue est la simulation dite de Monte Carlo. Pour ce type de simulation ainsi que pour tout type de simulation stochastique, le générateur de nombres pseudo-aléatoires (PRNGs) est un module fondamental pour modéliser correctement les variables aléatoires. Cette simulation générant un grand nombre de calculs indépendants est considérée comme facile à paralléliser dans le but d'améliorer les temps d'exécution séquentiels souvent très longs avec cette approche. Cependant, les méthodes numériques employées pour la parallélisation d'applications stochastiques sont souvent méconnues (Hill et al. 2013). De plus, si l'on ne prête pas de soin à la conception de ces simulations (séquentielles et parallèles), on ne peut pas obtenir des résultats numériques reproductibles et on ne peut pas comparer les résultats d'une simulation stochastique séquentielle avec son équivalent en parallèle. En effet, les résultats différents peuvent survenir pour la même expérience numérique déterministe du fait de l'impact des nouvelles

architectures, des compilateurs, des systèmes d'exploitation ou de l'ordre d'exécution en parallèle des opérations en arithmétique flottante (Diethelm 2012) (Hill 2015). Michaela Taüfer a récemment montré des problèmes d'absence de reproductibilité numérique lors de l'utilisation de processeurs de type GP-GPU (Taüfer 2012). De fait, nous ne regarderons pas ce type d'accélérateur dans notre manuscrit, mais nous aborderons les accélérateurs Intel Xeon Phi actuels qui proposent encore des exécutions '*in order*'.

La question de la réplication des expériences numériques est essentielle, ne serait-ce que pour mettre au point les programmes, mais également en regard de la méthode scientifique. Cependant, cet aspect est trop souvent négligé dans de nombreux articles de simulation computationnelle. Une étude récente de 2014 sur plus de 600 articles publiés dans des revues ou conférences ACM de haut niveau a montré que seulement 16.63% des articles décrivaient des résultats reproductibles (Collerge et al. 2014). Ceci entraîne de graves conséquences, non seulement dans le domaine de la simulation numérique, mais aussi et surtout dans le domaine des Sciences de la vie, avec notamment une crise de « foi scientifique ». En cas de non reproductibilité des expériences, quel crédit numérique peut-on apporter au logiciel parallèle développé ?

La recherche visant à garder la reproductibilité numérique est en train de devenir un sujet important dans la science du calcul numérique en général et dans la simulation stochastique en particulier. Il s'agit d'un critère majeur pour le débogage des programmes mais aussi pour garantir aux scientifiques que l'expérience publiée a été reproduite et qu'ils accéderont aux informations nécessaires pour obtenir eux aussi les mêmes résultats. Ce critère n'a pas été bien compris, ni pleinement pris en considération dans la simulation stochastique parallèle utilisant le calcul à haute performance.

Pour l'avancement de la Science, nous devons étudier aussi complètement que possible les causes de non-reproductibilité numérique, et plus précisément les problèmes spécifiques aux simulations stochastiques parallèles utilisant des systèmes de calcul à haute performance. Nous cherchons à proposer une méthode et/ou une approche permettant l'implémentation de logiciels stochastiques parallèles capables de produire des résultats reproductibles d'une exécution parallèle à l'autre, mais aussi comparable à une exécution séquentielle de référence (à petite échelle). Nous cherchons à réaliser ce type de reproductibilité numérique, non seulement sur une configuration matérielle et logicielle identique, mais aussi idéalement sur des contextes informatiques différents, le cas échéant nous cherchons à expliquer autant que possible pourquoi nous avons des différences.

2 - Plan détaillé

Dans ce manuscrit, nous débutons par un chapitre détaillant la problématique, où nous présentons les facteurs de non-reproductibilité d'une simulation en général. Puis nous abordons le cas des simulations stochastiques parallèles de type Monte Carlo utilisant des systèmes de calcul à haute performance.

Nous présentons dans le deuxième chapitre, un l'état de l'art dédié aux simulations stochastique de type Monte Carlo, des générateurs de nombres pseudo-aléatoires et de leurs techniques de parallélisation pour la réalisation des simulations stochastiques parallèles.

Dans le troisième chapitre, nous présentons le calcul parallèle, les architectures classiques du parallélisme ainsi que les architectures de type multi-cœurs et *manycore*. Nous présentons aussi les différents modèles de programmation parallèle avant de terminer par un aperçu des nouvelles architectures du calcul à haute performance (puces neuro-morphiques et accélérateurs quantiques).

Ensuite, nous présentons d'une part l'existant pour résoudre les problèmes de reproductibilité numérique, d'autre part, nous montrons les limitations pour le développement d'une simulation stochastique parallèle reproductible dans le quatrième chapitre. Nous identifions les apports de la reproductibilité numérique et nous présentons quelques méthodes autorisant une répétabilité numérique permettant d'obtenir des résultats identiques entre les implémentations parallèles et séquentielles. De plus, nous proposons notre approche, notre implémentation et notre outil « RSPR » pour lancer une simulation stochastique en parallèle et garantir la répétabilité, la reproductibilité numérique. La construction de cet outil est basée sur une approche *Single Program Multiple Data* qui correspond bien à l'approche MRIP (*Multiple Replications in Parallel*).

Le cinquième chapitre montre comment appliquer les principes avancés dans le chapitre précédent. Et aussi, aune implémentation de simulations stochastiques parallèles permett d'obtenir les mêmes résultats sur des architectures multi-cœurs et manycore (à base d'Intel Xeon Phi).

Enfin, la conclusion générale nous présente une discussion et un bilan du travail accompli dans cette thèse avant de passer aux perspectives à court terme et long terme.

Chapter 1 - Problématique

I - Introduction

La simulation est une méthode importante pour la recherche (Banks 1998) (Traore et Hill 2003). La simulation est appliquée dans de très nombreux domaines pour améliorer nos connaissances sur les systèmes étudiés et pour mettre au point des méthodes sur des modèles et pour réaliser des prévisions lorsque c'est possible. Elle permet d'optimiser, de tester et de valider des modèles proposés pour décrire des systèmes complexes qu'il n'est possible d'étudier que par simulation. Le présent travail de thèse se concentre particulièrement sur la simulation stochastique pour les calculs scientifiques, elle utilise des sources pseudo-aléatoires et très souvent la méthode de Monte Carlo (Coquillard et Hill 1997) (Raychaudhuri 2008). Pour ce type de simulations, le générateur des nombres pseudo-aléatoires (PRNGs) est un module fondamental pour la génération des variables aléatoires. On considère généralement que cette famille de simulations est facile à paralléliser pour améliorer ses performances. Les événements aléatoires dans la simulation sont simulés en reproduisant les lois de probabilité correspondantes. La simulation stochastique parallèle peut se baser sur la parallélisation de la simulation de variables aléatoires indépendantes. Pour cela on crée de multiples flux indépendants de nombres pseudo-aléatoires soit avec un seul générateur de nombres pseudo-aléatoires (PRNGs) soit avec plusieurs générateurs et des techniques de parallélisation de PRNGs (Hill et al. 2013).

La simulation stochastique parallèle permet d'optimiser la performance en temps de calcul (Jeffers and Reinder 2013). Elle peut passer facilement à l'échelle si les calculs sont indépendants (ce qui est très majoritairement le cas) et tirer bénéfice d'importantes puissances de calcul. Les systèmes de calcul à haute performance (HPC) sont conçus pour réduire les temps d'exécution, ils sont cependant plus adaptés aux calculs dit « compute bound » qui font plus de calcul que d'accès mémoire. Ces derniers sont dit « memory bound » et les codes qui font un grand usage des communications entre processus parallèles sont en général classés « I/O bound ». Le calcul à haute performance prend une place croissante dans la stratégie et les investissements des organismes de recherche publics et privés, notamment dans les domaines innovants tels que les NBIC (Nanotechnologie, Biologie, Informatique et Cognition) (PRACE report 2012) (PRACE report 2013). Le HPC est aussi devenu un outil indispensable pour le développement scientifique et industriel en Europe (PRACE report 2012), dans les domaines comme les sciences de la terre (météorologie et climatologie, …), les sciences de la vie (pharmacie, médecine, biologie, génomique, ...), la science des matériaux, la chimie,

l'astrophysique, la physique des plasmas, les sciences de l'ingénieur et les applications industrielles (industrie automobile, aéronautique, aérospatiale). Le développement du calcul à haute performance a conduit à des systèmes petaflopiques et les systèmes exaflopiques sont le prochain objectif important pour l'horizon 2020.

"*Le calcul à haute performance est critique pour les industries qui requièrent de la précision et de la vitesse comme : l'automobile et l'aviation, et le secteur de la santé. L'accès rapide aux simulations réalisées par la constante d'amélioration des supercalculateurs, peut faire la différence entre la vie et la mort, entre les nouveaux emplois et les profits ou la faillite*" (PRACE report 2012) et (PRACE report 2013).

Au cours de ces dernières années, grâce aux progrès technologiques, nous avons vu l'apparition d'architectures hybrides à base d'accélérateurs matérielsde type manycore à base par exemple de processeurs graphiques de type GP-GPUs (General Purpose Graphical Processing Units) et plus récemment de coprocesseurs Intel Xeon Phi. Cette approche est une généralisation des unités d'Hyper-Threading au sein des processeurs multi-cœurs classiques apparus depuis 2003. Grâce à ces innovations, des simulations stochastiques complexes et à grande échelle ont pu être effectuées plus rapidement, entraînant une augmentation considérable des publications scientifiques de résultats de simulations stochastiques parallèles. Une question demeure pour l'avancement de la Science, qu'en est-il de la reproductibilité des résultats ainsi publiés ? Selon (Diethelm 2012) (Taufer 2012) (Hill 2015), la reproductibilité des résultats de simulations est limitée sur les systèmes de calcul à haute performance. La recherche de reproductibilité, essentielle à toute approche scientifique, est négligée dans de nombreux articles de simulation computationnelle. Ceci entraîne de graves conséquences, non seulement dans le domaine de la simulation numérique, mais aussi et surtout dans le domaine des Sciences de la vie, avec notamment ***une crise de « foi scientifique »***. La recherche de la reproductibilité devient un problème important dans la science du calcul en général et dans la simulation stochastique utilisant le calcul à haute performance en particulier.

Dans ce chapitre, nous allons exposer tout d'abord les facteurs de non-reproductibilité d'une simulation numérique. Ensuite, nous présenterons plus spécifiquement les problèmes posés par la non-reproductibilité des simulations stochastiques parallèles utilisant des systèmes de calcul à haute performance.

II - Facteurs de non-reproductibilité d'une simulation numérique

Beaucoup de facteurs peuvent rendre une expérience non-reproductible. Selon Jimenez (Jimenez et al. 2014), les causes de la non-reproductibilité du résultat d'une expérience computationnelle sont : le matériel (le processeur (CPU), le système d'exploitation, les configurations, les bibliothèques) ; l'utilisateur (les scripts et les bibliothèques externes (« 3rd parties ») ; les paramètres/données d'entrée ; les services externes (les bases de données, le système de fichiers parallèles, etc.).

En général, lorsque l'on parle d'« une expérience computationnelle », il s'agit des résultats calculés d'une expérience numérique en informatique, qui inclut bien sûr la simulation numérique. Cependant, nous constatons que la classification et les causes de non-reproductibilité proposées dans la littérature ne sont pas suffisantes, particulièrement dans le cas de la simulation stochastique parallèle utilisant le calcul à haute performance. Nous avons proposé de classer les causes de non-reproductibilité d'une simulation numérique en cinq catégories : la culture de publication scientifique, le calcul en virgule flottante, le matériel, le logiciel et la technique utilisée pour la distribution et la parallélisation des flux stochastiques (Dao et al. 2014b) et (Dao et al. 2016).

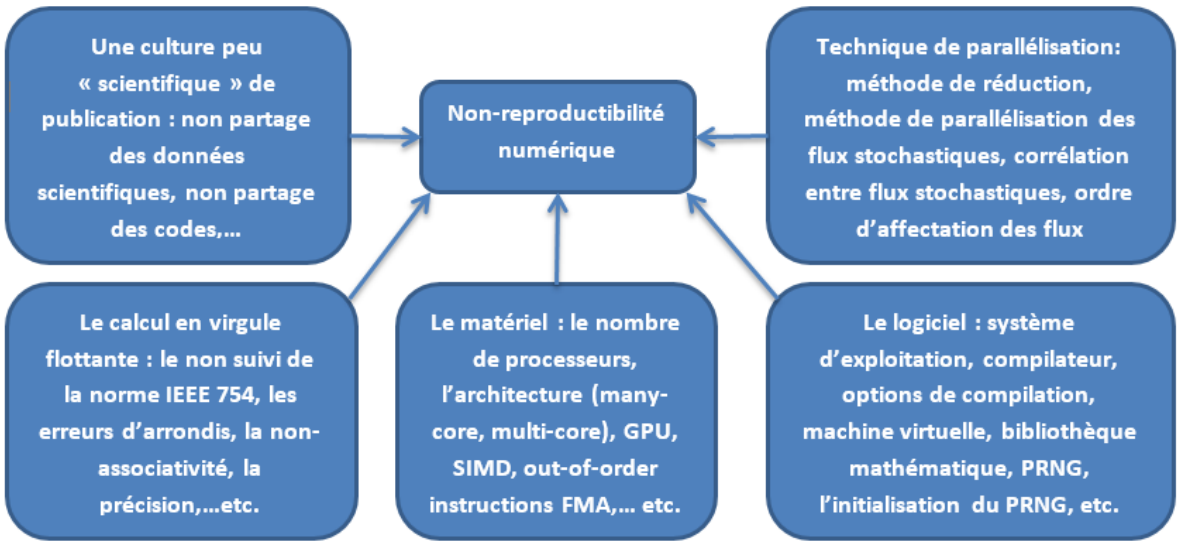


Figure 1.1. Les principales sources de non-reproductibilités numériques dans le contexte des simulations stochastiques parallèles et distribuées.

Dans ce qui suit, nous présentons en détail ces cinq catégories listées dans la Figure 1.1.

II.1 - La culture de la publication scientifique

Une raison majeure de la non-reproductibilité dans les simulations numériques scientifiques est que dans de nombreux domaines scientifiques le comportement des scientifiques est très individualiste.

Les chercheurs, développeurs et scientifiques publient leurs articles avec des résultats de qualité mais souvent ne partagent pas leurs données, leurs codes sources ou ne présentent pas leurs méthodes avec suffisamment de détails.

Ce premier facteur a été mis en évidence dans le cadre d'une réflexion sur la culture du partage des informations dans le domaine de la recherche. Pawlikowski et ses collègues (Pawlikowski et al. 2002) ont en effet étudié et présenté le problème de non-reproductibilité des simulations stochastiques dans le domaine des réseaux de télécommunication. Ils ont fait une enquête sur 2246 articles publiés dans les actes des conférences IEEE INFOCOM (1992-1998), IEEE Transactions on Communications (1996-1998) et IEEE/ACM Transactions on Networking (1996-1998) et dans le journal scientifique Performance Evaluation Journal (1996-1998). Selon cette étude, la simulation à événements discrets stochastiques est devenue un outil couramment utilisé par les scientifiques et les ingénieurs du domaine, contribuant à plus de 51% de tous les résultats de recherche publiés. Néanmoins, cet enthousiasme n'est pas partagé par tous les développeurs de simulation et les utilisateurs finaux. En effet, même s'il y a beaucoup de publications intéressantes, personne ne partage ses codes, ses données, etc.

This enthusiasm is not shared by all simulation developers and users. Some claim that stochastic simulation as a performance evaluation tool of various dynamic systems, including telecommunication networks, is misused, and that the spread of this phenomenon is so wide that one can speak about ***a deep credibility crisis****. It is even claimed that one cannot rely on the majority of the published results of performance evaluation studies of dynamic systems based on stochastic simulation. Editorials such as [5] or panel discussions of specialists on the topic, such as those organized at the Winter Simulation Conference in 1994 and 1996, or at the IEEE INFOCOM in 1996, have not changed the situation. (Pawlikowski et al. 2002)*

Kurkowski et ses collègues (Kurkowski et al. 2005) ont également présenté une enquête basée sur 151 articles publiés à la conférence ACM's MobiHoc de 2000 à 2005. Se concentrant sur des articles complets, Kurkowski a conclu que moins de 15% des articles publiés sont reproductibles.

Simulator and Environment		
Totals	Percentage	Description
114 of 151	75.5%	Used simulation in the research
0 of 114	**0%**	Stated the code was available to others

80 of 114	70.2%	Stated which simulator was used
22 of 80	27.3%	Used self-developed or custom simulators
7 of 58	12.1%	Stated which version of the public simulator was used
8 of 114	7.0%	Addressed initialization bias
0 of 114	0%	Addressed the PRNG used

Table 1.1. Les résultats de l'enquête sur 151 articles publiés dans la conférence ACM's MobiHoc de 2000 à 2005 (Kurkowski et al. 2005)

La table 1.1 présente certains résultats intéressants tirés des travaux de Kurkowski. Selon ses études, 75.5% des publications ont recours à l'expérimentation numérique mais aucune ne propose le partage de codes, 12.1% mentionnent la version du simulateur publique utilisé et seulement 7.0% ont abordé des questions importantes comme le biais d'initialisation. Les études de Pawlikowski et Kurkowski sont détaillées dans une publication de Dalle (Dalle 2012) sur la reproductibilité et la traçabilité des simulations.

When citing reasons not to share data and code, the top of the list of factors preventing the sharing of data was time to prepare for release (cited by 56% percent of respondents) and concern over lack of citation in downstream use (44%), then legal barriers such as copyright (41%). The story is similar for code: the top reason for not sharing is time to prepare (78%), then dealing with questions from users (52%), concerns over use without citation (45%), and third the possibility of patents or other IP constraints (40%) and legal barriers such as copyright (32%). (Stodden 2010)

Stodden (Stodden 2010) a fait une enquête auprès de 638 personnes sur le partage de codes, de données et d'idées à la conférence Neural Information Processing Systems – NIPS, une très grosse conférence du domaine des neurosciences. Son enquête met en lumière différentes raisons pour le non-partage des données et des codes. On peut citer notamment le temps nécessaire pour nettoyer le code et le rendre présentable, les données et les documents, la perte potentielle de publications futures, la crainte que le travail initial ne soit pas cité, le droit d'auteur, etc.

Une autre étude importante de Vandewalle, Kovacevic et Vetterli (Vandewalle et al. 2009) porte sur 134 articles publiés dans le journal de traitement d'image (IEEE Transactions on Image Processing) en 2004. Ils ont demandé pour chacun des articles à deux ou trois examinateurs de vérifier sa reproductibilité à l'aide d'une courte liste de questions sur trois aspects de la reproductibilité : l'algorithme, le code et les données. Ils ont ainsi répété une étude plus ancienne de Kovacevic portant sur 15 articles (Kovacevic 2007).

Finally, code (9%) and data (33%) are only available online in a minority of the cases, with data being available more often thanks to the frequent use of standard image data sets, such as Lena. Remark, however, that several versions of many of those popular test images exist, which might therefore still introduce uncertainty about reproducibility. An issue with URLs, if they are mentioned in a paper, is their generally limited lifetime. Several reviewers reported URLs in a paper that had become invalid. (Vandewalle et al. 2009)

Selon cette étude, 91% des articles omettent de fournir le code et 67% les données. En d'autres mots, quasiment toutes les publications de 2004 ne sont pas reproductibles.

Les études de Stodden, Vandewalle et al sont référencées dans l'étude de Collerge et ses collègues (Collerge et al. 2014). Collerge a conduit une analyse de la reproductibilité dans la recherche en informatique sur 613 articles dans 8 conférences ACM dont 515 publications computationnelles. Les auteurs de 105 articles sur les 613 ne sont pas joignables tandis que 179 ne répondent pas au courrier. Comme le montre la figure 1.2, 129 codes échouent à la compilation ou au lancement. Il recense au final seulement 102 publications reproductibles.

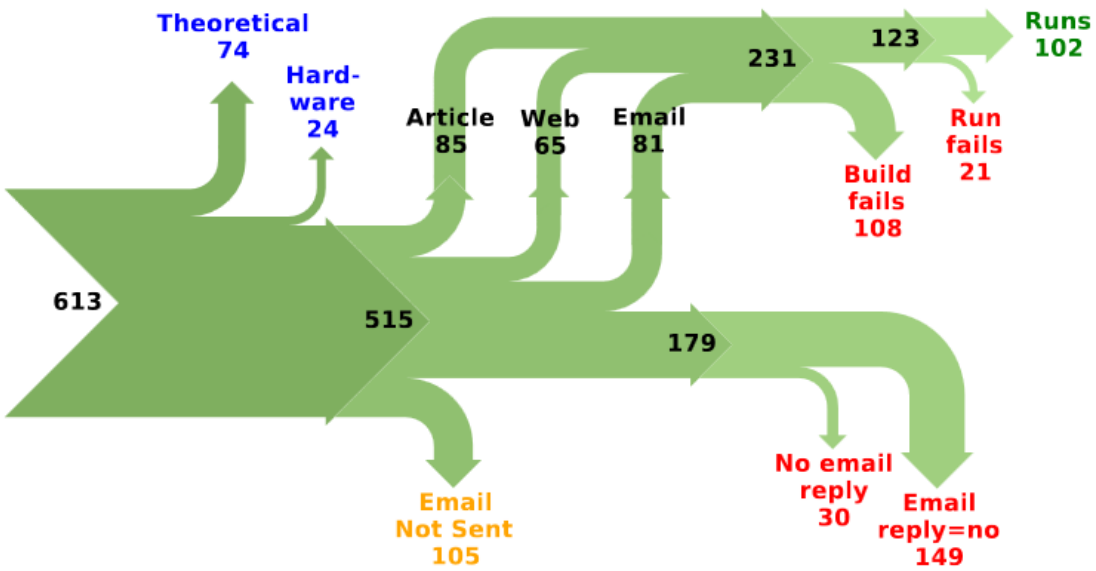


Figure 1.2. Les résultats d'une l'étude de reproductibilité conduite sur 613 articles de recherche en informatique : les résultats en bleu correspondent aux papiers qui n'ont pas été analysés, les résultats en rouge indiquent les causes de non-reproductibilité et le résultat en orange correspond aux articles dont les auteurs n'ont pu être contactés (Collerge et al. 2014).

Nussbaum (Nussbaum 2015) a aussi fait une bonne synthèse de la situation dans la communauté de recherche en informatique à partir des travaux de Lukowicz (Lukowicz et al. 1994), de Zelkowitz et

Wallace (Zelkowitz et Wallace 1998), et de Naicken (Naicken et al. 2007). Selon ces auteurs, en 1994, 40-50% des 400 articles publiés dans les journaux ACM nécessitent une validation expérimentale qui était absente des articles. En 1998, une enquête sur 621 articles dans les journaux IEEE Transactions on Software Engineering, IEEE Software et dans les actes de la conférence internationale sur l'ingénieur logiciel, a révélé de manière similaire que trop d'articles n'ont aucune validation expérimentale ou utilisent une méthode informelle de validation (assertion). En 2007, la plupart des articles dans la recherche P2P utilisent un simulateur indéterminé ou personnalisé.

Hofner (Hofner et al. 2015) a également étudié la reproductibilité de 56 articles acceptés dans le journal Biometrical de Mars 2014 à Février 2015.

Reproducibility issues	
Missing data or code	25 (44.6%)
Code produced errors	21 (37.5%)
Code ran but…	
… did not reproduce results	9 (16.1%)
… did not reproduce all tables/figures	29 (51.8%)
No seed used for RNG	14 (25.0%)
Code and paper were difficult to match	23 (41.1%)
Documentation issues	
Missing README	45 (80.4%)
Bad documentation of code and functions	23 (41.1%)
Unnecessary comments and code	17 (30.4%)
Code supplied as PDF/Word/…	8 (14.3%)
Programming style issues	
Usage of absolute paths	10 (17.9%)
Bad coding style	12 (21.4%)
Too many code files/difficult file names	6 (10.7%)

Table 1.2. Vue d'ensemble des problèmes de reproductibilité, de documentation et de style de programmation détectés des articles acceptés pour la publication dans le journal Biometrical (Hofner et al. 2015)

Comme le montre la table 1.2, le problème le plus fréquent est le manque de disponibilité du code ou des données (44.6% des articles), suivi d'un échec à l'exécution du code (37.5%). Même dans le cas où le code marche, il ne reproduit pas les résultats numériques publiés dans 16,1% des cas ou pas toutes les tables et les figures dans 51,8 % des cas. 25% des articles présentant des résultats de

calculs stochastiques ne fournissent pas l'initialisation du générateur de nombres pseudo-aléatoires. Un autre problème grave est que pour 41.1% des articles il est difficile d'associer le code et le papier. Ces quelques chiffres impressionnants de l'enquête de Hofner montrent bien les difficultés rencontrées dans la production scientifique. La culture actuelle est un obstacle considérable à la vérification des résultats numériques publiés.

Les études ci-dessus révèlent la partie visible de l'iceberg. Clairement, la propension à la rétention d'information affecte négativement la recherche scientifique, spécialement la reproductibilité numérique. Stodden (Stodden et al. 2013) a qualifié ces obstacles de « culture scientifique de non-reproductibilité », qui conduit à une crise de crédibilité scientifique (Pawlikowski et al. 2002).

Indeed, the spread of this phenomenon is so wide that one can speak about a ***deep crisis of credibility****. (Pawlikowski et al. 2002)*
Unfortunately, the scientific culture surrounding computational work has evolved in ways that make it difficult to verify findings, efficiently build on past research, or even apply the basic tenets of the scientific method to computational procedures. (Stodden et al. 2013)

La non-reproductibilité des résultats retarde le développement de la science et accroit les coûts par la duplication des efforts. Nous avons choisi de présenter la culture scientifique comme première cause de non reproductibilité, car il est impossible d'essayer de reproduire une expérience si les informations nécessaires à cette action ne sont pas fournies.

Nous allons nous intéresser maintenant à une deuxième cause : l'environnement et le type d'exécution.

II.2 - L'environnement et le type d'exécution

La reproductibilité numérique est fortement impactée par le choix du logiciel et du matériel (Dao et al. 2014b) (Dao et al. 2016). Ces choix touchent de multiples paramètres : compilateurs, options et versions du compilateur, bibliothèques associées et supplémentaires, machine virtuelle, système d'exploitation, pseudocode, algorithme, langages de programmation, architecture matérielle, nombre de cœurs physiques et logiques ou nombre de processeurs, etc. Nous mettons les deux causes ensemble parce qu'elles sont étroitement liées.

Considérons tout d'abord l'impact du logiciel sur la non-reproductibilité numérique. Dès 1991, Goldberg (Goldberg 1991) a fait le commentaire suivant sur le choix du langage de programmation, des débogueurs et du compilateur.

When a program is moved between two machines and both support IEEE arithmetic, ***if any intermediate result differs, it must be because of software bugs not differences in arithmetic…***
*Remarkably enough, some languages do not clearly specify that if x is a floating point variable (with say a value of 3.0/10.0), then every occurrence of (say) 10.0*x must have the same value…*
Another ambiguity in most language definitions concerns what happens on overflow, underflow, and other exceptions… concerns the interpretation of parentheses…
Compiler texts tend to ignore the subject of floating point… (Goldberg 1991)

Et très récemment, Glatard (Galatard et al. 2015) a également présenté tous les éléments qui peuvent influencer la non-reproductibilité numérique d'une exécution. Ce sont le code source, le processus de compilation, les bibliothèques, le noyau et le matériel.

…the execution of an application depends on its source code, on the compilation process, on software libraries, on an operating system (OS) kernel, and on a hardware processor. Libraries may be embedded in the application, i.e., statically linked, or loaded from the OS, i.e., dynamically linked. The reproducibility of results may be influenced by any variation in these elements, in particular: versions of the source code, compilation options, versions of the dynamic and static libraries (in particular when these libraries implement mathematical functions), or architecture of hardware systems. Some programming languages, for instance MATLAB, Java, Python, Perl, and other scripting languages, additionally rely on specific runtime software, which can further influence the results. (Glatard et al. 2015)

Sur la question du matériel, Goldberg a également commenté le manque de considération des concepteurs de matériels pour l'analyse numérique. Il a ainsi pointé l'incompatibilité entre le matériel conçu pour gérer les opérations en virgule flottantes selon la norme IEEE 754 et les langages de programmation tels que C, Pascal ou FORTRAN, ...

Computer system designers rarely get guidance from numerical analysis texts, which are typically aimed at users and writers of software not at computer designers…

Thus, there is usually a mismatch between floating-point hardware that supports the standard and programming languages like C, Pascal or FORTRAN… (Goldberg 1991)

En ce qui concerne les architectures matérielles différentes, on trouve un bon exemple de non-reproductibilité numérique avec l'arithmétique binaire 32 bits IEEE-754 dans l'article de Jézéquel (Jézéquel et al. 2015). La table 1.3 donne les différents résultats obtenus lorsqu'on effectue un calcul selon deux schémas identiques en 3 points de l'espace sur différents processeurs et avec des compilateurs différents (voir Table 1.3).

	Point in the space domain		
	p_1: (0, 19, 62)	p_2: (50, 12, 2)	p_3: (20, 1, 46)
AMD Opteron CPU with gcc			
scheme 1	-1.110479	54.54238	614.1038
scheme 2	-1.110426	54.54199	614.1035
NVIDIA C2050 GPU with CUDA			
scheme 1	-1.110204	54.54224	614.1046
scheme 2	-1.109869	54.54244	614.1047
NVIDIA K20c GPU with OpenCL			
scheme 1	-1.109953	54.54218	614.1044
scheme 2	-1.111517	54.54185	614.1024
AMD Radeon GPU with OpenCL			
scheme 1	-1.109940	54.54317	614.1038
scheme 2	-1.110111	54.54170	614.1044
AMD Trinity APU with OpenCL			
scheme 1	-1.110023	54.54169	614.1062
scheme 2	-1.110113	54.54261	614.1049

Table 1.3. Les résultats de transformations de trois points en 3 dimensions avec différents processeurs et compilateurs (Jézéquel et al. 2015)

Jézéquel a également présenté 3 scénarios d'exécution différents afin de détecter les problèmes de reproductibilité numérique : d'une exécution à l'autre dans des unités de traitement accéléré de type APU (« Accelerated processing unit) ou GPU, d'une implémentation de schéma (différences finies) à l'autre et finalement d'une architecture à l'autre.

Plus récemment, Corden a aussi fait un commentaire très important sur la famille de processeurs Intel Xeon. Ce commentaire concerne la limitation de la reproductibilité entre les architectures ou les matériels d'Intel Xeon classique et Xeon Phi si l'on considère une comparaison « bit-for-bit » (Corden 2013).

Sur une même machine ou un même matériel, on peut aussi obtenir des résultats différents en changeant le nombre des cœurs. Un bon exemple a été présenté par Diethelm (Diethelm 2012). Il a comparé les résultats du calcul de l'épaisseur d'une feuille métallique avec une simulation numérique utilisant 4 processeurs ou un seul. Les résultats des deux simulations étaient différents (voir Figure 1.3).

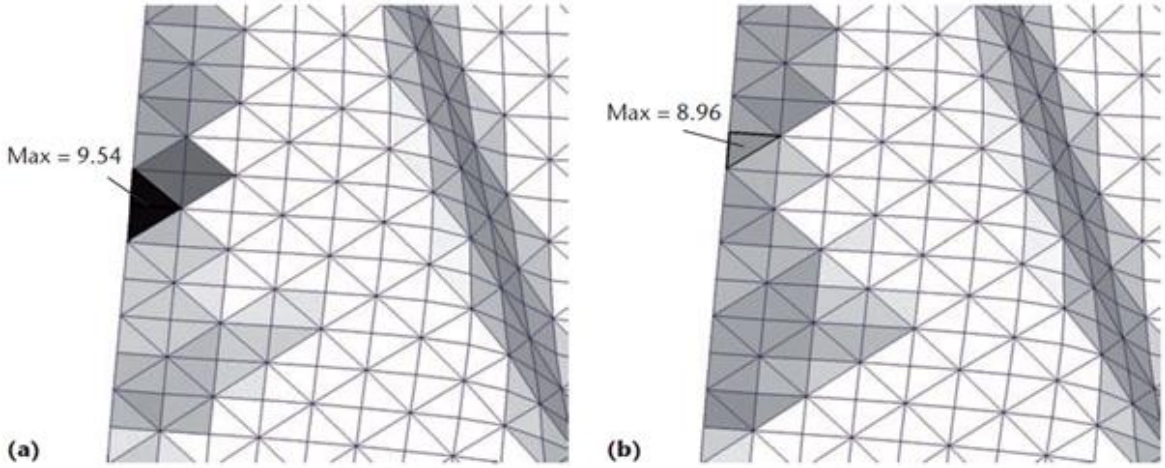


Figure 1.3. Les résultats différents des simulations sur les nombres différents de processeurs. La valeur maximale de la simulation est donnée avec un calcul sur un processeur (a), puis sur 4 processeurs (b) (Diethlem 2012).

Sur la Figure 1.3, on peut voir que les valeurs maximales obtenues par la simulation avec respectivement un processeur (= 9.54) et quatre processeurs (= 8.96) sont bien différentes. La table 1.4 présentent les valeurs extrêmes obtenues pour différentes configurations de la simulation. Les résultats à 4 processeurs, y compris les valeurs minimales et maximales, diffèrent dans deux mises en œuvre successives bien que ces différences soient petites.

Description of the simulation (no. of processors)	Minimal value of the sheet thickness change (%)	Maximal value of the sheet thickness change (%)
1	-29.21	+9.54
2	-29.34	+9.14
3	-29.04	+9.02
4 (1st attempt)	-29.04	+8.98
4 (2nd attempt)	-29.09	+8.96

Table 1.4. Valeurs minimale et maximale des changements de l'épaisseur d'une feuille métallique pour différentes configurations de la simulation (Diethlem 2012)

Ces résultats illustrent l'impact du nombre de processeurs et même **ce qui est sensiblement plus grave**, **la difficulté à reproduire deux fois les mêmes résultats en conservant le même nombre de processeurs**.

Un autre exemple d'échec dans la reproductibilité numérique en cas de changement du nombre de processeurs est décrit par Langlois (Langlois et al. 2015) sur le calcul de la suite ouverte Telemac-Mascaret.

	the sequential run	a 64 procs run	a 128 procs run
depth H	0.350012E-01	0.**2748817**E-01	0.**1327634**E-01
velocity U	0.4029747E-02	0.**4935279**E-02	0.**4512116**E-02
velocity V	0.7570773E-02	0.**3422730**E-02	0.**7545233**E-02

Table 1.5. L'échec de reproductibilité de la simulation DAM BREAK (Langlois et al. 2015)

Cet exemple s'appuie sur la simulation 2D de la catastrophe de la rupture du barrage de Malpasset (Malpasset Dam Break) où 433 personnes périrent en 1959, effectuée avec une résolution par éléments finis des équations de Saint-Venant. La table 1.5 montrent les résultats obtenus pour trois paramètres : la profondeur H et les vélocités U et V. Les résultats obtenus par exécution séquentielle diffèrent souvent très sensiblement de ceux obtenus avec des exécutions parallèles sur 64 processeurs et 128 processeurs.

Un autre paramètre de l'environnement d'exécution responsable de problèmes de non reproductibilité est l'utilisation du générateur de nombres pseudo-aléatoires (Dao et al. 2014a) avec différents systèmes d'exploitation, différentes machines virtuelles partageant le même langage et compilateur mais gérant différemment les threads, des architectures différentes, des compilateurs différents avec le même matériel et système d'exploitation, des plateformes différentes entre la machine virtuelle et la machine réelle, etc.

Hill (Hill 2015) a résumé les problèmes de non-reproductibilité numérique liés à l'environnement d'exécution que nous venons de présenter. Il a aussi ajouté l'impact des différents langages de programmation et l'habitude de non-observation d'une exécution à l'autre même si elle est dans le même contexte. On oublie de vérifier la répétabilité d'une exécution à l'autre car on considère à tort que c'est un acquis.

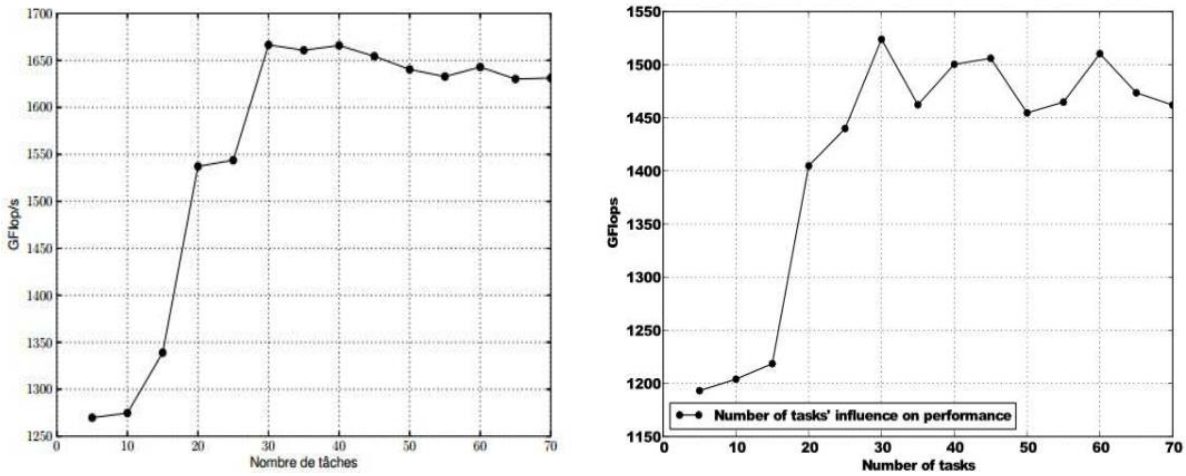


Figure 1.4. Les résultats obtenus pour le temps d'exécution dans la publication originale à gauche et dans l'article de Nussbaum à droite (Nussbaum et Richard 2014).

Nussbaum et Richard (Nussbaum et Richard 2014) ont étudié la reproductibilité expérimentale des articles publiés à ComPas'2014 en termes de parallélisme, d'architecture et de système. A côté des erreurs de compilation et d'exécution (y compris sur le choix de la version de gcc, sh et bash), ils se sont intéressés aux temps d'exécution qui sont différents (spécifiques aux machines, mais il n'a pas été possible de se connecter à la machine distante pour mesurer le temps d'exécution publié). Comme le montre la figure 1.4, leurs résultats sont très différents de la publication originale. Il faut souligner que, si une expérience numérique (y compris la simulation stochastique parallèle) ne marche pas sur un support matériel, c'est un problème de « portabilité » qui est à la base de la non-reproductibilité. Si elle marche sur un autre support matériel mais que les résultats obtenus sont différents, alors on peut dire qu'elle n'est pas numériquement reproductible. Si les résultats obtenus sont presque identiques, on considère que la reproductibilité est présente, mais que l'on n'a pas de répétabilité (on peut perdre la capacité à déboguer certaines applications ce qui serait très grave).

Les exemples que nous venons de présenter montrent que l'environnement et le type d'exécution sont des obstacles importants à la reproductibilité des résultats de la simulation numérique. L'augmentation du nombre de cœurs, d'éléments de traitement ou la mise en œuvre de la même simulation sur un système plus puissant ont pour but d'accélérer le temps d'exécution mais les résultats obtenus ne sont souvent pas identiques entre les exécutions séquentielle et parallèle, et même entre exécutions parallèles. Cela a conduit à un conflit entre l'obtention de résultats de qualité et la performance du calcul.

II.3 - Le calcul en virgule flottante

La majorité des résultats de calcul avec des nombres réels obtenus lors d'expériences computationnelles ne sont pas reproductibles à cause des limites liées aux opérations en virgule flottante. Dans cette partie, nous présentons deux causes de non-reproductibilité liées aux opérations en virgule flottante : les problèmes liés aux opérations en virgule flottante avec la norme IEEE 754 et les bibliothèques mathématiques de mauvaise qualité utilisées dans les simulations numériques. Pour la première cause, on parle généralement de l'impact du type « nombre en réel ». Pour la deuxième, on parle directement de l'impact des bibliothèques qui gèrent de nombreuses opérations en virgule flottante pour notre simulation numérique.

II.3.a - Les opérations en virgule flottante et la norme IEEE 754

La virgule flottante (floating-point) est une méthode pour représenter l'*approximation* d'un nombre réel sur un ordinateur avec une mantisse (significande), une base, un exposant et un signe. C'est un type de donnée numérique de base important en l'informatique et on le trouver partout dans tous les systèmes de calcul informatique, du matériel au logiciel (Goldberg 1991).

> *Almost every language has a floating-point datatype; computers from PCs to supercomputers have floating-point accelerators; most compilers will be called upon to compile floating-point algorithms from time to time; and virtually every operating system must respond to floating-point exceptions such as overflow. (Goldberg 1991)*

Pour les langages de programmation de haut niveau, les nombres réels sont représentés par les types de donnée « float », « double » qui sont des types qui utilisent les virgules flottantes. Avec les calculs en virgule flottante, chaque système fournit des bibliothèques mathématiques normalisées. Les compilateurs supportent également les types de données en virgule flottante et les adaptent aux langages de programmation. Les ordinateurs de monsieur tout le monder ainsi que les supercalculateurs font des calculs en virgule flottante grâce à leurs accélérateurs implémentés dans des unités arithmétiques et logiques spécifiques appelés Unités Virgule Flottante (FPU ou Floating point Units en anglais).

La norme IEEE-754 pour l'arithmétique binaire en virgule flottante est celle qui est le plus couramment utilisées. Elle est disponible sur presque toutes les machines modernes, y compris les dernières générations de GPUs et les derniers accélérateurs de la famille Intel Xeon Phi

(« manycores »), ou encore sur les processeurs Cell et plus récemment les processeurs Power d'IBM. Tous les langages de programmation du calcul à haute performance (Fortran, C et C++) doivent imposer le respect de la norme IEEE-754. Malheureusement, par défaut, certains langages utilisent les 10 octets disponibles au sein des FPU Intel modernes pour les calculs de précision, mais ils ignorent souvent des propriétés d'arrondi de la norme IEEE-754. C'est notamment le cas de Java. Ceci a des impacts très significatifs sur les résultats. Entre 1998 et 2004, Kahan et Darcy (Kahan et Darcy 1998) signalaient déjà des problèmes de ce type principalement avec le langage Java.

*Java's floating-point arithmetic is blighted by **five gratuitous mistakes**:*

1. Linguistically legislated exact reproducibility is at best mere wishful thinking.

2. Of two traditional policies for mixed precision evaluation, Java chose the worse.

3. Infinites and NaNs unleashed without the protection of floating-point traps and flags mandated by IEEE Standards 754/854 belie Java's claim to robustness.

4. Every programmer's prospects for success are diminished by Java's refusal to grant access to capabilities built into over 95% of today's floating-point hardware.

5. Java has rejected even mildly disciplined infix operator overloading, without which extensions to arithmetic with everyday mathematical types like complex numbers, intervals, matrices, geometrical objects and arbitrarily high precision become extremely inconvenient.

To leave these mistakes uncorrected would be a tragic sixth mistake. (Kahan et Darcy 1998)

Kahan et Darcy pointent cinq erreurs majeures dans l'arithmétique en virgule flottante du langage Java et le fait de ne pas corriger ces erreurs serait une sixième erreur tragique. Bloch (2008) a représenté ce problème dix ans plus tard à l'article 48 de son ouvrage avec une recommandation pour éviter l'utilisation « float », « double » dans le langage Java. Bloch propose l'utilisation du type BigDecimal beaucoup plus lent mais bien plus sûr pour le calcul.

Dans la dernière version de la norme IEEE 754 sortie en 2008, les limitations clairement identifiées au niveau de l'associativité (concernant l'ordre des opérations) et de la gestion des erreurs d'arrondi sur l'addition et la multiplication ont été à nouveau mises en évidence pour l'ensemble de la communauté scientifique. De nombreux travaux signalent ces limites dans le contexte de la reproductibilité (Villa et al. 2009) (Demmel et Nguyen 2013) (Revol et Théveny 2014) (Whitehead et Fit-Florea 2011) (Hill 2015) (Jézéquel et al. 2015) et rappellent les fondamentaux souvent méconnus que Goldberg enseignait dès 1991 (Goldberg 1991).

…floating point operations like addition are not associative, so computing sums in different orders often lead to different results. (Demmel et Nguyen 2013)
As floating-point addition is not associative (neither is multiplication), the order according to which the operations are performed matters. Let us illustrate this with an example in double precision: RN(1 + (2^{100} - 2^{100})) (where RN stands for rouding-to-nearest) yields 1, since RN(2^{100} - 2^{100}) = 0, where as RN((1+2^{100}) – 2^{100}) yields 0, since RN(1+2^{100})=2^{100}. The last result is called a catastrophic cancellation: most or all accuracy is lost when close (and even equal in this example) large numbers are subtracted and only roundoff errors remain, hiding the meaningful result. (Revol et Théveny 2014)

L'exemple tiré de (Revol et Théveny, 2014) illustre ce que les auteurs appellent une « catastrophic cancellation ». Avant, Einarsson (Einarsson 2005) avait aussi mis en évidence une chaîne de problèmes dans le calcul numérique (avec virgule flottante et la norme IEEE 754) à cause de l'usage de l'arrondi. Nous allons discuter l'impact de la précision en virgule flottante dans le calcul scientifique dans la partie II.3.b de ce chapitre.

Two problems in numerical computation are that often the input values are not known exactly, and that some of the calculations cannot be performed exactly. The errors obtained can cooperate in later calculations, causing an error growth, which may be quite large. Rounding is the cause of an error, while cancellation increases its effect and recursion may cause a build-up of the final error (Einarsson 2005).

Whitehead (Whitehead et Fit-Florea 2011) ont également présenté deux exemples de limitation des opérations en virgule flottante : une limitation sur les opérations de fusion multiplication-addition (FMA) avec une seule étape d'arrondi (voir ci-dessous) et une autre sur le produit scalaire (« dot product » en anglais, c'est une opération algébrique, elle prend deux vecteurs (séquences de longueur égale de nombres et retourne un seul nombre) illustrée sur le table 1.6.

x = 1.0008
x^2 = 1.00160064
*x^2 – 1 = 1.60064 * 10^{-4} true value*
*rn(x^2 - 1) = 1.6006 * 10^{-4} fused multiple-add*
*rn(x^2) = 1.0016 * 10^{-4}*
*rn (rn(x^2) - 1) = 1.600 * 10^{-4} multiply, then add*

*In this particular case, computing rn(rn(A*A) +B) as an IEEE 754 multiply followed by an IEEE 754 add loses all bits of precision, and the computed result is 0. (Whitehead et Fit-Florea 2011).*

Celle-ci présente les résultats obtenus pour le produit de deux vecteurs de quatre éléments. Les résultats obtenus sont légèrement différents et diffèrent du calcul exact. Pour les deux exemples, on voit clairement que l'erreur d'arrondi perd tous les bits de précision après la 6ème décimale.

method	result	float value
exact	.0559587528435...	0x3D65350158...
serial	.0559588074	0x3D653510
FMA	.0559587515	0x3D653501
parallel	.0559587478	0x3D653500

Table 1.6. Les trois algorithmes (séquentiel, FMA, parallèle) donnent les résultats légèrement différents pour le produit de deux vecteurs. (White et Fit-Florea 2011).

Villa (Villa et al. 2009) a présenté un problème de non-associativité de l'accumulation en virgule flottante généré par la précision limitée et la portée de la représentation en virgulle flottant IEEE pour les calculs de gradient conjugué avec une implémentation multithread parallèle sur le Cray XMT dans le contexte d'une analyse du réseau électrique (PSE). La table 1.7 montre les valeurs de la norme Euclidienne pour différentes itérations de la méthode pondérée des moindres carrés (WLS). Les résultats en double précision ont été obtenus sur 16 processeurs du Cray XMT utilisant 100 flux par processeur pour chaque région d'exécution en parallèle. Ils diffèrent dès la quatrième itération. Nous sommes dans un cas de non reproductibilité « run to run ».

WLS iteration	Run 1	Run 2	Run 3	Diff 2 vs. 1	Diff 3 vs. 1
1	1.64E+09	1.64E+09	1.64E+09	0.00%	0.00%
2	1.88E+09	1.88E+09	1.88E+09	0.00%	0.00%
3	3.29E+07	3.29E+07	3.29E+07	0.00%	0.00%
4	4.01E+05	4.01E+05	4.01E+05	0.02%	0.01%
5	1.50E+02	1.29E+02	1.24E+02	14.25%	17.63%
6	5.92E+00	5.13E+00	7.37E+00	13.30%	24.64%
7	5.22E-01	4.46E-01	4.59E-01	14.52%	12.06%

Table 1.7. Différences dans un calcul de norme euclidienne en fonction de l'itération pour des exécutions différentes avec le même nombre de threads et le même ensemble d'entrée (Villa et al. 2009).

Villa a aussi testé la précision quadruple pour cette application mais elle ne résout pas complètement le problème de la propagation d'erreur « non-déterministe » même si elle l'a réduit sensiblement. Le principal problème est de ne pas pouvoir cerner exactement ce non déterminisme dans ce contexte parallèle, même si les sources sont connues. Cette propagation d'erreur vient des

limitations de l'associativité (de l'ordre des opérations) et de l'erreur d'arrondi dans les opérations en virgule flottante comme l'addition, la multiplication, etc. Ces problèmes de la norme IEEE 754 limitent l'obtention des mêmes résultats numériques lorsque le calcul parallèle ou multithread est considéré. La virgule flottante est une boite grise pour le matériel et les langages de programmation comme l'illustre la citation de Goldberg plus haut dans la partie II.2 de ce chapitre.

II.3.b - Les bibliothèques mathématiques et les problèmes liés à la précision des calculs

Les bibliothèques mathématiques sont très importantes pour les simulations numériques. La qualité d'une bibliothèque dépend de sa précision dans les calculs. Elles jouent malheureusement un rôle important dans la non-reproductibilité en simulation numérique.

Dans le passé, des problèmes regrettables ont été observés liés aux erreurs numériques en virgule flottante. Concrètement, la précision du calcul scientifique appliqué est en jeu comme Einarsson (Einarsson 2005) l'a bien rappelé dans son article avec plusieurs exemples : le missile Patriot en 1991, l'échange boursier de Vancouver en 1982, l'élection locale Schleswig-Holstein en 1992, etc.

Récemment, Bailey (Bailey et al. 2015) a également présenté deux exemples concrets d'applications scientifiques pour illustrer l'influence des bibliothèques mathématiques et de leur précision dans la reproductibilité numérique. Le premier exemple est sur le changement de bibliothèque mathématique dans l'application ATLAS de physique des hautes énergies et le deuxième sur l'impact de la précision arithmétique dans un modèle atmosphérique servant de composant à un grand modèle climatique.

(1) Recently, some ATLAS (acronym for "A Toroidal LHC Apparatus") researchers reported to us that in an attempt to improve performance of the code, they changed the underlying math library. When this was done, they found some collisions were missed and others were miss-identified. That such a minor code alteration (which should only affect the lowest-order bits produced by transcendental function evaluations) had such a large high-level effect suggests that their code has significant numerical sensitivities, and results may even be invalid in certain cases.
(2)…researchers working with an atmospheric simulation code had been annoyed by the difficulty of reproducing benchmark results. Even when their code was merely ported from one system to another, or run with a different number of processors, the computed data typically diverged from a benchmark run after just a few days of simulated time. As a result, ***it was very difficult even for the***

developers of this code to ensure that bugs were not introduced into the code when changes are made, and even more problematic for other researchers to reproduce their results…
After an in-depth analysis of this code, researchers He and Ding found that merely by employing double-double arithmetic (roughly 31-digit arithmetic) in two key summation loops, almost all of this numerical variability disappeared. With this minor change, benchmark results could be accurately reproduced for a significantly longer time, with very little increase in run time. (Bailey et al. 2015)

Dans le premier exemple, le changement de bibliothèque mathématique dans le code d'ATLAS a provoqué la perte de certaines collisions ou leur mauvaise identification. Ce changement avait pour but d'améliorer la performance du code mais il ne garantissait plus la reproductibilité numérique. Dans le deuxième exemple, Bailey a présenté une étude importante sur l'utilisation de la haute précision arithmétique où deux chercheurs ont amélioré significativement la reproductibilité des résultats dans le code avec l'utilisation de l'arithmétique en double-double (environ 31 chiffres arithmétiques) dans deux boucles de sommation clés. Ainsi la haute précision arithmétique se révèle parfois vraiment importante pour l'expérimentation numérique en général, mais plus particulièrement pour les grandes applications scientifiques qui font des calculs intensifs.

Bien avant, Villa (Villa et al. 2009) a présenté un autre exemple où le choix de la précision décide de la qualité des résultats calculés.

WLS iteration	Double precision			Quadruple precision		
	Run 1	Run 2	Run 3	Run 1	Run 2	Run 3
5	1.50E+02	1.29E+02	1.24E+02	1.43E+02	1.43E+02	1.43E+02
6	5.92E+00	5.13E+02	7.37E+00	6.14E+00	6.14E+00	6.14E+00
7	5.22E-01	4.46E+02	4.59E-01	5.73E-01	5.73E-01	5.73E-01

Table 1.8. Différences dans les résultats d'itération avec une précision double et une précision quadruple pour le même nombre de threads et le même ensemble d'entrée (Villa et al. 2009)

Un autre bon exemple de l'impact de la bibliothèque mathématique sur la non-reproductibilité numérique est présenté par Glatard (Glatard et al. 2015). Glatard a analysé les performances de trois logiciels d'analyses d'image cérébrales : FSL, Freesurfer et CIVET utilisant des bibliothèques et des appels systèmes d'interception différents. Ces expériences ont été faites avec le compilateur *gcc* et des versions différentes embarquées dans l'exécutable de la bibliothèque C (*glibc)* sur des systèmes d'exploitation Linux qui contiennent une même bibliothèque mathématique *libmath*. Selon lui, il y a

beaucoup de fonctions mathématiques dans cette bibliothèque qui ne garantissent pas une reproductibilité des résultats numériques, notamment : log(), expf(), cosf(), ceilf(), floorf(), logf(), sinf(), etc. Ces fonctions utilisent la précision simple (single) de la virgule flottante. Nous avons extrait des résultats numériques de sa publication dans la Table 1.9 pour illustrer la non-reproductibilité de cette bibliothèque dans deux versions différentes du compilateur *gcc*.

FSL FAST: Brain Extraction	
glibc 2.5	**glibc 2.18**
expf(1.5405185222625732421875000000000) =4.6670093536376953125000	expf(1.5405185222625732421875000000000) =4.6670098304748535156250
FSL FIRST : Subcortical Tissue Classification	
cosf(0.523598790168762207031250000000) =0.866025447845458984375000	cosf(0.523598790168762207031250000000) =0.866025388240814208984375000
FSL : Resting-state fMRI (fixed random seeds)	
sinf(0.042260922491550445556640625000) =0.042248345911502838134765625000	sinf(0.042260922491550445556640625000) =0.042248342186212539672851562500

Table 1.9. Différences numériques entre deux versions de la bibliothèque mathématique glib utilisée par le logiciel d'imagerie FSL (Glatard et al. 2015).

Avec la bibliothèque mathématique *glibc*, on a aussi un autre exemple de Whitehead (Whitehead et Fit-Florea 2011) sur le calcul de cos(5992555.0) utilisant la même bibliothèque mathématique glibc avec la même version de gcc-4.4.3 (sur la plate-forme x86 du système d'exploitation Ubuntu 64 bits). Dans cet exemple, les résultats obtenus d'un même programme compilé en 32 bits et en 64 bits sont différents (voir Figure 1.5).

```
volatile float x = 5992555.0;
printf("cos(%f): %.10g\n", x, cos(x));

gcc test.c -lm -m64
cos(5992555.000000): 3.320904615e-07

gcc test.c -lm -m32
cos(5992555.000000): 3.320904692e-07
```

Figure 1.5. Le calcul de cosinus utilisant la bibliothèque mathématique *glibc* présente les résultats obtenus différents avec deux options de compilation –m32 et –m64 (Whitehead and Fit-Florea 2011)

Whitehead a rencontré le même problème quand il utilise deux bibliothèques mathématiques différentes sur deux architectures différentes, la plateforme x86 (glibc on Linux) et un GPU (bibliothèque mathématique de CUDA NVIDIA). Taufer (Taufer et al. 2010) avait identifié que l'origine des erreurs des opérations virgule flottante avec CUDA venait de la division et de la racine carrée dans les unités de basse précision (« *units* in the last place or unit of least precision – ULP » en anglais). En fait, comme l'implémentation de ces opérations en CUDA n'est pas conforme à la norme IEEE, alors les combinaisons de multiplication et d'addition traitées de manière non-standard conduisent aux problèmes d'arrondi et de troncature incorrecte.

…CUDA implements single-precision floating-point operations e.g., division and square root operations, in ways that are not IEEE-compliant. Their error, in ULP (Units in the Last Place) is nonzero. While addition and multiplication are IEEE-compliant, combinations of multiplication and addition are treated in a nonstandard way that leads to incorrect rouding and truncation (Taufer et al. 2010).

Taufer a donné un bon exemple des résultats différents par le niveau de précision sur GPU dans Figure 1.6 pour une simulation de dynamique moléculaire de la solution NaI incluant 988 eaus, 18 Na+ et 18l-.

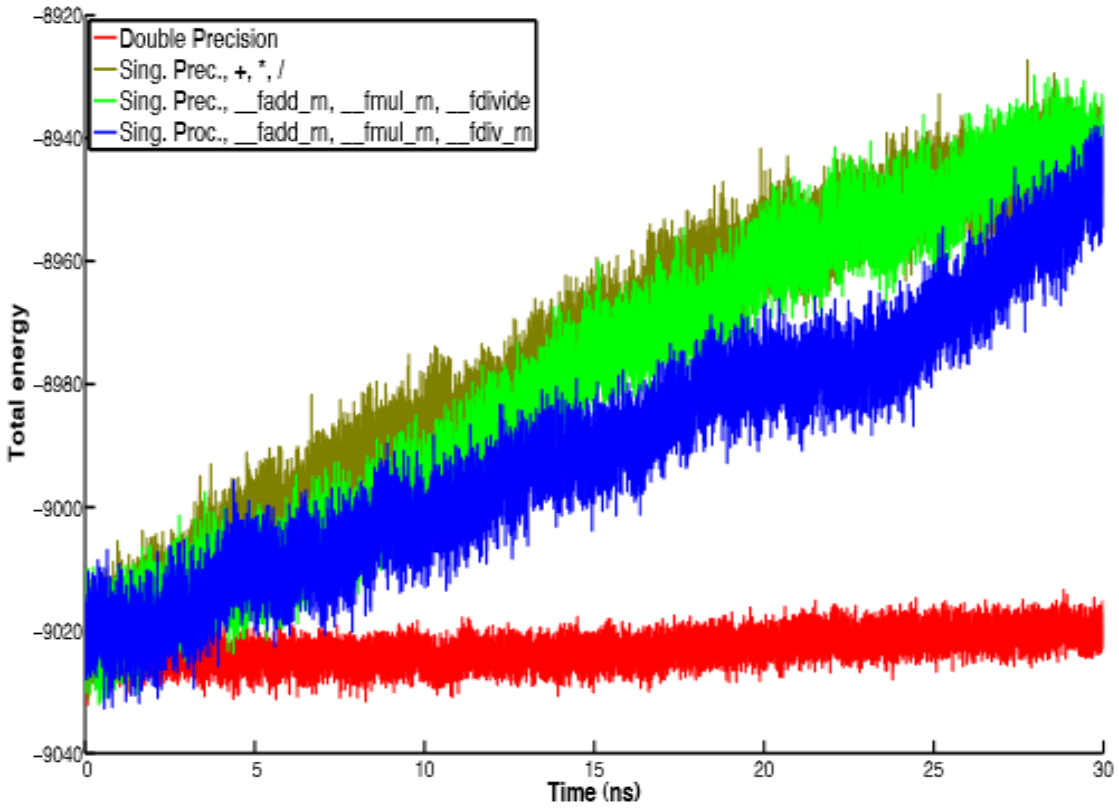


Figure 1.6. La figure compare les résultats numériques obtenus en simple précision, simple précision avec correction pour l'arrondi hors norme et la troncature, et en double précision (Taufer et al. 2010).

Pour la compilation et l'exécution de code sur le coprocesseur Intel Xeon Phi, on a des exemples intéressants dans le livre de Colfax (Colfax 2013). Avec deux options différentes de compilation : « *-fp-model fast = 1* » *et* « *-fp-model fast = 2* » les résultats obtenus sont complètement différents à partir de l'itération 2500. On a le même problème avec des options sur la précision « *-fimf-precision = low* » et « *-fimf-precision = high* » ou entre la bibliothèque normale en C et la bibliothèque de Intel MKL « -mkl » (Math Kernel Library).

Ces exemples montrent qu'il faut faire vraiment attention aux problèmes de précision et à la reproductibilité numérique des résultats calculés par la bibliothèque mathématique utilisée qui peut être de qualité moyenne.

La virgule flottante et les composants qui y sont associés affectent directement et fortement le résultat numérique. Ils sont des causes majeures de la non-reproductibilité numérique. En effet, la virgule flottante est souvent une « boite grise » pour le matériel et les langages de programmation. Si on ne s'intéresse pas au choix et à la robustesse de la bibliothèque mathématique, cela peut avoir des conséquences sérieuses.

III - Les causes de non-reproductibilité dans la simulation stochastique parallèle

La simulation stochastique est un type spécial de simulation numérique qui utilise des modèles aléatoires et souvent la méthode de Monte Carlo. Le générateur de nombres pseudo-aléatoires est un module fondamental pour la production des variables aléatoires de ce type de simulation. Normalement, ce type de simulation est facile à paralléliser sur des architectures parallèles principalement car les calculs sont très souvent indépendants.

Bien que cette technique de simulation soit très utilisée dans beaucoup de domaines de recherche, il n'est pas simple de respecter la reproductibilité numérique. Certains pensent même (à tord) que c'est normal de ne pas pouvoir obtenir les mêmes résultats puisque ces simulations sont stochastiques. Or toutes les simulations numériques utilisent des sources pseudo-aléatoires – donc déterministes – justement pour pouvoir être déboguées et reproduites. Les causes évoquées précédemment en tant que sources de non-reproductibilité impactent aussi les simulations stochastiques. Dans cette section, nous allons ajouter d'autres causes qui sont spécifiques à ce type de simulation. Celles-ci sont liées notamment à la qualité du générateur de nombres pseudo-

aléatoires utilisé, à la technique de parallélisation du ou des générateurs, éventuellement à la distribution des générateurs, à l'initialisation des générateurs, à la conception de la simulation stochastique, etc.

Avant de commencer, rappelons un message d'avertissement de Hellekalek (Hellekalek 1998) citant un physicien expérimenté. Cette remarque est vraiment importante pour comprendre pourquoi la corrélation des nombres pseudo-aléatoires influence les résultats du calcul d'applications de type Monte Carlo. En d'autres mots, elle souligne la relation entre la corrélation dans les nombres pseudo-aléatoires (expliqué plus loin dans la Figure 1.11) et la pertinence des résultats obtenus.

Monte Carlo results are misleading when correlations hidden in the random numbers and in the simulated system interfere constructively (Hellekalek 1998)

La corrélation et la reproductibilité sont deux critères différents pour un bon générateur que nous allons présenter en détail dans le chapitre 2. L'absence de corrélations dans les flux de nombres pseudo-aléatoires ne conduit pas directement à la reproductibilité numérique d'une simulation stochastique mais elle est essentielle à la qualité des résultats obtenus. S'il y a corrélation dans les flux de nombres pseudo-aléatoires, les résultats d'une simulation stochastique sont de qualité très basse, très loin de la prédication théorique et donc non crédibles ou utilisables. La thèse de Romain Reuillon donne de nombreux exemples, notamment dans le domaine de la simulation pour la médecine nucléaire (Reuillon 2008).

On peut voir les conséquences importantes de mauvais résultats dans les exemples tirés de Einarsson et de Bailey (cf. II.3.b). ***La reproductibilité numérique d'un résultat mauvais ou erroné n'apporte rien.*** *Dans ce cas-là, la reproductibilité est non-nécessaire et inutile*. Il existe donc une relation indirecte importante entre la corrélation des flux de nombres pseudo-aléatoires et la reproductibilité numérique. ***Même si on n'admet pas cette relation ou que l'on oublie la corrélation dans le flux de nombres pseudo-aléatoires, on ne peut pas garantir la reproductibilité numérique dans la simulation stochastique parallèle***. En fait, le titre de l'article de Hellekalek (Hellekalek 1998) a tout dit : « ***Don't trust parallel Monte Carlo*** ». Nous allons analyser le problème dans la suite de cette section.

III.1 - Des mauvais générateurs pseudo-aléatoires dans de bons logiciels

Tout d'abord, nous pouvons aujourd'hui classer un générateur comme « mauvais » s'il ne passe pas les tests empiriques et statistiques de la batterie de tests TestU01 développée par L'Ecuyer et Simard (L'Ecuyer et Simard 2007). Il peut également être qualifié de mauvais s'il n'est pas reproductible en séquentiel selon la méthode que nous avons proposée dans (Dao et al. 2014). Des informations plus détaillées vous seront présentées dans les chapitres traitant des propositions et des résultats obtenus.

Des mauvais générateurs pseudo-aléatoires, qui ne sont pas aptes au calcul scientifique, peuvent être présents dans le système d'exploitation, dans des logiciels ou des langages de programmation. On peut en citer quelques-uns : les générateurs rand() et srand() dans la langage C, random() dans Java et Python, rand() dans Perl et Matlab, etc. Ils existent depuis longtemps comme module par défaut et sont faciles à utiliser. Nous les considérons mauvais car leur qualité n'est pas suffisante pour les applications scientifiques des simulations stochastiques.

Dans l'histoire pas si ancienne de l'informatique Deley (Deley 1991) a présenté les problèmes d'aspects non-aléatoires dans le générateur rand() qui présentait historiquement une période ridiculement petite (taille maximale de $2^{15} - 1$). Il s'agissait du générateur présenté dans la norme ANSI de langage C/C++. Selon lui, on peut probablement obtenir seulement environ 20000 nombres aléatoires puis, la séquence de nombres générés commence à perdre son caractère aléatoire. Ce problème a aussi été évoqué par Srinivasan (Srinivasan et al. 2003) pour plusieurs générateurs de nombres pseudo-aléatoires qui montrent des comportements corrélés bien avant la fin de la période.

> *Each PRNG has a finite number of possible states, and hence the ''random'' sequence will start repeating after a certain ''period,'' leading to non-randomness. Typically, sequences stop behaving like a truly random sequence much before the period is exhausted, since there can be correlations between different parts of the sequence (Srinivasan et al. 2003).*

Reuillon (Reuillon 2008) a clairement présenté une limitation de la fonction rand() de Linux (déjà bien plus évoluée que celle de la norme ANSI) dans une comparaison entre rand() et l'algorithme Mersenne Twister 19937 (Matsumoto et al. 1998) dans le contexte de la simulation d'un mouvement brownien. La figure 1.7 présente les résultats obtenus après 10.000 réplications, à gauche en

utilisant la méthode rand(), à droite en utilisant le générateur Mersenne Twister. Nous constatons que les résultats obtenus avec le générateur rand() de Linux sont nettement moins réguliers.

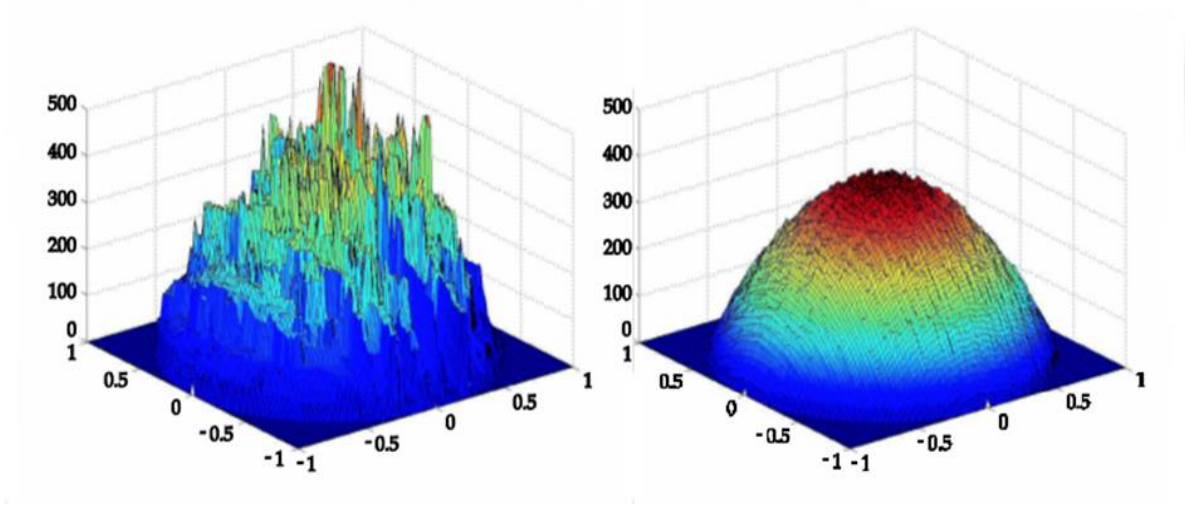


Figure 1.7. Résultats d'un calcul d'Equations aux Dérivées Partielles probabiliste après 10000 réplications, à gauche en utilisant la méthode rand() et à droite en utilisant l'algorithme Mersenne Twister 19937. (Reuillon 2008)

Jones (Jones 2010) a également présenté une liste de mauvais générateurs présents dans les systèmes ou dans les langages de programmation C (avec rand(), random(), drand48()), Java (avec la classe Random), Perl et Matlab (avec le générateur rand), Mathematica (avec SWB) et R (avec ran0(), ran1()). Comme bien d'autres chercheurs, il a donné la recommandation de « ne pas utiliser les générateurs par défaut des systèmes ou langages » pour des applications scientifiques.

The easiest (laziest) option is to make use the standard library « rand » function e.g. those found in the standard C library or the standard Perl rand for example. Almost all of these generators are badly flawed. Even when they are not, there is no guarantee that they were not flawed in earlier releases of the library (e.g. see note on Python below) or will not be flawed in future release.
Note that all of these standard generators have been shown to have serious defects:
Standard Perl rand
Python random() (versions before V2.3; V2.3 and above are OK)
Java.util.Random
C - library rand(), random() and drand48()
Matlab's rand
Mathematica's SWB generator
ran0() and ran1() in the original Numerical Recipes book. (Jones 2010)

Une dizaine d'année plus tôt, Coddington et Ko (Coddington and Ko 1998) avaient déjà tiré une leçon précieuse du test des générateurs CMF_RANDOM, CMSSL FAST_RNG et CMSSL VP_RNG sur la Connection Machine, fournis par le vendeur d'ordinateur parallèle, et aussi des générateurs P_RANDOM #1, P_RANDOM #2 et PRAND qui n'avaient pas passé leurs tests.

One lesson from these results is not trust random number generators provided by computer vendors (Coddington and Ko 1998).

Il y a d'autres générateurs de qualité inférieure que Reuillon (Reuillon 2008) a détaillé dans sa thèse : TRandom avec le germe 2^{28} dans le logiciel ROOT ainsi que les générateurs de la bibliothèque CLHEP pour la physique des hautes énergies déjà pointés du doigt par Heinrich (Heinrich 2004) (*random, lrand48, randu, rndm, ranmar, ranlux, jamesRandom, ranecu, ranarray,…).* Des projets de plate-forme logicielle tels que le projet GATE et la boîte à outils Geant4 s'appuient sur ces outils (ROOT, CLHEP) et leurs bibliothèques. Suite à ces travaux le générateur par défaut du logiciel GATE a été changé pour un MersenneTwister. Bien des scientifiques ne sont souvent pas conscients des bonnes pratiques liées à l'usage de ces générateurs et encore moins aux besoins de reproductibilité. Ensuite, Reuillon a également présenté le générateur *ran0* aussi appelé « minimal standard » dans l'OMNet++ avec sa version 2.2p3 de l'étude de Hechenleitner (Hechenleitner et al. 2003) et les anciens générateurs dans le simulateur de réseau NS-2 des versions avant 2.1b9 de l'étude de Umlauft (Umlauft et al. 2007).

Le problème des générateurs de nombres pseudoaléatoires défectueux a aussi été soulevé par Matsumoto (Matsumoto et al. 2006). Certains générateurs récents restent toujours défectueux. Matsumoto donne l'exemple de deux structures de générateurs classiques de son point de vue : LCG (Linear Congruential Generator) et le générateur GFSR (Generalized Feedback Shift Register). Ils sont encore largement utilisés alors qu'ils donnent des résultats erronés. La Figure 1.8 présente la distribution spatiale dans un cube (test spectral) des nombres aléatoires générés par LCG et la compare à ceux générés avec Mersenne Twister. La structure en treillis des nombres générés par LCG conduit à des résultats mauvais ou erronés. Ces générateurs influencent donc la qualité du résultat obtenu, en particulier dans les simulations à grande échelle.

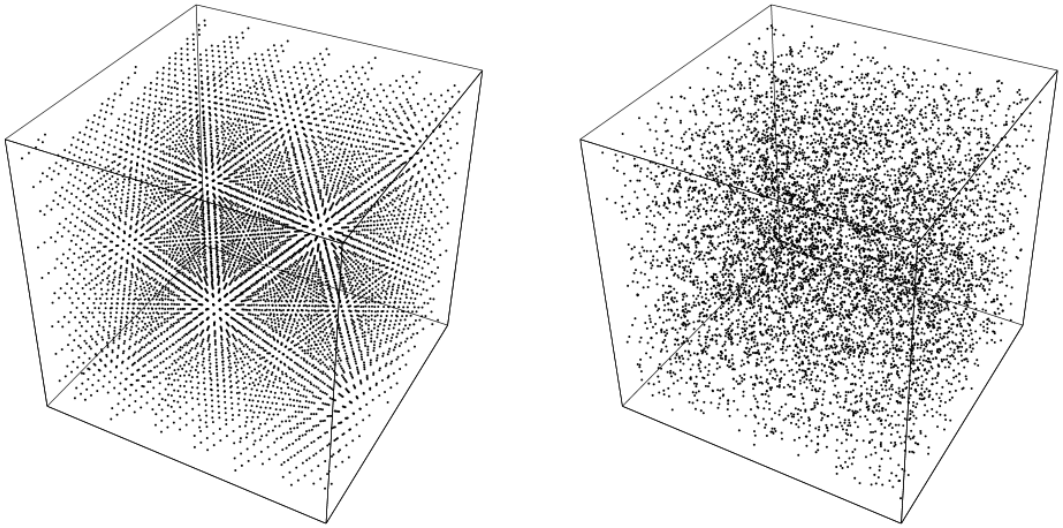


Figure 1.8. Les points pseudo-aléatoires dans le cube de gauche ont été générés par un générateur de type congruentiel linéaire, à droite par Mersenne Twister (Matsumoto et al. 2006).

Des générateurs basiques peuvent être bons pour des jeux ou pour de petites applications où la quantité de nombres pseudo-aléatoires à générer ne dépasse pas leur période de corrélation et que leur utilisation dans l'espace n'est pas requise. Cependant, ils ne sont jamais un bon choix pour des applications scientifiques et encore moins pour du calcul parallèle à hautes performances. Plusieurs références dans la littérature ont montré que l'utilisation de ces générateurs entache la qualité des résultats scientifiques.

III.2 - Des mauvaises techniques d'initialisation des générateurs

On sait qu'un générateur commence son flux par un état initial (une simple valeur initiale germe ou « seed » en anglais pour les anciens générateurs précisément parmi les plus mauvais). Le choix de cet état est très important pour générer un bon flux de nombres pseudo-aléatoires. Cependant, Kim et Yang (Kim et Yang 2006) ont fait un commentaire important : « le choix du vecteur d'initialisation ne peut pas être déterminé par une base théorique ». C'est-à-dire, il n'existe pas de façon spécifique et/ou mathématique basée sur la théorie permettant de choisir un bon état (vecteur d'initialisation dans le cas de la publication de Kim et Kang) pour un générateur quelconque. Dans la réalité, on utilise les tests par force brute et la méthode empirique pour choisir un bon état d'initialisation.

The choice of the initial seed vectors in random number generator could not be determined by the theoretical basis. The recommendation to select initial values at random is doubtful. In general, the initial seed vectors could be chosen by empirical methods. (Kim and Kang 2006).

Dans le cas des anciens (et mauvais générateurs), la méthode d'initialisation la plus simple utilisait une des fonctions de temps du système telles que *time()* ou d'identification du processus *getpid()* pour créer une valeur initiale. Jones (Jones 2010) précise que c'était une mauvaise méthode dans le cas où plusieurs tâches étaient soumises successivement. D'autre part, si ce type de méthode peut convenir pour des jeux, elle exclue totalement la reproductibilité des résultats. L'emploi de ce type d'approche pour du calcul scientifique frise l'incompétence.

The simplest way to seed a RNG is to take something like the current time e.g. using the time() function found in Unix and most C libraries…What's wrong with this? Well, it's more or less acceptable if you are running a program once a day – but think of what happens when you submit 1000 jobs to a Beowulf cluster in one batch. Hundreds of jobs on different nodes will be starting at almost exactly the same time – therefore many of your jobs will be starting with exactly the same seed and therefore those that have the same seed will generate exactly the same results (assuming your code has no bugs in it). Not only is it a waste of CPU time to repeat the exact same calculation many times, but if you don't know it has happened, any statistics you carry out on the final data will be biased by the repetitions. (Jones 2010).

Katzgraber (Katzgraber 2010) a souligné l'importance du choix de la valeur initiale pour un générateur de nombres pseudo-aléatoires. Ce choix influencera la qualité des flux, la corrélation entre les flux et donc la reproductibilité des résultats calculés.

The seed determines the sequence of random numbers. Therefore, it is crucial to seed the PRNG carefully. For example, if the period of the PRNG is rather short, repeated seeding might produce overlapping streams of random numbers. Furthermore, there are generators where a poor choice of the seed might produce correlated random number. (Katzgraber 2010).

Matsumoto a également indiqué que la majorité des générateurs de nombres pseudo-aléatoires modernes ont des défauts communs dans l'initialisation. Des états corrélés conduisent à des séquences de nombres aléatoires corrélées. Ce défaut se traduit par des symptômes communs comme la dépendance linéaire, illustrée sur la Figure 1.9 et les collisions de différence (« difference collision » en anglais, on dit qu'une collision s'est produite à chaque fois que l'on observe des répétitions (Srinivasan et al. 2003)), illustrées dans la Table 1.10 (Matsumoto et al. 2007).

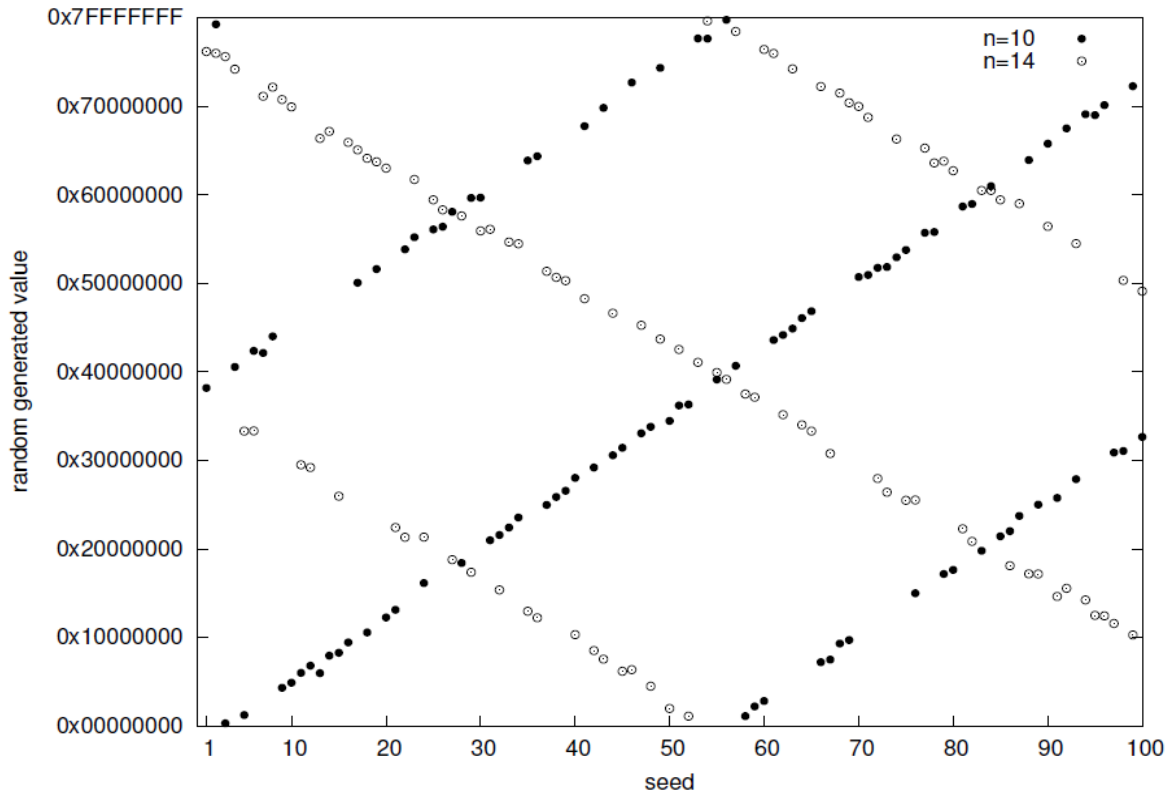


Figure 1.9. Les valeurs de sortie au 10ème et 14ème tirage du générateur random **de la GNU Scientific Library avec le le langage C FreeBSD et les germes s = 1 ; 2 ; … ; 100. (Matsumoto et al. 2007).**

La Figure 1.9 présente la dépendance linéaire sur la valeur initiale du générateur *random* de la bibliothèque scientifique GNU avec le langage de programmation C de FreeBSD. Normalement, un générateur de nombres pseudo-aléatoires va générer des valeurs différentes quand on utilise des valeurs initiales différentes. Mais ici, ces valeurs de sortie ont une relation sur les bits les plus significatifs. Selon cette figure, on voit visuellement la relation des bits après 10 et 14 tirages du générateur random(). Clairement, la dépendance sur la valeur initiale est observée seulement à cause d'une corrélation sur un bit. Matsumoto a aussi présenté ce même problème avec le générateur *ranlux* pour les vingtième, vingt cinquième et quarante et unième tirages (correspondant à n = 20, 25, 41) mais aussi avec drand48, et rand().

n	s=1	s=6	s=3	s=10
$x_0(s+1) - x_0(s)$	0x0ef1bc75	0x0ef1bc75	0x5e240b4b	0x5e240b4b
$x_1(s+1) - x_1(s)$	0x1e145efa	0x1e145efa	0x095a16e1	0x095a16e1
$x_2(s+1) - x_2(s)$	0x569876c8	0x569876c8	0x0f3e4c65	0x0f3e4c65
$x_3(s+1) - x_3(s)$	0x1f7dfaba	0x1f7dfaba	0x7d41feab	0x7d41feab
$x_4(s+1) - x_4(s)$	0x129d6b03	0x129d6b03	0x06ec1ff8	0x06ec1ff8
$x_5(s+1) - x_5(s)$	0x7c0e50d1	0x7c0e50d1	0x1a7c458d	0x1a7c458d
$x_{10}(s+1) - x_{10}(s)$	0x6dffd0a3	0x6dffd0a3	0x53d428ef	0x53d428ef
$x_{100}(s+1) - x_{100}(s)$	0x56f28400	0x56f28400	0x1628f237	0x1628f237
$x_{1000}(s+1) - x_{1000}(s)$	0x27a58ffe	0x27a58ffe	0x71cf9f1f	0x71cf9f1f

Table 1.10. La collision différente du générateur *mrg* avec le modulo de 2^{31}–1 (Matsumoto et al. 2007).

La Table 1.10 illustre un autre problème lié au choix des valeurs initiales. C'est une collision au sein du générateur *mrg* dans la bibliothèque scientifique GNU avec le modulo de $2^{31} - 1$ pour deux valeurs initiales différentes. Dans ce cas-là, on a deux collisions aux seeds s = 1 et s = 6 à gauche, s = 3 et s = 10 à droite. Ce problème est également au générateur *cmrg* sauf pour quelques irrégularités.

De plus, Matsumoto a fait deux tests sur la dépendance affine et sur les collisions pour 58 générateurs proposés dans la bibliothèque scientifique GNU. Seuls 13 générateurs ont passé ces tests, mais plusieurs des 13 restants sont défectueux sur d'autres aspects.

Among 58 available generators in the GNU Scientific Library, 45 of them show such defects. These are not because of the recursion, but because of carelessly chosen initializing schemes in the implementations…
Other generators do not have affine dependency, because of their non-linearity introduced in the recurrence or in the initialization. Note that some of these 13 are known to be defective for other reasons. (Matsumoto et al. 2007)

Même si l'on a choisi un bon générateur comme MRG32k3a ou Mersenne Twister 19937, il faut choisir un état initial de qualité sous peine d'obtenir de mauvais flux de nombres pseudo-aléatoires et de fait des risques d'avoir des résultats entachés par ce type d'erreur. Nous avons une illustration très claire de ce problème à travers l'exemple intéressant du calcul de π présenté dans (Reuillon 2008) illustré sur la Figure 1.10.

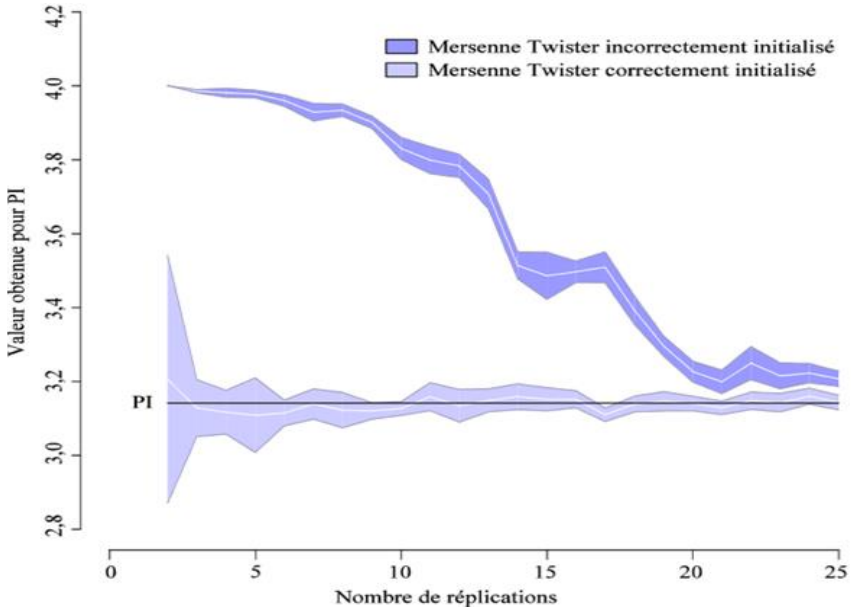


Figure 1.10. Influence d'une mauvaise initialisation du générateur Mersenne Twister 19937 (Reuillon 2008).

Cette expérience montre l'influence d'une mauvaise initialisation du générateur Mersenne Twister 19937. On peut voir que l'intervalle de confiance autour de la valeur π calculée avec une mauvaise initialisation de Mersenne Twister ne contient toujours pas la vraie valeur de π , même après 25 réplication (même si on observe une convergence).

Coddington et Ko (Coddington and Ko, 1998) ont aussi souligné que l'initialisation joue un rôle important dans la qualité des flux de nombres pseudo-aléatoires et cela impact les résultats des calculs parallèles.

The quality of a parallel random number generator can be heavily dependent on the initialization of the generator (Coddington et Ko 1998).

Le choix de l'état initial d'un générateur joue un rôle vraiment important qui mérite une attention même si on utilise un bon générateur. Un mauvais choix conduit à des problèmes de collision, de corrélation de flux, de dépendance entre bits. Cala peut aussi être une cause de non-reproductibilité numérique des résultats pour les simulations stochastiques. La qualité des résultats dépend également d'une initialisation correcte du générateur de nombres pseudo-aléatoires utilisé. Il n'existe cependant pas de façon théorique pour choisir une initialisation correcte, en fonction du générateur choisi et de sa structure, il convient de connaître la façon précise la façon de bien l'initialiser.

III.3 - De mauvaises techniques de parallélisation et de distribution des générateurs

La simulation stochastique est facile à paralléliser. Il convient cependant de s'assurer que la méthode de parallélisation des flux de nombres pseudo-aléatoires employée est correcte. Ces techniques existent, elles restent méconnues, de plus, certaines techniques de parallélisation et de distribution des flux de générateurs sont incompatibles avec une reproductibilité numérique des simulations stochastiques.

Dès 1998, Hellekalek (Hellekalek 1998) a indiqué l'inconvénient de la technique de mise en parallèle avec des flux multiples, dans laquelle chaque processeur a son propre générateur. Cette technique peut conduire au problème de corrélation entre flux ou à des collisions dans le processus de mise en œuvre des flux générés.

Method I assigns different RNGs to different processors….The danger with the first method is the following. There might be unknown correlations between the different RNGs we use. (Hellekalek 1998).

Hellekalek (Hellekalek 1998) et L'Ecuyer (L'Ecuyer et al. 2015) ont souligné les difficultés pour trouver de bons paramètres à la volée. Les pré-calculer et les stocker est une solution mais avec un surcoût.

Method I: … If we happen to use the same type of RNG but with different parameters, then we might encounter very unpleasant surprises. (Hellekalek 1998).
A different generator for each stream: This usually means using the same type of RNG, but with different parameters…Finding many different good parameter sets is usually feasible, but this approach is generally not very convenient to manage, because finding those good parameters on fly or precomputing and storing them adds overhead either way, in time or space. Moreover, different parameters give different streams but it does not necessarily imply that the streams can be considered as independent…Some have suggested to choose parameters at random among those that give a maximal period, but this is dangerous, because maximal period does not suffice…(L'Ecuyer et al. 2015)

Hill (Hill et al. 2013), Brugger (Brugger et al. 2014) et L'Ecuyer (L'Ecuyer et al. 2015) ont également fait une analyse de la technique de parallélisation par le choix des initialisations différentes pour un même type de générateur. Avec cette technique, les flux générés peuvent être dépendants, comme l'illustre la Figure 1.11 tirée de Brugger, même si la probabilité reste faible.

Method I: … If we happen to use the same type of RNG but with different parameters, then we might encounter very unpleasant surprises. (Hellekalek 1998).
Random spacing: The random spacing or indexed sequences method builds a partition of n streams by initializing the same generator with n random status…The risk is of course to have a bad initialization linked to the fact that two random statuses could be too close to each other, implying an overlapping of corresponding sequences. (Hill et al. 2013)
A single PRNG with a "random" seed for each stream: There is a possibility of overlap, but the probability is often negligible (L'Ecuyer et al. 2015).

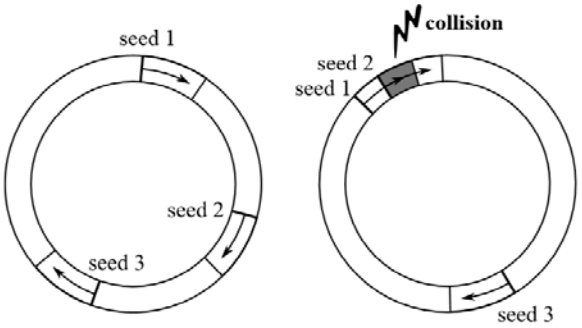


Figure 1.11. Initialisations sans (à gauche) et avec chevauchement (à droite) d'un générateur (Brugger et al. 2014)

Sur l'illustration à gauche de la figure 1.11, on a 3 initialisations avec 3 flux générés correspondants et ces 3 flux ne sont pas corrélés. Par contre, a droite on a une corrélation des flux générés pour les choix de graines initiales 1 et 2 dans l'illustration de droite.

Srinivasan (Srinivasan et al. 1998) et Hill (Hill et al. 2013) ont aussi présenté une faiblesse importante de l'utilisation d'un serveur central. Cette technique ne sera pas reproductible avec un nombre de processeurs différents où lorsque les nombres pseudo-aléatoires sont appelés dans un ordre différent par les différentes étapes du programme. Elle créera aussi un engorgement des flux si l'on a trop d'éléments de traitement (PE) qui sollicitent le serveur.

Central server: … It also hinders reproducibility because the different processes may request random numbers in different orders in different runs of the program, depending on the network traffic and implementation of the communication software (Srinivasan et al. 1998)
Central server technique. This first approach is not truly parallel…First, a simulation using this technique will not be reproducible because of scheduling polices that might change the order in which numbers will be provided to PEs…Second, the central server approach will create a bottleneck if too many PEs are considered. (Hill et al. 2013)

La technique du serveur central permet de créer facilement de multiple flux de nombres pseudo-aléatoires. Pourtant, c'est une mauvaise technique de parallélisation pour une simulation stochastique parallèle parce qu'elle génèrera des codes non-reproductibles numériquement.

En ce qui concerne les autres techniques de sous-flux, les auteurs ont également identifié des problèmes de corrélation et expliquent comment les éviter. Hill (Hill et al. 2013) a fait un commentaire important pour préserver la reproductibilité entre deux exécutions de la même

simulation : « il ne faut pas utiliser la technique du saut de grenouille lorsque le grain de parallélisme n'est pas encore fixé ».

Cycle division (leap frog and sequence splitting):… If the number of random numbers consumed is greater than expected, then the sequences on different process could overlap. (Srinivasan et al. 1998)
Leap frog: … This technique needs to be used with caution depending on the environment it is set up in. Thus, to preserve reproducibility between two executions of the same simulation, one should not use this technique when the parallelism grain is not fixed yet. As a matter of fact, different random streams will be assigned to PEs depending on the chosen grain. (Hill et al. 2013)
Sequence splitting:… If overlapping can be easily avoided, long-range correlations in the initial RNG can lead to small-range correlations between the potential substreams. (Hill et al. 2013)
Splitting a single RNG into equally-spaced blocks (streams)…One potential with splitting is the dependence between the different streams. (L'Ecuyer et al. 2015)

Une bonne illustration de la corrélation interne des flux en parallèle est présentée sur la figure 1.12, tirée de Reuillon (Reuillon et al. 2008). Elle illustre un exemple de l'effet des corrélations entre séquences sur les résultats de simulation dans le contexte de la reconstruction d'une image tomographique.

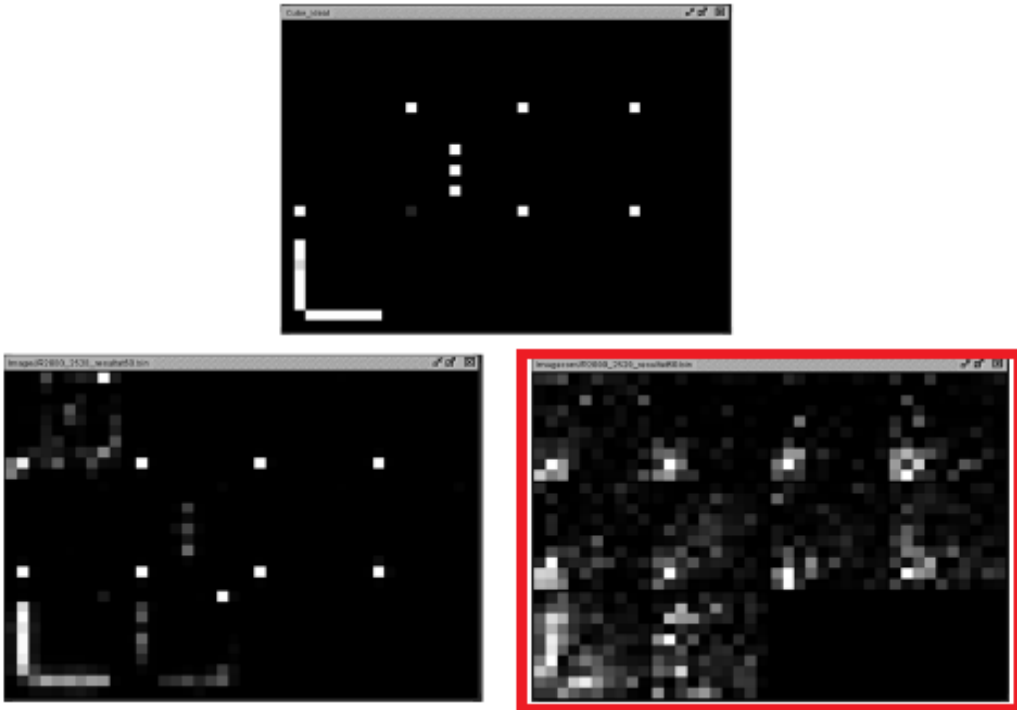


Figure 1.12. L'effet des corrélations entre séquences de nombres pseudo-aléatoires utilisées en parallèle sur les résultats d'une simulation Monte Carlo distribuée. L'image en haut est le résultat parfait idéal. Celle en bas à gauche est un résultat très correct obtenu en utilisant une simulation stochastique parallèle. L'image en bas à droite est un résultat obtenu en utilisant une simulation qui consomme des séquences avec des chevauchements importants (Reuillon et al. 2008).

Dans cette exemple, l'image en bas à droite (nous l'avons entouré par une bordure (rouge) pour bien présenter le problème de corrélation inter-séquence) a été obtenue en utilisant le même algorithme de reconstruction, en utilisant de très bonnes séquences de nombres pseudo-aléatoires (avec très peu de corrélations inter-séquences) parallèles, mais les séquences ont été intentionnellement créées pour se chevaucher.

Clairement, les résultats de l'image en bas à droite sont faussés. Si cette image est ici le produit d'une construction volontaire, il faut savoir que ce type de corrélation a d'abord été obtenu de façon involontaire entre 2003 et 2005 dans le cadre des travaux de la thèse de Ziad El Bitar (El Bitar 2006) (Reuillon 2008). Comme il s'agissait d'une simulation dans le domaine médical, il était important de creuser cette question. Ceci a abouti à mieux comprendre les causes et l'impact des corrélations entre flux stochastiques pour des simulations parallèles.

Plus récemment, Taufer (Taufer et al. 2015) a aussi montré l'impact de l'assignation de calculs en virgule flottante à des threads dans un ordre d'exécution différent. Les problèmes sont liés aux erreurs d'arrondis et au manque d'associativité du calcul en virgule flottante. Le nombre de threads, la quantité de nombres pseudo-aléatoires par flux et le nombre de calculs par flux affectent aussi la précision des résultats obtenus. De plus c'est principalement la phase de réduction (calcul des moyennes et des intervalles de confiance) qui est impacté par l'ordre d'exécution des threads. Cette source de non-reproductibilité numérique impacte les comparaisons entre exécutions séquentielles et parallèles ou entre exécutions parallèles.

A partir des analyses et des exemples ci-dessus, on voit que des mauvaises techniques de parallélisation créent de la corrélation entre les flux générés et la distribution des flux aux processeurs ne garantit pas la reproductibilité numérique. Ainsi, la parallélisation d'une simulation stochastique ne sera pas forcément satisfaisante parce que les résultats obtenus peuvent être faussés. Le choix des techniques de parallélisation et de distribution décidera de la qualité et de la reproductibilité numérique d'une simulation stochastique parallèle.

III.4 - Des mauvaises conceptions du logiciel stochastique séquentiel empêchant la reproductibilité sur des implémentations parallèles.

Beaucoup d'applications scientifiques ont été développées depuis longtemps et améliorées pour être déployées sur des systèmes de calcul à hautes performances. Cependant, cette mise à jour se

heurte au problème de la reproductibilité numérique à cause des différences de conception entre la version séquentielle et la version parallèle, à cause de l'obsolescence de la version séquentielle, ou potentiellement d'erreurs dans la conception et l'implémentation parallèle.

Dans le cas d'un modèle simple comme celui d'Ising en 2D et 3D, Diaconu (Diaconu et al.2003) a souligné l'importance de veiller à la reproductibilité numérique des résultats et à l'indépendance des calculs sur des processeurs différents.

Most serial Monte Carlo codes are readly adaptable to a parallel environment. Strengths are still for multi-dimensional problems and complex geometries. Care must be taken to assure reproducibility results and assure that the calculations on different processors are independent. (Diaconu et al. 2003)

Gropp (Gropp 2005) a bien présenté les obstacles à la reproductibilité numérique dans le codage d'un programme en parallèle : la compétition entre les tâches, l'ordre d'exécution, le message en mémoire tampon, le non blocage et les opérations asynchrones, etc. Ces causes sont également introduites par Revol (Revol et al. 2014).

Races: *One of the most dangerous and common errors in shared-memory parallel programs is a race condition. This occurs in parts of the code where a race between two or more threads of execution determines the behavior of the program. If the "wrong" thread wins the race, then the program behaves erroneously.*

Out-of-order execution: *In writing algorithms with the shared-memory model, one often has to ensure that data on one thread is not accessed by any other thread until some condition is satisfied. For this purpose locks are often used…Unfortunately, locks are often quite expensive to execute. Programmers are often tempted to use a simple flag variable to mediate access to the data…However, this code may not execute correctly. The reason is there is no guarantee that either compiler or the processor will preserve the order of the operations to what appears to be (within a single thread) independent statements.*

Message buffering: *Many message-passing programs are immune to race conditions because message passing combines data transfer and notification into a single routing and because there is no direct access to the memory of another process. However, message-passing programs are susceptible to resource limits.*

Nonblocking and asynchronous operations: *A common technique for working around high-latency operation is to split the operation into separate initiation and completion operations. This is commonly used with I/O operations; many programmers have used nonblocking I/O calls to hide the relatively slow performance of file read and write operations. The same approach can be used to hide the latency of communication between processes.*
Writing correct parallel programs is also difficult. (Gropp 2005)

En effet, ces obstacles dans la conception d'un programme en parallèle sont aussi liés aux problèmes du matériel que nous présenterons plus en détail dans la section III.5 de ce chapitre. Villa (Villa et al. 2009) a mentionné l'intérêt de l'option du compilateur (***#pragma mta block schedule***) dans une boucle parallèle sur le système Cray XMT pour changer l'ordre d'exécution des threads.

We tightened our PSE code to use statically scheduled parallel loops on the XMT using the compiler directive (**#pragma mta block schedule**), so that all the effects of dynamically scheduling loop iterations onto threads are suppressed. This guarantees deterministic assignment of iterations to threads. We performed this modification on each accumulation or reduction loop in the code. **This unfortunately does not guarantee determinate results** since the XMT compiler automatically recognizes the reductions and generates code that performs accumulations using atomically updated shared variables. In this case, the order in which different threads perform the atomic updates is influenced by run-time and OS thread scheduling policies (Villa et al. 2009).

Un autre problème peut venir d'une version trop ancienne du code séquentiel. Kroese (Kroese et al. 2014) a indiqué que la plupart des techniques de Monte Carlo ont évolué directement à partir des méthodes développées dans les premières années de l'informatique. Ces méthodes ont été conçues pour des machines avec un seul processeur (puissant pour l'époque). Le calcul à haute performance moderne, cependant, tend de plus en plus vers l'utilisation de nombreux processeurs fonctionnant en parallèle. Alors que de nombreux algorithmes de Monte Carlo sont intrinsèquement parallélisables, certains ne peuvent pas être facilement adaptés à ce nouveau paradigme d'exécution informatique.

Most Monte Carlo techniques have evolved directly from methods developed in the early years of computing. These methods were designed for machines with a single (and at that time, powerful) processor. Modern high performance computing, however, is increasingly shifting towards the use of many processors running in parallel. While many Monte Carlo algorithms are inherently parallelizabe, others cannot be easily adapted to this new computing paradigm. (Kroese et al. 2014)

Dans une présentation sur la reproductibilité numérique d'une application scientifique climatique sur un système de calcul à haute performance avec des processeurs Intel Xeon Phi, Thomas (Thomas 2012) indique que la reproductibilité numérique du code en parallèle est influencée par : l'ordre des calculs, les opérations de réduction avec les instruction DDOT, DGEMM), l'ensemble des instructions du modèle SIMD (SSE, AVX, VSX), l'alignement de mémoire non-déterministe, les fonctions de mémoire (malloc, allocate), l'alignement des segment de mémoire et de pile, l'utilisation des bibliothèques MKL et ESSL directement dans le code ou dans un code utilisé, le processus de compilation, etc. Nous allons voir ces points plus en détail dans la section III.5 de ce chapitre.

**It takes a combination of:*
- Code sensitive to the order of computations
- "reduction" operations (DDOT, DGEMM)
- A SIMD instruction set (SSE, AVX, VSX)
- Non deterministic memory alignment
- In your code or in someone else's (MKL, ESSL)
**malloc()/allocate() do not always return 16 bytes (SSE/VSX) or 32 (AVX) aligned data*
**Heap and stack alignment can vary due to*
- Varying run time coditions (date, directory, pid,…)
- ASLR (Address Space Layout Randomization)
- Check /proc/sys/kernel/randomize_va_space
**The compiler will process loops with a prologue (scalar) up to the first aligned index, the loop body (SIMD) and an epilogue (scalar). (Thomas 2012)*

III.5 - Les problèmes liés aux architectures matérielles pour le calcul parallèle

Les causes de non-reproductibilité abordées dans cette partie concernent l'architecture matérielle des ordinateurs.

Gropp (Gropp 2005) signale le risque d'erreurs sur les réseaux interconnectés pour les simulations de longue durée et la difficulté créée par les systèmes parallèles hétérogènes.

Hardware errors: Because parallel computers are often used for the most challenging problems, another source of problems is the low, but not zero, probability of an error in the computer

hardware. This has sometimes led to problems with long-running simulations, particularly with high-performance interconnect networks.
Heterogeneous parallel systems: is one in which the processing elements are not all the same. They may have different data representations or ranges… On a heterogeneous system, such an approach may not be valid, depending on how the results are used. Additional issues concern the different handling of the less specified parts of the IEEE 754 floating-point specification… (Gropp 2005)

Il signale notamment les problèmes liés au calcul en virgule flottante déjà évoqués dans la section III.4 de ce chapitre.

Taufer (Taufer 2012) a également expliqué que la reproductibilité numérique et la stabilité des résultats ne peuvent pas être assurées sur une grande plateforme multi-cœurs, à cause de la limitation dans la représentation de l'arithmétique virgule flottante. Les threads sont planifiés pour être exécutés de façon aléatoire sur les cœurs et les scientifiques sont confrontés aux impacts de cumul des erreurs d'arrondi.

There is no doubt that exascale architectures will be multi-core platforms, with the number of cores expected to increase substantially. The increased number of cores brings a radical change to what scientists can study on exascale systems; it also exposes and emphasizes a well-known and so far often ignored problem: large-scale, massively parallel simulations lacking numerical reproducibility and stability. In other words, the numerical reproducibility and stability of simulation results cannot be assured on large multi-core platforms because computers limit the representation of floating-point arithmetic; threads are scheduled randomly among cores; and scientists fail to consider mathematical properties in their applications (e.g., the monotonicity of rounding errors) (Taufer 2012).

Récemment, Hill (Hill et al. 2015) a présenté les limitations du matériel dans les systèmes de calcul à hautes performances et son influence sur la reproductibilité numérique. Il a particulièrement mis en avant le problème des exécutions d'instructions dans le désordre (« out-of-order ») sur les nouveaux processeurs multi-cœurs et les many-coeurs. Il expose aussi le problème de la réduction du temps moyen de bon fonctionnement sans impact des « soft errors » (interaction des composants électroniques avec des particules alpha par exemple). Ce temps est particulièrement réduit sur les supercalculateurs (moins de 24 heures de bon fonctionnement sur la majorité des systèmes leaders du Top 500).

The introduction of new processors with the multi and manycores twelve years ago impacted the obtaining of correct results because of the out-of-order execution of floating point instructions.
The last point reducing our chance to obtain reproducibile results on first class computing machines is linked to silent data corruption and soft errors such as rare interactions of circuitry with stray subatomic particles. If those events are extremely rare for desktop and cluster computing, it is a fact that with our current technology, the Mean Time Between Failure (MTBF) is now measured in hours on the largest machines (Hill et al. 2015).

Gropp, Taufer, Thomas, Revol et Hill ont bien présenté les problèmes liés aux nouvelles architectures matérielles pour le calcul parallèle, elles influencent également à la non-reproductibilité numérique pour la simulation stochastique parallèle. Ces problèmes sont complexes. Comme Goldberg l'a remarqué : les concepteurs de systèmes informatiques possèdent rarement une formation solide en analyse numérique, ces formations sont généralement destinées aux utilisateurs et aux auteurs de logiciels scientifiques, et malheureusement pas aux concepteurs des supercalculateurs.

IV - Conclusion

La reproductibilité numérique est importante pour la simulation et elle est nécessaire pour la recherche scientifique. Il y a principalement deux types de reproductibilité : la reproductibilité des expériences au sens scientifique du terme reproductibilité (c'est-à-dire une personne indépendante peut répéter et/ou reproduire une simulation numérique sur son système) et aboutir aux mêmes conclusions scientifiques sans avoir les mêmes résultats et la reproductibilité numérique (c'est-à-dire le résultat obtenu par le calcul numérique est identique à ce qui a été fait auparavant ou ce qui a été publié par l'auteur). La reproductibilité d'expérience est fortement liée à la culture scientifique et la reproductibilité numérique touche particulièrement ceux qui font des expériences computationnelles, si les exécutions ne sont plus déterministes sur un système de calcul, nous ne sommes plus en présence d'une machine de Turing authentique et il deviendra impossible de mettre au point les programmes dont les résultats changent d'une exécution à l'autre. Tous les problèmes ne se réduisent pas au calcul en virgule flottante, le problème est plus profond et multifactoriel comme nous l'avons montré dans ce chapitre.

Obtenir une reproductibilité des d'expériences computationnelles est nécessaire pour la reproductibilité numérique et pour l'avancement de la Science. Si l'auteur ne partage pas ses codes et ses données, alors on ne peut rien faire. S'il partage ces informations dans les publications

(comprenant un accès aux codes sources, aux paramètres d'entrée, aux données utilisées…) alors on "peut" tester une répétabilité ou une reproductibilité numérique. Nous tenons à souligner le mot "peut" parce qu'on a besoin de bien des éléments.

Même si la reproductibilité de l'expérience est acquise, la reproductibilité de la simulation numérique reste encore une tâche difficile parce qu'il y a de nombreux problèmes identifiés comme nous l'avons montré dans ce chapitre, notamment l'impact de l'environnement d'exécution, le calcul en virgule flottante. Le cas de la simulation stochastique parallèle utilisant un système de calcul à hautes performances est encore plus complexe que celui d'une application scientifique purement déterministe. Les problèmes dans le cadre de cette thèse sont également liés aux problèmes de génération de nombres pseudo-aléatoires en parallèle (comprenant l'initialisation, le générateur, la technique de parallélisation des flux stochastiques), à la conception du logiciel et aussi aux nouvelles architectures matérielles pour le calcul parallèle.

Taufer (Taufer 2012) a souligné que beaucoup des solutions matérielles et logicielles avaient été proposées pour résoudre les problèmes de la reproductibilité et de stabilité numérique, mais qu'elles ne marchaient pas pour les systèmes Exascale qui seront principalement des machines hybrides avec processeurs classiques et accélérateurs. Les changements à apporter dépendent de l'architecture physique proposée par le fournisseur du matériel.

Dans ce chapitre, nous avons recensé les principales causes de non-reproductibilité pour la simulation numérique et notamment la simulation stochastique parallèle utilisant des systèmes de calcul à hautes performances. Parce que ces causes ne sont pas indépendantes les unes des autres, on doit considérer la non-reproductibilité d'une simulation numérique dans une vision d'ensemble.

Sur le plan logiciel, les problèmes de non-reproductibilité numérique rencontrés dans la simulation stochastique parallèle (différence de résultats entre les versions séquentielles, entre les versions séquentielles parallèles et entre version parallèles) dépendent de la conception du programme, du ou des générateurs de nombres pseudo-aléatoires, des états d'initialisation de ces générateurs, de la technique de parallélisation et même des options de compilation. Pour obtenir la reproductibilité numérique, il faut être capable de gérer au moins ces facteurs de façon satisfaisante. Des solutions existent et seront présentées et analysées dans les chapitres qui suivent.

Chapter 2 - Simulation Stochastique Parallèles

I - Introduction

Le chapitre 1 a introduit les causes de non-reproductibilité numérique que nous avons rencontrées dans divers types de simulation et dans le calcul numérique en général, avec une attention particulière pour les simulations stochastiques parallèles qui sont au cœur de cette thèse. Les méthodes de Monte Carlo utilisant des systèmes de calcul à haute performance constituent un cas d'application qui nous intéresse en particulier. Une simulation stochastique parallèle de type Monte Carlo est un programme déterministe dont les résultats obtenus devraient être toujours reproductibles si l'on est rigoureux dans son implémentation. Mais nous avons observé que ce n'est pas souvent le cas, il existe en effet bien des causes de non reproductibilité que nous avons présentées dans le chapitre 1. Les systèmes de calcul à haute performance sont essentiels pour une amélioration des performances, mais ils peuvent également être mis en cause dans la perte de reproductibilité.

Dans ce chapitre, nous allons continuer notre présentation des notions nécessaires à une bonne compréhension de ce manuscrit. Nous allons donner un aperçu des simulations dites de Monte Carlo en rappelant qu'elles reposent essentiellement sur l'utilisation de générateurs de nombres pseudo-aléatoires. Nous présenterons ensuite les différentes techniques de parallélisation des générateurs de nombres pseudo-aléatoires. Enfin, nous réviserons les architectures parallèles en usage actuellement avec une attention particulière pour celles utilisant de nombreux cœurs de calcul et de threads.

II - La simulation stochastique parallèle

La simulation est une technique qui suppose de faire évoluer l'état d'un modèle qui représente un système. L'état de ce système est représenté par un ensemble de variables. Lorsque ces variables changent de valeur, de façon continue ou discrète, nous avons changé d'état. Cette évolution de l'état du système modélisé peut se faire ou non au cours du temps. La simulation est une des méthodes importantes de la recherche scientifique, on en cite de nombreux ~~d'~~avantages dans (Banks 1998) ou dans l'article de (Traoré et Hill 2003) qui s'intéresse à l'épistémologie de la

simulation (façon dont on construit de nouvelles connaissances grâce aux simulations). Parmi ces avantages, nous pouvons citer : observer (passivement) le système, assimiler son mode de fonctionnement nominal, évaluer les performances du système, prédire le futur du système dans sa structure et dans son fonctionnement, interagir avec le système dans des simulations visuelles interactives, etc. Le domaine d'application des simulations est très large de la fabrication et la manutention de matériaux au génie civil, l'aéronautique, la médecine, les applications militaires, l'environnement, etc. Parmi les différents types de simulation existants, notre attention se porte sur les simulations stochastiques. Celles-ci utilisent des modèles reposant sur des mécanismes aléatoires. Au niveau des implémentations, les générateurs de nombres pseudo-aléatoires sont des composants essentiels de ces simulations. Ils sont utilisés pour générer les variables aléatoires des modèles stochastiques. Ce type de simulation est facile à paralléliser, grâce notamment à la production de nombreux flux et sous-flux « indépendants » de nombres pseudo-aléatoires. Hill (Hill 2014) a mentionné que ce type de simulation est très fréquemment utilisé dans le contexte du calcul à haute performance et notamment sur les supercalculateurs.

De fait, plus de 50% des simulations réalisées sur des supercalculateurs sont stochastiques, elles couvrent des domaines très variés avec notamment les prévisions météorologiques ou environnementales (tremblement de terre, tsunami,…), l'exploration minière et pétrolifère, la biologie moléculaire, mais aussi plus récemment, la recherche d'information sur Internet, la finance et la simulation de modèles quasi-complets de cerveaux (Hill 2014).

Nous présenterons dans cette section la simulation stochastique avec la méthode dite de Monte Carlo. Ensuite, nous aborderons sommairement les générateurs de nombres pseudo-aléatoires, les techniques de parallélisation de ces générateurs pour les applications stochastiques dans un environnement de calcul à haute performance et aussi, les générateurs capables de s'adapter aux nouveaux matériels d'accélération parallèle des calculs.

II.1 - La Simulation dite de Monte Carlo

Tout d'abord, mentionnons brièvement l'histoire de la méthode dite de Monte Carlo. Le nom de cette méthode provient du casino de Monte Carlo, réputé pour ses jeux de hasard. L'idée de la méthode Monte Carlo a été présentée par Nicholas Metropolis et Stanislaw Ulam en 1949 pour l'étude des équations différentielles et intégrodifférentielles en général (avec au départ une équation impliquant deux intégrales et la dérivée d'une fonction). Ensuite, cette méthode a été

appliquée par Stanislaw Ulam et John von Neumann pour résoudre le problème de la distribution de neutron dans des matériaux fissibles. L'idée de cette méthode repose essentiellement sur une approche statistique.

En simulation numérique, la technique statistique est utilisée pour modéliser les systèmes probabilistes (ou stochastiques). C'est-à-dire, on cherche à compter des événements ou des états du modèle stochastique qui nous intéressent à partir d'un ensemble d'entrées aléatoires. Après avoir répété les expériences indépendantes un grand nombre de fois, on calcule statistiquement le taux de succès. Une simulation de Monte Carlo est donc une simulation d'un modèle stochastique utilisant une technique statistique de résolution. Le résultat obtenu par la méthode Monte Carlo est un résultat approché que l'on encadre par un intervalle de confiance.

Dans Coquillard et Hill (Coquillard and Hill 1997) les différentes notions utiles aux simulations de Monte Carlo sont présentées : système, modèle, méthode, expérience, générateurs de nombres aléatoires, variables aléatoires, etc. De plus, ils ont aussi indiqué que toutes les simulations discrètes stochastiques peuvent être considérées comme des simulations de Monte-Carlo.

On peut parler de méthode de Monte-Carlo lorsque la solution utilise un générateur de nombres pseudo-aléatoires. Une expérience de Monte-Carlo se base sur des modèles du hasard où la représentation du temps n'est pas nécessaire. Par contre si le modèle évolue également de manière dynamique dans le temps, on peut encore lui donner le nom de ***simulation de Monte-Carlo****. Cette technique consiste à résoudre un problème non probabiliste complexe grâce à l'utilisation d'un processus stochastique qui satisfait les relations du problème déterministe. Le caractère stochastique des simulations de Monte-Carlo conduit à de nombreux essais successifs du modèle et au calcul de statistiques. Par extension, tous les modèles de simulation discrète stochastique peuvent être considérés comme des simulations dites de Monte-Carlo (Coquillard and Hill 1997).*

Raychaudhuri (Raychaudhuri 2008) explique qu'une simulation de Monte Carlo se compose des phases suivantes : le choix d'une distribution statistique, le tirage des échantillons aléatoires, la collecte des valeurs de sortie, l'analyse statistique des sorties, etc. Il a donné le mode opératoire d'une simulation Monte Carlo étape par étape.

In Monte Carlo simulation, we identify a statistical distribution which we can use as the source for each of the input parameters. Then, we draw random samples from each distribution, which then represent the values of the input variables. For each set of input parameters, we get a set of output parameters. The value of each output parameter is one particular outcome scenario in the simulation run. We collect such output values from a number of simulation runs. Finally, we perform statistical

analysis on the values of the output parameters, to make decision about the course of action (whatever it may be). We can use the sampling statistics of the output parameters to characterize the output variation (Raychaudhuri 2008).

On peut voir dans la définition de Raychauduri que le générateur de nombres pseudo-aléatoires joue un rôle essentiel dans la distribution statistique.

Kroese (Kroese et al. 2014) a présenté des nombreuses raisons pour expliquer pourquoi la méthode de Monte Carlo est si importante aujourd'hui et dans l'avenir : facilité et efficacité, prise en compte de l'aléatoire, justification théorique, etc. En plus, selon Kroese, son domaine d'application est très vaste et il demande que plus de recherches relatives à la méthode Monte Carlo soient conduites à l'avenir, notamment sur le calcul parallèle, l'analyse d'erreur non-asymptotique, l'algorithme Monte Carlo adaptatif, la simulation améliorée des processus spatiaux, les événements rares et l'approche de quasi Monte Carlo.

En effet, la simulation stochastique avec la méthode Monte Carlo est un outil très important pour la recherche scientifique. Elle est d'ailleurs majoritairement utilisée dans les cas où il est impossible ou infaisable de faire un calcul exact avec un algorithme déterministe.

II.2 - Les générateurs de nombres pseudo-aléatoires

Nous venons de voir que les sources aléatoires jouent un rôle très important pour les simulations stochastiques utilisant la méthode de Monte Carlo. Ces sources sont utilisées en particulier pour générer les variables aléatoires. Dans ce manuscrit, nous travaillons seulement sur les générateurs de nombres pseudo-aléatoires bien qu'on distingue 2 autres classes de sources aléatoires : les sources aléatoires pures avec des dispositifs physiques et celles dites quasi-aléatoires. Hill (Hill 2014) distingue ces sources de la façon suivante :

Les générateurs pseudo-aléatoires sont la catégorie de générateurs la plus utilisée dans le contexte des simulations informatiques stochastiques. Il ne s'agit pas d'un dispositif cherchant une 'source' aléatoire pure (TRNG – True Random Generator), mais comme dans le cas des générateurs quasi-aléatoires, il s'agit d'un algorithme. Par contre, contrairement aux générateurs QRNG (Quasi-Random Generator) qui ne reproduisent pas vraiment les caractéristiques statistiques du hasard, les générateurs pseudo-aléatoires (PRNG : Pseudo-Random Number Generator) cherchent à mimer au mieux les propriétés du hasard…. (Hill 2014).

Selon tout le monde, les PRNGs (Pseudo Random Number Generators) sont des algorithmes et peuvent reproduire des suites de nombres qui cherchent à reproduire au mieux les caractéristiques statistiques du hasard. La compréhension du maniement des générateurs est donc une compétence nécessaire pour la simulation stochastique parallèle utilisant des systèmes de calcul à haute performance. Par la suite, nous présentons notre définition d'un générateur pseudo-aléatoire et nous utilisons l'acronyme anglo-saxon « PRNG » qui est très répandu. Nous regarderons aussi les critères permettant de savoir si l'on est en présence d'un « bon » générateur.

Définition d'un générateur de nombre pseudo-aléatoire

Un générateur de nombres pseudo-aléatoires (PRNG : Pseudo-Random Number Generator) est un algorithme « déterministe » capable de générer les nombres mimant une source aléatoire uniforme utilisée pour la simulation stochastique. En informatique, un algorithme déterministe est un algorithme qui produira toujours la même sortie quand on utilise la même entrée. Clairement, la propriété déterministe est vraiment importante pour ce générateur parce qu'il nous permet d'obtenir la reproductibilité numérique des flux de nombres pseudo-aléatoires générés. Cette propriété de reproductibilité des générateurs, bien mise en avant par Michael Mascagni (Pryor et al. 1994), est essentielle pour le débogage et la traçabilité des applications scientifique stochastiques.

A partir des générateurs suivant des lois uniformes, il est possible de générer n'importe quel type de variables aléatoires. L'Ecuyer (L'Ecuyer 1990), présente un générateur générique de variables aléatoires comme une structure mathématique qui se compose d'un ensemble fini d'états, possédant une distribution initiale, une fonction de transition, un ensemble fini de sorties et une fonction de sortie.

A generator is a structure G = (S,μ,f,U,g), where S is a finite set of states, μ is a probability distribution on S, called the initial distribution, f: S → S is the transition function, U is a finite set of output symbols, and g: S → U is the output function.

A generator operates as follows: (1) Select the initial state $s_0 \in S$ according to μ; let $u_0 := g(s_0)$; (2) for i:= 1,2,…, let $s_i := f(s_{i-1})$ and $u_i := g(s_i)$. The sequence of observations (u_0, u_1, u_2...) is the output of the generator. The initial state s_0 is called the seed. (L'Ecuyer 1990)

Le terme « seed » ou graine proposé par L'Ecuyer n'est plus adapté aux générateurs modernes qui ont des états initiaux conséquents. Suite aux travaux du LIMOS et à la venue de Pierre L'Eucuyer au

jury de thèse de Jonathan Passerat-Palmbach, nous notons que la terminologie proposée par l'Ecuyer a changée et il utilise maintenant le terme « statut » et « statut initial » dans une de ces dernières publications (L'Ecuyer et al. 2015b). Il faut revenir à la notion d'état initial que nous appelons statut. Partageant les mêmes idées sur la structure mathématique des générateurs, Srinivasan (Srinivasan et al. 2003) a aussi présenté une définition similaire. Néanmoins, il a ajouté à sa définition la notion de période d'un générateur et des cas concrets de la fonction de sortie. Selon lui, cette fonction peut donner un nombre pseudo-aléatoire entier ou en virgule flottante. La Figure 2.1 tirée de ses travaux illustre le fonctionnement d'un PRNG : le caractère I_n présente un nombre entier, le caractère U_n présente un nombre en virgule flottante et un cycle présente la période d'un générateur. Après une période, les nombres pseudo-aléatoires seront répétés, il est donc essentiel que cette période soit suffisamment grande, ce qui est le cas de tous les générateurs modernes.

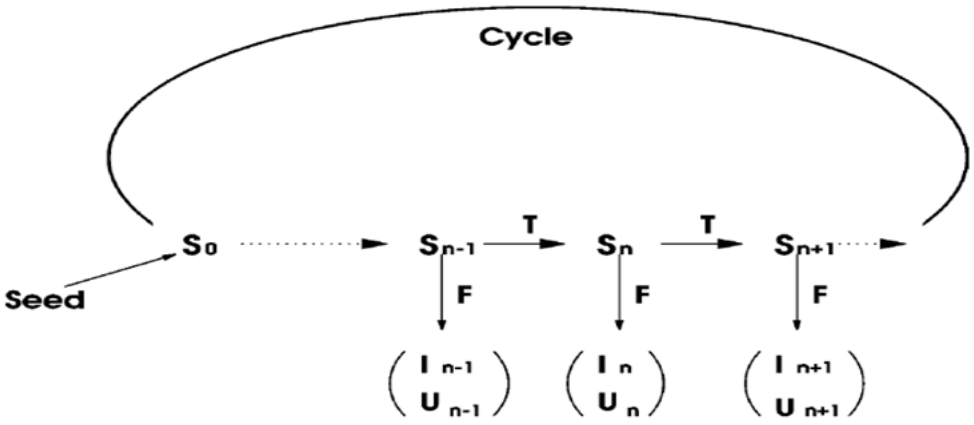


Figure 2.1. Graphique présentant le fonctionnement d'un PRNG (Srinivasan et al. 2003)

Matsumoto (Matsumoto et al. 2006) a proposé une définition utilisant le concept d'automate à états finis. Selon lui, les générateurs peuvent être vus comme des automates à états finis sans entrée. L'état est changé par récursion d'une fonction de transition. Il propose aussi une définition pour les générateurs de nombres pseudo-aléatoires qui souligne l'importance de l'aspect déterministe de l'algorithme employé et de ses caractéristiques statistiques.

A pseudorandom number generator is an algorithm generating a sequence of numbers that appears to be random, by some deterministic algorithm. (Matsumoto et al. 2006)

Nous voulons souligner là aussi le mot « déterministe » car, comme nous l'avions annoncé, il revêt une grande importance pour la reproductibilité d'un générateur. Une simulation stochastique est donc déterministe lorsqu'elle utilise un PRNG pour un statut initial donné.

Un « bon » générateur ?

Tout comme nous l'avons présenté au Chapitre 1 la qualité des générateurs est d'autant plus importante que les simulations stochastiques font usage de machines parallèles. Il est donc nécessaire d'identifier des critères de qualité permettant de savoir si l'on est en présence de « bons » générateurs.

Coddington (Coddington 1997) a donné une liste des exigences pour caractériser les « bons » générateurs dans le cadre d'une utilisation parallèle. Cette liste propose 9 points : l'uniformité, la non-corrélation, l'aspect aléatoire, la qualité statistique face à des tests, la reproductibilité, la portabilité, la capacité à modifier les suites de nombres grâce à des initialisations par des statuts différents, la capacité à être découpé en multiples sous-flux, et la vitesse de génération des nombres pseudo-aléatoires avec une consommation de mémoire ou empreinte mémoire modérée.

> *Ideally a pseudo-random number generator would produce a stream of numbers that*
>
> 1. *are uniformly distributed,*
> 2. *are uncorrelated,*
> 3. *never repeats itself,*
> 4. *satisfy any statistical test for randomness,*
> 5. *are reproduceable (for debugging purposes),*
> 6. *are portable (the same on any computer),*
> 7. *can be changed by adjusting an initial "seed" value,*
> 8. *can easily be split into many independent subsequences,*
> 9. *can be generated rapidly using limited computer memory.*

> *In particle it is impossible to satisfy all these requirements exactly. Since a computer uses finite precision arithmetic to store the state of the generator, after a certain period the state must match that of a previous iteration, after which the generator will repeat itself. Also, since the numbers must be reproducible, they are not truly random, but generated by a deterministic iterative process, and therefore cannot be completely uncorrelated (Coddington 1997).*

Coddington a souligné dans le dernier paragraphe qu'il est « impossible » de satisfaire exactement toutes les exigences qu'il préconise. De plus, il existe toujours des conflits entre ces exigences.

Hellekalek (Hellekalek 1998) a également présenté 8 exigences : l'uniformité, la non-corrélation, la portabilité, la reproductibilité, la période, la capacité d'analyser la structure interne de la séquence de sortie, la capacité à paralléliser et l'obtention nécessaire de bons résultats pour des simulations stochastiques arbitraires. Il a détaillé ces exigences et précisé ce qui n'est pas atteignable en raison

de l'état actuel de la théorie sur les PRNG. Selon lui, la portabilité et la reproductibilité sont incontournables. En fait, la reproductibilité numérique contient la notion de portabilité, comme l'exprime Hellekalek ci-dessous :

> *They should be (i) uniformly distributed, (ii) uncorrelated, (iii) reproducible, in order to debug programs and simulation models, and (iv) portable, in order to get the same random numbers on various platforms. Further, (v) their period should be several magnitudes larger than the largest samples we use, (vi) the inner structure of the output sequence should be analyzable, (vii) parallelization should be possible, and (viii) in arbitrary stochastic simulations, we should get the right results.*
>
> *Reproducibility and portability of an RNG, see (iii) and (iv) above, is a must. It is a question of clever scientific programming, see Lendl [24] for an instructive example. Points (i), (ii), (v) and (vi) are the main task of the theory of RNGs. Condition (vii) is difficult to satisfy, as we will see later in this paper, and (viii) is an impossible dream (Hellekalek 1998).*

Coddington et Hellekalek ont donc 5 exigences communes pour un « bon » générateur séquentiel. Par contre, Coddington présente en plus 5 exigences pour un « bon » générateur parallèle : « le PRNG devrait fonctionner pour n'importe quel nombre de processeurs, la séquence de nombres aléatoires générés sur chaque processeur devrait satisfaire toutes les exigences d'un bon générateur séquentiel, il ne doit pas y avoir de corrélation entre les séquences générées sur des processeurs différents, la même séquence de nombres aléatoires devrait être produite pour un nombre quelconque de processeurs et dans le cas spécial d'un seul processeur, et l'algorithme devrait être efficace (c'est-à-dire, il n'y aura aucun mouvement de données entre les processeurs) ». Fondamentalement, la première exigence correspond à une forme de portabilité d'un générateur de nombres pseudo-aléatoires pour un travail en parallèle. La quatrième concerne la reproductibilité numérique du flux généré par un PRNG entre séquentiel et parallèle, ou entre parallèle et parallèle. En d'autres termes, la séquence générée doit être « *indépendante* » avec le nombre de processeurs utilisés.

Avec ce que nous avons vu, les notions de portabilité et de reproductibilité sont rappelées comme étant des caractéristiques importantes pour un PRNG. Pour ces deux caractéristiques, Dao (Dao et al. 2014a) a présenté une méthode pour étudier la portabilité et la reproductibilité numérique d'un générateur. Nous présenterons ces travaux dans les chapitres suivants. Par contre, en ce qui concerne les caractéristiques théoriques des PRNG, elles sont très difficiles à évaluer en raison de l'absence d'une définition adéquate de la notion de pseudo-aléatoire. Cette limite a été indiquée par

Matsumoto (Matsumoto et al. 2006). Selon lui, il n'existe pas de théorie ou de bonne méthode de test qui permettent de s'assurer de l'aspect vraiment « pseudo-aléatoire » lors de l'évaluation d'un PRNG. Seuls des tests statistiques et empiriques sont disponibles et ils ne s'adaptent pas forcément à tous les générateurs.

Because of the lack of an adequate definition of pseudorandomness, there is no theory nor are there methods that assure genuine pseudorandomness.

Statistical tests are empirical, and the obtained evaluation is probabilistic and has some uncertainty.

A problem of theoretical evaluations is that a different type of generator has a different type of mathematical structure. Consequently, one test is applicable only to one type of generators, which makes it difficult to compare different types of generators by such a test (Matsumoto et al. 2006).

Du fait de cette remarque de Matsumoto, il ressort qu'il n'est pas facile de certifier que nous sommes en présence d'un « bon » générateur. Il est par contre possible d'identifier des mauvais générateurs qui suite à des évaluations probabilistes sur des tests statistiques empiriques, montrent des faiblesses significatives. Néanmoins, concernant la qualité des résultats numériques obtenus avec des générateurs modernes satisfaisant aux critères de qualité statistique, il convient de s'assurer cependant de la portabilité et de la reproductibilité de ces générateurs.

Nous avons présenté les exigences théoriques pour un bon générateur (séquentiel et parallèle) dans cette sous-section. Dans la suivante, nous aborderons un logiciel de test statistique qui fait référence pour l'évaluation de la qualité des générateurs.

TestU01 et quelques méthodes de test parallèle

TestU01 est une bibliothèque en langage C, développée par P. L'Ecuyer et R. Simard (L'Ecuyer and Simard 2007), dont le but est d'effectuer des tests statistiques empiriques sur des générateurs de nombres pseudo-aléatoires.

Pour un générateur séquentiel, on utilise des batteries de test pour évaluer son caractère pseudo-aléatoire. TestU01 fournit 3 principales batteries :

1. La plus petite batterie, « SmallCrush », a besoin d'un peu moins de 51 320 000 nombres pseudo-aléatoires en virgule flottante dans [0,1), stockés dans un fichier texte et appliqués dans 10 tests statistiques.

2. La batterie « Crush » utilise environ 2^{35} nombres pseudo-aléatoires dans 96 tests statistiques. Elle calcule au total 144 statistiques et p-values
3. La plus grande batterie, « BigCrush », utilise environ 2^{38} nombres pseudo-aléatoires dans 106 tests statistiques. Elle calcule au total 160 statistiques et p-values.

En d'autres termes, les trois batteries sont composées d'un ensemble plus ou moins grand de tests statistiques différents, appliqués à un nombre de plus en plus grand de tirages pseudo-aléatoires. Ces batteries comprennent les tests classiques décrits dans Knuth et bien d'autres, citons par exemple : « *equidistribution (frequency), serial, gap, poker (partition), coupon collector, permutation, run, maximum-of-t, collision, serial correlation, tests on subsequences,…* » .

Pour tester des flux stochastiques pour des utilisations parallèles, nous disposons des approches proposées par (L'Ecuyer and Simard 2007), Srinivasan (Srinivasan et al. 2003), Salmon (Salmon et al. 2011) et Ismay (Ismay 2013). Nous classifions ces approches dans les 2 principales catégories, en fonction des générateurs proposant un grand flux unique, des multiples flux (en continant deux technique de test : univarié et multivarié), comme illustré sur la Figure 2.2.

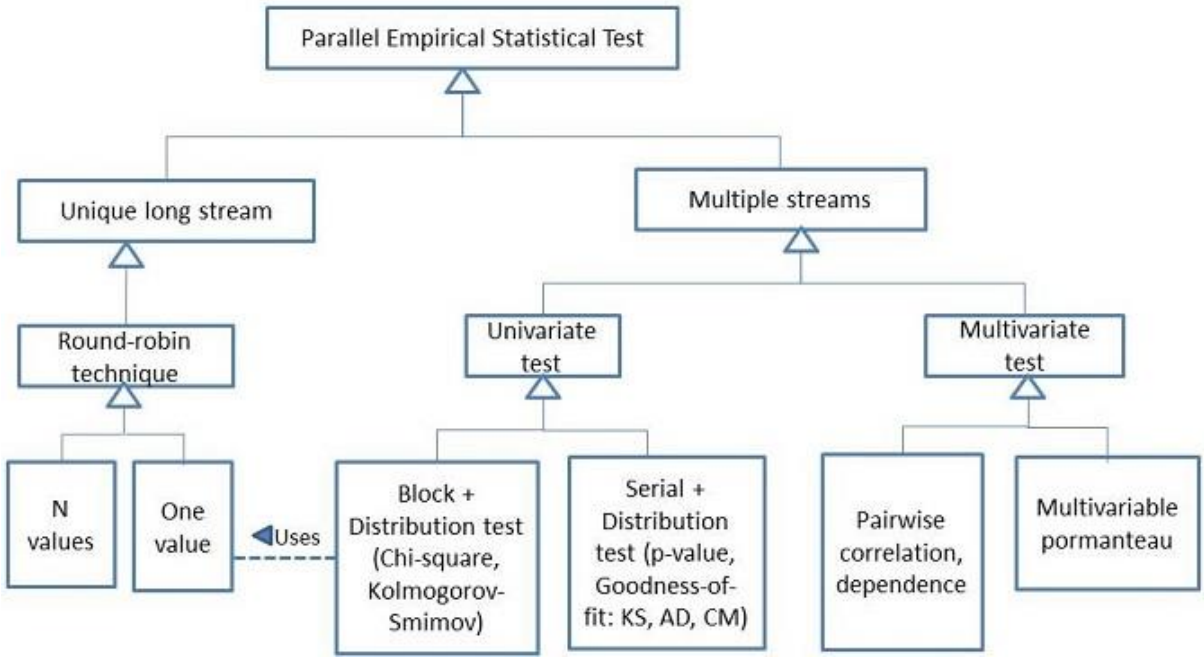


Figure 2.2. Une taxonomie des techniques de tests statistiques empiriques de TestU01 pour une utilisation parallèle d'un PRNG

Du fait que TestU01 a été initialement conçu pour tester un flux unique, l'idée principale est d'effectuer d'abord la fusion des sorties d'un ou plusieurs générateurs (ou de sous-parties différentes d'un même générateur) en un seul flux de nombres pseudo-aléatoires, puis de tester ce flux avec une méthode dite de « tourniquet » (« *round-robin* »). Pour cette idée, l'approche de

Srinivasan et de Salmon propose une taille pour les tourniquets avec N valeurs et cela correspond au « parallel filter » dans le logiciel TestU01.

L'idée de tester des blocs de nombres issus de multiples flux avec le test univarié a été présentée par Srinivasan et Mascagni dans la suite logicielle SPRNG (Mascagni et Srinivasan 2000). Il est possible d'utiliser TestU01 pour des tests équivalents dits tests de deux niveaux (*two-level tests*). TestU01 permet de tester diverses combinaisons de lignes et de colonnes (L'Ecuyer et Simard 2007). TestU01 fournit quelques fonctions pour tester des générateurs parallèles (*unif01_CreateParallelGen, unif01_DeleteParallelGen*) même si ce n'est pas une fonctionnalité phare de cette suite logicielle. En effet, selon Ismay (Ismay 2013), ces techniques ne sont pas suffisantes parce qu'elles ne prennent pas en compte la nature multivariée des flux fournis par des générateurs exécutés en parallèle. Il a donc proposé des extensions (*mccor, mmult, mport*) pour TestU01 afin de tester des générateurs parallèles. Ismay propose en plus de créer une matrice des flux de nombres pseudo-aléatoires pour détecter des dépendances entres les flux.

Pour évaluer les qualités d'un PRNG pour une utilisation en parallèle, il doit être testé d'une part en séquentiel et d'autre part avec des tests parallèles. A notre avis, il arrive en effet qu'un « bon » générateur séquentiel puisse ne pas avoir de grande aptitude au parallélisme, mais le cas reste rare.

Même si l'évaluation de la qualité des PRNGs n'est que relative, les générateurs modernes identifiés comme ayant des bonnes propriétés pour des simulations séquentielles avec TestU01 seront fréquemment évalués positivement par d'autres approches de test : les aspects aléatoires et non-corrélation avec les tests de Matsumoto (Matsumoto et al. 2006) et les tests parallèles de Ismay (Ismay 2013), la portabilité et la reproductibilité selon une méthode qui sera présentée par la suite (Dao et al. 2014a), ou l'uniformité et l'indépendance à partir des choix d'initialisation avec les tests empiriques de Kim et Yang (Kim et Yang 2006).

II.3 - Les techniques de parallélisation des générateurs de nombres pseudo-aléatoires

Les simulations stochastiques utilisant la méthode de Monte Carlo sont faciles à paralléliser grâce aux techniques de parallélisation des générateurs de nombres pseudo-aléatoires. Le principal écueil rencontré reste le fait que ces techniques sont méconnues des scientifiques. Les techniques utilisées pour la parallélisation des générateurs de nombres pseudo-aléatoires sont diverses selon qu'elles exploitent une approche à base de serveur central (intéressante pour les jeux et à fuir pour les applications scientifiques), à base de multiples flux ou de multiples sous-flux stochastiques. Ces

différentes techniques sont présentées partiellement dans (Hellekalek 1998), (Coddington et Ko 1998) et (Srinivasan et al. 1998 and 2003) et de façon plus complète dans (Hill et al. 2013) et (L'Ecuyer et al. 2015a).

Dans l'article (Hill et al. 2013), nous trouvons une analyse complète des techniques de parallélisation des PRNGs. De plus, on y trouve une taxonomie claire et détaillée sous forme de méta-modèle UML que nous reprenons dans la Figure 2.3.

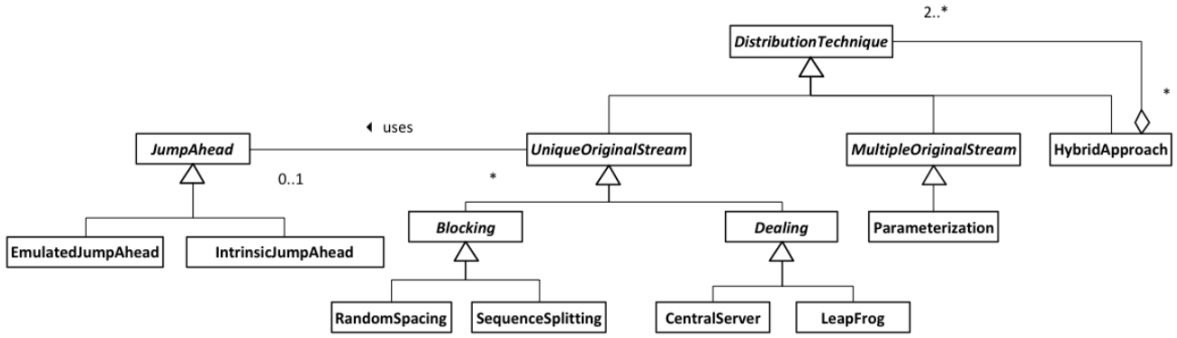


Figure 2.3. Méta-modèle UML proposant une taxonomie des techniques de distribution et de parallélisation des flux stochastiques (Hill et al. 2013)

Cette Figure 16 classe les techniques de parallélisation avec un diagramme de classe en UML (*Unified Modeling Language*). Elle montre deux principales techniques de parallélisation : la première se base sur le découpage d'un grand flux en multiples sous-flux avec la superclasse « *UniqueOriginalStream* », et la deuxième utilise de multiples flux avec la superclasse « *MultipleOriginalStream* ». Hill et ses collègues ont ajouté une troisième classe pour prendre en compte les approches combinant les deux approches précédentes, nommée « *HybridApproach* ». Dans ce méta-modèle, nous retrouvons aussi une classe pour représenter la technique nommée « *JumpAhead* ». Cette technique est utilisée pour proposer rapidement de multiple sous-flux à partir d'un grand flux. Cet article présente également des conseils sur les bonnes techniques à utiliser avec les meilleurs générateurs actuels pour différents types d'architectures de calcul à haute performance, notamment des cartes accélératrices à base de GP-GPUs (*General Purpose – Graphical Processing Units*). Récemment, L'Ecuyer (L'Ecuyer et al. 2015a) a également présenté ces techniques de parallélisation de flux et sous-flux pour les générateurs modernes. Il présente aussi différentes manières d'attribuer les flux en réalité :

- **un flux par thread**: pour L'Ecuyer, cette méthode utilise un moniteur central qui calcule séquentiellement les états initiaux pour différents flux et qui les assigne aux threads (matériels), en utilisant une technique de découpage (connue sous le terme anglo-saxon *sequence splitting*). Une fois que le thread dispose de son état initial, il peut évoluer de

façon autonome, il n'a plus besoin du moniteur central. Cette approche est reprise dans JAPARA (Coddintong et Newell 2004). Néanmoins, le résultat de la simulation dépend de l'architecture et de l'organisation du matériel, notamment de la manière dont sont attribués les threads. En effet, des écarts sur les résultats, voire des dysfonctionnements ont été observés sur des matériels identiques, principalement lorsque les threads sont lancés sur des GP-GPUs (Taufer et al. 2010) ; la reproductibilité numérique n'est pas forcément au rendez-vous avec cette approche.

- **un flux par tâche**: cette approche ne dépend pas du matériel. Comme dans l'exemple précédent, nous avons la reproductibilité des flux qui est essentielle, ce ne serait que pour déboguer l'application. Un exemple récent utilisant cette approche est proposé par notre équipe dans les travaux de Passerat-Palmbach (Passerat-Palmbach et al. 2014). Cette approche se nomme *TaskLocalRandom* et utilise un flux de nombres pseudo-aléatoire appelé *RngStream* pour paralléliser des simulations en Java. Chaque tâche est donc créée avec une identification unique – un ID_t et un flux est assigné à cette identification. Néanmoins, cette approche ne permet pas d'avoir plusieurs flux par tâche dans le modèle de simulation. L'autre inconvénient consiste dans l'utilisation du langage Java pour le calcul scientifique. En effet, la représentation et l'implémentation de l'arithmétique flottante en Java ont été dénoncées depuis longtemps par des collègues de Berkeley [Kahan et Darcy 1998].
- **plusieurs flux par tâche** : les flux peuvent être créés par un moniteur central et passés aux tâches par la même manière que tous les autres données ou objets utilisés par ces tâches. Pour obtenir une reproductibilité numérique et une bonne synchronisation, il faut pouvoir s'assurer que les flux soient toujours créés dans le même ordre. L'ordre d'exécution des tâches et leur synchronisation sont également importants pour obtenir une reproductibilité numérique. Ce procédé a été utilisé plus tard pour son le générateur clRNG avec les deux fonctions *inventorySimulateRunsSubStreams, inventorySimulateRuns ManyStreams* (L'Ecuyer et al. 2015b).

L'implémentation d'une technique de parallélisation et l'attribution des flux aux tâches peuvent être faciles pour des calculs indépendants, ou compliquées, par contre, dans le cas de synchronisation entre tâches, en fonction du principe de fonctionnement du PRNG retenu ou du type d'architecture matérielle retenue. Des conseils pour ces cas ont été proposés dans (Hill et al. 2013). Néanmoins, la spécificité de la reproductibilité numérique des simulations stochastiques parallèles n'était pas abordée. Parmi les aspects et contraintes à suivre, les cinq exigences de Coddington montrées dans

la partie 2.b de ce chapitre 2 sont très importantes, à savoir : l'indépendance du PRNG par rapport au nombre de processeurs (ou plutôt de cœurs de calculs dans le cas actuel), la non-corrélation entre flux ou sous-flux, la reproductibilité des flux ou des sous-flux, et l'efficacité d'un PRNG. Dans l'idée de Passerat-Palmbach (Passerat-Palmbach et al. 2014), on a besoin d'une identification pour chaque thread et chaque flux. Par contre, l'idée proposée par L'Ecuyer dans (L'Ecuyer 2015b) utilise deux flux pseudo-aléatoires différents *stream_demand* et *stream_order* pour générer un flux demandé et garantir l'ordre d'un flux.

II.4 - L'adaptation des PRNGs aux nouveaux matériels de traitement parallèle

Pour les nouvelles architectures matérielles conçues pour le traitement parallèle, il faut que les meilleurs générateurs soient bien portables pour obtenir le 1er niveau de répétabilité. Dans cette sous-section, nous présentons une liste des PRNGs qui s'adaptent aux nouvelles architectures de traitement parallèle que nous aborderons dans le chapitre suivant. Ces PRNGs sont déjà de bons générateurs modernes de nombres pseudo-aléatoires en séquentiel, créés pour des CPU séquentiels, mais on peut les utiliser sur des architectures parallèles. Il existe aussi de très bons générateurs modernes qui exploitent des formes de parallélisme précises, telle que la vectorisation avec des CPUs modernes ou des GPUs.

L'Ecuyer (L'Ecuyer et al. 2015a) a confirmé qu'on n'a pas besoin de changer l'implémentation des bons PRNGs modernes parce qu'ils s'adaptent simplement au calcul parallèle. Il a présenté des modèles de calculs parallèles utilisant la génération de nombres pseudo-aléatoires et la simulation. Il a également discuté les méthodes spécifiques et les logiciels proposés pour chacune des approches qu'il décrit.

Most of the best currently available RNGs have been designed for conventional CPUs with « standard » instruction sets and plenty of fast-access memory. To use them on a parallel computer whose PEs have these characteristics and can execute their instructions independently of each other at any given time step, it suffices to make multiple streams as explained earlier. There is no need to change their implementation.

The PEs (cores) in recent multi-core CPUs and APUs share a large common fast-access memory in addition to their separate registers, so they can accommodate the same RNGs as in traditional CPUs. The hundreds of thousands of PEs in largest supercomputers today also behave much like traditional CPUs. (L'Ecuyer et al. 2015a).

L'Ecuyer a aussi présenté deux approches pour les PRNGs qui s'adaptent aux nouveaux matériels de traitement parallèle. Dans la première approche, plusieurs flux peuvent évoluer en parallèle, chaque flux fonctionne sur un simple élément de traitement (« *Processing Element* » – PE). Pour la deuxième, l'objectif est de produire les nombres pseudo-aléatoires soit à partir d'un flux unique, soit à partir de plusieurs flux à la fois avec donc une vitesse plus rapide que ce qui serait possible sur un seul PE. Pour ce faire, il propose une approche vectorielle avec un groupe de plusieurs PEs. Si dans la première approche, les nombres pseudo-aléatoires sont souvent consommés localement sur le PE où ils sont générés, dans la $2^{ème}$, ces nombres sont généralement copiés dans une mémoire globale et consommés ailleurs ou stockés pour une utilisation ultérieure.

Jusqu'à maintenant (2016), les scientifiques disposent de beaucoup de générateurs de qualité, ainsi que de plusieurs bibliothèques de générateurs qui s'adaptent aux matériels de traitement parallèle :

MLFG : ce générateur a été créé par Mascagni et Srinivasan (Mascagni and Srinivasan 2004). C'est un générateur non linéaire qui utilise une formule de Fibonacci décalée (*Multiple Lagged Fibonacci Generator,* d'où son nom) pour générer des suites de nombres pseudo-aléatoires. C'est un cousin intéressant du générateur ALFG (*Additive Lagged-Fibonacci Generator*) car ses propriétés statistiques sont excellentes. La version 64 bits de MLFG dispose d'une période de 2^{124}. Avec cette période, plus d'un mécanisme de paramétrage, MLFG est capable de fournir 2^{51} sous-flux avec une longueur de 2^{72}, ce qui est amplement suffisant pour des machines Exascale et plus.

LFSR113 et LFSR258 : cette famille de générateurs a été créée par L'Ecuyer (L'Ecuyer 1999b). Le générateur LFSR (« *Linear Feedback Shift Register* » en anglais) est basé sur les récurrences linéaires modulo 2 avec un polynôme caractéristique primitif permettant d'accéder à la période optimale. LFSR113 et LFSR258 sont respectivement les versions 32 et 64 bits de LFSR. Des limites statistiques existent pour ces générateurs que nous ne pouvons recommander pour les meilleurs calculateurs classés au Top500 dans l'état actuel des connaissances.

MRG31k3p : L'Ecuyer et Touzin (L'Ecuyer et Touzin 2000) ont proposé un générateur combiné de plusieurs générateurs récursifs de nombres pseudo-aléatoires. Il a été construit de façon à ce que chaque composant s'exécute rapidement et il est facile à mettre en œuvre : les multiplications par une puissance de 2 peuvent être mis en œuvre par un décalage à gauche et quelques opérations supplémentaires suffisent pour la réduction modulaire.

MRG32k3a : ce générateur propose des flottants en 32 bits, il est le résultat de la combinaison de multiples générateurs récursifs avec deux composants. Créé par L'Ecuyer (L'Ecuyer 1999a), l'algorithme a été conçu pour être mis en œuvre directement pour des nombres réels flottants. Sa

structure de données ne nécessite que 6 entiers, ce qui est un atout pour l'architecture GP-GPU. D'autre part, il a été spécifiquement conçu en parallèle avec la suite de test statistique publiée en 2007 (TestU01) (L'Ecuyer et Simard 2007) et présente d'excellentes garanties en termes de propriétés statistiques. Avec sa période de 2^{191}, il est possible de diviser la séquence d'origine en 2^{64} sous-flux, chacun proposant un flux de 2^{76} éléments.

MT19937 : le générateur Mersenne Twister (MT) est une version spécifique de registre à décalage bouclé généralisé (« *generalized feedback shift register* »). Créé par Matsumoto et Nishimura (Matsumoto and Nishimura 1998), il présente de très bonnes propriétés statistiques (même s'il échoue à certains tests de qualité cryptographique). Le statut de ce générateur dépasse les 6 Ko et le rend peu adapté aux accélérateurs de type GP-GPUs. De même, la méthode *JumpAhead* a été complexe à mettre en œuvre comme montrée par les travaux (Haramoto et al. 2008). Il n'est rapide qu'avec des générateurs qui ont un statut (état courant) de très petite taille. Par contre, son implémentation des tirages classiques est particulièrement efficace, jusqu'à 15 fois la performance de MRG32k3a sur des processeurs Intel modernes avec les bonnes options des compilateurs C. Un autre point fort de ce générateur est qu'il est livré avec un exemple des nombres attendus pour vérifier la répétabilité et la portabilité, contrairement à d'autres générateurs fournis par des auteurs qui ne sont pas sensibles aux aspects reproductibilité. D'autre part, sa période immense (2^{19937}) peut se révéler très utile pour des algorithmes utilisant des accélérateurs quantiques. En compagnon du générateur MT, le logiciel Dynamic Creator (Matsumoto et al. 2000), permet de créer différents « petits » générateurs de la famille MT qui peuvent être affectés chacun à un élément de calcul (PE).

Philox-4x32-10 : c'est un des 3 générateurs dans l'article proposé par Salmon et son équipe à la Supercomputing Conference de 2011 aux Etats-Unis (Salmon et al. 2011). Ces générateurs sont inspirés des techniques de cryptographie en dégradant l'aspect de complexité cryptographique pour privilégier la vitesse de tirage de nombres. Philox est un générateur du type « *counter-based* » qui utilise une S-box (fonction Feistel générée en utilisant le *xor* bit-à-bit comme groupe d'opérateur). La S-box utilise la multiplication pour la diffusion, ses performances sont excellentes en terme de rapidité de génération de nombres.

Tyche RNG : c'est un générateur non-linéaire de qualité cryptographique. Il est conçu pour la simulation et a été créé par Neves et Aurjo (Neves and Araujo 2012). Il a un petit état de 128-bits et une période de 2^{127}. Il est toujours rapide pour les architectures actuelles en raison de sa fonction d'itération très simple, dérivée de la technique de cryptage ChaCha. Il se prête particulièrement bien aux environnements hautement parallèles à base de GP-GPUs où il permet à un très grand nombre de flux parallèles non-corrélés fonctionnant indépendamment.

MTGP : c'est la version « Mersenne Twister » adaptée aux processeurs de calculs de type GP-GPU. Il a été créé par Saito et Matsumoto (Saito and Matsumoto 2013). La taille de l'état du générateur est significativement réduite par rapport au générateur initial MT de 19937 bits (son état nécessite 11213 bits) et il utilise le parallélisme élevé des GPUs dans le calcul des nombreuses étapes de récurrence en parallèle. Ce générateur est basé sur une récurrence linéaire et a une très bonne équipartition dans de grandes dimensions. Tout comme le générateur Mersenne Twister, un logiciel de création dynamique permet de créer un très grand nombre de petits générateurs de type MTGP disposant d'une petit empreinte mémoire adapté au calcul sur GP-GPU.

clRNG : est une bibliothèque pour la génération de nombres pseudo-aléatoires uniformes pour OpenCL (*Open Compute Language*) permettant une abstraction pour la production de codes parallèles pour des architectures disposant d'accélérateurs de calculs différents que ce soient des CPUs de type SMP manycore (Intel Xeon Phi), des GP-GPUs et des FPGAs (*Field Programmable Gate Arrays*). Créé récemment par l'équipe de L'Ecuyer (L'Ecuyer et al. 2015b), il fournit de multiple flux qui sont créés sur l'ordinateur hôte et utilisés pour générer des nombres pseudo-aléatoires soit sur l'hôte, soit sur les dispositifs d'accélération des calculs. Dans son approche, L'Ecuyer considère que les flux de nombres pseudo-aléatoires agissant sont comme des générateurs de nombres pseudo-aléatoires virtuels.

En même temps qu'avec des principaux générateurs de qualité présentés ci-dessus, plusieurs bibliothèques de génération parallèles de nombres pseudo-aléatoires ont été proposées. Nous en avons retenu les 4 principales présentées brièvement ci-dessous : SPRNG, RNGStream, PRAND et MKL avec la bibliothèque VSL.

SPRNG : c'est une bibliothèque évolutive qui a été créée par Mascagni et Srinivasan (Mascagni and Srinivasan 2000). Cette bibliothèque est utilisée par de nombreux logiciels de calcul à haute performance et en particulier par des codes de calculs de Monte Carlo. Elle est conçue pour utiliser des générateurs « paramétrables » de nombres pseudo-aléatoires pour fournir plusieurs flux aux éléments de calcul parallèles. Cette bibliothèque contient plusieurs générateurs anciens comme : LCG48, LFG, LCG64, ALFG, MLFG, CMRG, …. Le fleuron de cette bibliothèque est le générateur MLFG sur 64 bits.

PRAND : C'est une bibliothèque des générateurs de nombres pseudo-aléatoires modernes pour CPU et GPU. Elle contient deux implémentations, l'une mono-threadée et l'autre multi-threadée pour respectivement CPU et GPU. La réalisation mono-threadée pour CPU est basée sur l'ensemble des instructions vectorielles SSE (maintenant étendues aux instructions vectorielles AVX). Cette

bibliothèque a été créée par Barash et Shchur (Barash et Shchur 2014). Une des caractéristiques utiles lors de l'utilisation PRAND dans les simulations parallèles est la possibilité d'initialiser jusqu'à 10^{19} flux indépendants avec la méthode de découpage par blocs. En plus, cette bibliothèque propose la technique de « *jump ahead* » pour les générateurs suivants : MT19937, MRG32k3a, LFSR113, GM19, GM31, GM61, GM29, GM55, CQ58.1, CQ58.3, CQ58.4. Les derniers générateurs de cette liste sont moins connus et ont été présentés dans le package RNGSSELIB qui prend en compte les possibilités des instructions vectorielles SSE.

RngStreams : cette bibliothèque fournit une solution souple et viable au problème de la création de flux et de sous-flux de nombres pseudo-aléatoires indépendants pour les simulations dans les environnements de traitement parallèle. Le générateur qui sert de colonne vertébrale (« *backbone* » en anglais) pour RngStreams est MRG32k3a. Une version de cette bibliothèque, d'abord conçue pour les langages C, C++, Java, Fortran (L'Ecuyer et al. 2015a) a été proposée pour R en 2005. Elle est utile pour la parallélisation avec OpenMP et MPI. Cette bibliothèque est très souple à utiliser, le seuil écueil est la performance du générateur qui peut être très pénalisante pour du calcul intensif. Une implémentation optimisée en C de Mrg32ka avec un compilateur Intel peut être 15 fois plus lente que celle en C de Mersenne Twister.

Intel MKL avec la bibliothèque VSL : cette bibliothèque proposée par Intel propose une prise en charge des besoins statistiques au sein de la bibliothèque mathématique du noyau (*Math Kernel Library* - MKL). Elle propose plusieurs générateurs de nombres pseudo-aléatoires : MCG31m1, R250, MCG59, MRG32k3a, MCG59, WH, MT19937, MT2203, SFMT19937 (Intel MKL 2014). Plusieurs sont connus pour être déficients. En effet, L'Ecuyer a notamment indiqué récemment que les trois premiers générateurs échouent à plusieurs tests statistiques de son outil TestU01 et ne doivent pas être utilisés (L'Ecuyer et al. 2015a).

Dans ce même article de L'Ecuyer (L'Ecuyer et al. 2015a), nous trouvons une classification des générateurs et des bibliothèques que nous avons présentée ci-dessus en fonction des deux approches qu'il a retenu et que nous avons discutées précédemment. Il s'agit soit de générer des nombres pseudo-aléatoires plus rapidement grâce aux architectures vectorielles, soit d'utiliser des générateurs en parallèle sur chacun des éléments de calcul. Dans ce cas, il s'agit de disposer d'une petite empreinte mémoire car les architectures parallèles massives modernes ne proposent pas beaucoup de mémoire pour chacun des éléments de calcul. Ainsi, les générateurs disposant de petits états tels que LFSR113, MRG31k3p, MRG32k3a peuvent être implémentés aisément dans un seul thread d'un élément de calcul. Les générateurs de type « *counter-based* » sont également une solution capable de fournir des millions de flux, chacun ayant un petit état (comme Philox-4x32-10,

Tyche RNG). Dans le cas de l'utilisation de plusieurs éléments de calcul pour fournir un seul flux stochastique, on trouve les bibliothèques suivantes : VSL, RNGSSELIB, MTGP, clRNG, PRAND.

III - Conclusion

Dans ce chapitre, nous avons présenté la simulation stochastique parallèle, les techniques de parallélisation d'une simulation stochastique, les techniques de parallélisation des générateurs ainsi que les générateurs qui s'adaptent aux architectures de traitement parallèle. En fonction des contextes propres à chaque application (le modèle de programmation, la technique de parallélisation, le nombre de flux, etc.), ainsi que de la plate-forme matérielle (multicores, manycores de type GPU ou autres, MPPAs, FPGAs, etc.), on choisira le générateur ou la bibliothèque de générateurs de nombres pseudo-aléatoires le plus adapté. Les choix doivent respecter les approches logicielles et le matériel de traitement parallèle. Dans les faits, il y a beaucoup de générateurs de nombres pseudo-aléatoires qui ont été créés pour des calculs séquentiels sur CPU, mais qui s'adaptent très bien au cas du calcul parallèle. On peut notamment citer : MRG32k3a, MLFG et la famille des générateurs de type Mersenne Twister qui sont particulièrement rapides sur les processeurs récents. Ces générateurs passent les meilleurs tests statistiques du moment (en séquentiel et en parallèle). Dans le cas de la recherche de reproductibilité et de répétabilité numérique de simulations stochastique, il existe un ensemble suffisamment important de générateurs sur lesquels nous pourront baser nos travaux.

Chapter 3 - Architecture Parallèles et Modèles De Programmation Parallèle

I - Introduction

Le calcul parallèle prend une place de plus en plus significative dans la recherche scientifique. Il est particulièrement bien adapté pour la modélisation et la simulation des systèmes complexes. Une analyse détaillée des motivations pour l'usage du calcul parallèle en science est disponible dans la littérature, que ce soit pour les applications actuelles ou pour de futures applications (Barney 2015) (Meuer and Gietl 2013). Un rapport européen a aussi discuté le rôle important des applications du calcul parallèle ou de l'utilisation de supercalculateurs pour les sciences et la vie (PRACE 2012). Dans ce chapitre, nous allons passer en revue les bases du calcul parallèle, présenter les nouvelles architectures multi-cœurs et manycores ainsi que les différents modèles de programmation parallèle.

II - Les architectures et modèles de programmation parallèles

II.1 - Concepts et terminologie

Dans cette partie, nous présentons les concepts et la terminologie pour les architectures parallèles. Ces concepts sont très utilisés dans le domaine du calcul à haute performance, ainsi que dans celui des simulations stochastiques parallèle et de la reproductibilité numérique.

Quelques définitions de base

Pour présenter la notion de parallélisation, nous retenons une définition récente de Hager et Wellein (Hager and Wellein 2008) : « la **parallélisation** est le processus de formulation d'un problème d'une manière qui se prête à l'exécution simultanée sur plusieurs « unités d'exécution » d'un certain type ». L'objectif principal de la parallélisation est de réduire le temps d'exécution.

Par ailleurs, Barney (Barney 2015) a donné un sens simple à la notion de calcul parallèle (« *parallel computing* » en anglais) : « le calcul parallèle est l'utilisation simultanée de plusieurs ressources de calcul pour résoudre un problème de calcul ». Il existe des approches séquentielles (on trouve aussi

parfois le terme « sériel ») pour résoudre un problème, et fréquemment on peut trouver des méthodes parallèles pour résoudre le même problème. Pour ce faire il faut :

- Que le problème soit divisible en parties distinctes qui peuvent être résolues simultanément.
- Que chacune de ces parties soit décomposée en une série d'instructions.
- Que les instructions de chaque partie puissent s'exécuter simultanément sur des processeurs différents.
- Qu'un mécanisme global de contrôle puisse être employé pour coordonner les calculs et présenter le résultat final

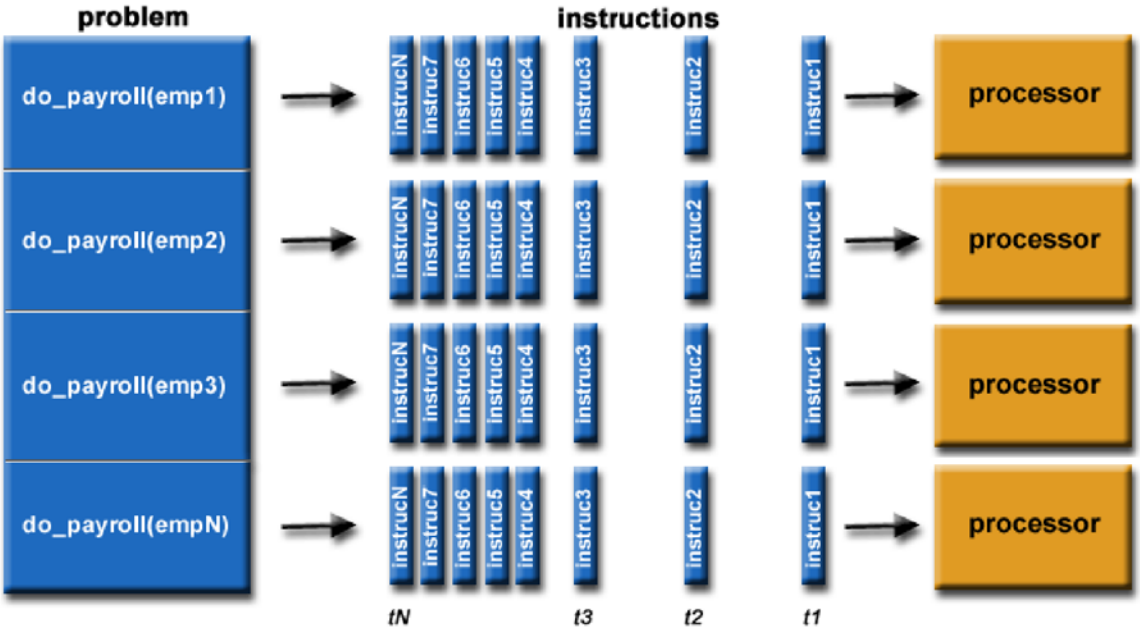


Figure 3.1. Un exemple de calcul parallélisable (Barney 2015).

L'idée principale du calcul parallèle réside donc dans le découpage d'un grand problème en plusieurs petits problèmes élémentaires (cette étape s'appelle la phase de « partitionnement ») que l'on peut résoudre simultanément.

La Figure 3.1 présente deux types de parallélisation, l'un au niveau logiciel et l'autre au niveau matériel. Pour le logiciel, la parallélisation présentée dans la figure 3.1 est celle au niveau de la prise en charge des instructions. Il existe en effet, bien des niveaux de parallélisme comme le bit, les données, les instructions, les tâches, les threads, les transactions ou les jobs systèmes (Malek 1993) (Hennessy and Patterson 2007). La figure 3.1 illustre la prise en charge de la parallélisation par plusieurs processeurs. Les architectures des microprocesseurs modernes permettent différents

types de parallélisation au sein d'un seul micro-processeur. En effet, les techniques de « *pipelining* » des architectures dites « superscalaires », les exécutions parallèles au sein d'un micro-processeur multi-cœurs, ou la parallélisation avec des approches vectorielles au sein même des instructions d'un microprocesseur sont des bons exemples.

Barney a aussi distingué deux types principaux de ressources de calcul : soit un seul ordinateur (ou nœud de calcul) avec plusieurs processeurs/cœurs, soit un nombre arbitraire de nœuds de calcul reliés par un réseau (ou « *InterConnect* », terme anglo-saxon utilisé fréquemment dans le milieu du calcul à haute performance).

Un ordinateur parallèle (« *parallel computer* » en anglais) est composé d'un ensemble de processeurs qui sont capables de travailler en collaboration pour résoudre un problème de calcul (Foster 1995). Selon Foster, cette définition est suffisamment large pour inclure les superordinateurs qui ont des centaines ou milliers de processeurs, les réseaux de stations de travail, les postes de travail multiprocesseurs et les systèmes embarqués qui disposent de multiples cœurs de calcul (c'est maintenant le cas de bon nombre de téléphones mobiles).

Jack Dongarra (Dongarra 2004) a proposé une définition large pour la notion de superordinateur ou supercalculateur. Cette catégorie en perpétuelle évolution désigne la classe des systèmes de calcul les plus performants de la planète. Ces derniers sont classés en fonction de leurs performances pour des tests appelés benchmarks. Le classement le plus connu est le Top500 qui évalue les systèmes suivant leurs performances en calcul pur pour des problèmes d'algèbre linéaire. De nombreux autres tests sont maintenant disponibles pour évaluer la performance des superordinateurs pour d'autres classes d'applications.

Meuer et Gietl (Meuer and Gietl 2013) intègrent l'évolution des technologiques dans leur façon de présenter la notion de superordinateur sur le site en ligne wiseGEEK. Un superordinateur est pour eux défini comme suit : « Un superordinateur est un ordinateur qui exécute les programmes de calcul à une vitesse qui est bien supérieure à celle des autres ordinateurs. Étant donné que le monde de l'informatique est en constante évolution, il ne faut donc pas être surpris d'apprendre que la plupart des superordinateurs ne portent leurs titres superlatifs que pour quelques années, au mieux. Les programmeurs se plaisent à dire que le superordinateur d'aujourd'hui est le poste de travail (« *workstation* » en anglais) de demain ».

Amundsen (Amundsen et al. 2015) propose encore une autre définition : « Un superordinateur est un ordinateur qui domine la majorité des autres systèmes en termes de capacité de traitement – la vitesse du calcul – au moment de son introduction. Il s'agit d'un ordinateur extrêmement rapide

dont la puissance de traitement (« *number-crunching power* » en anglais) se mesure à l'heure actuelle en millions de milliards d'opérations en virgule flottante par seconde (« *Floating Point Operations Per Second* » - flop/s en anglais). Le superordinateur d'aujourd'hui est destiné à devenir l'ordinateur de base de demain. Pour beaucoup, une meilleure définition du superordinateur peut être n'importe quel ordinateur qui est seulement une génération derrière ce dont vous avez vraiment besoin ! ».

Amundsen a aussi présenté la structure de base d'un superordinateur actuel. La majorité des superordinateurs est constituée de nombreux petits ordinateurs (ou nœud de calcul) – parfois des milliers – connectés via les connexions les plus rapides du moment. Ces petits ordinateurs fonctionnent comme une « armée de fourmis » (« *army of ants* » en anglais) pour résoudre très rapidement les calculs difficiles des scientifiques ou des ingénieurs.

La capacité d'un superordinateur est définie par sa vitesse de traitement. Actuellement, l'unité de vitesse d'exécution est le flop/s : Kflop/s = 10^3 flop/s, Mflop/s = 10^6 flop/s, Gflop/s = 10^9 flop/s, Tflop/s = 10^{12} flop/s, Pflop/s = 10^{15} flop/s, Eflop/s = 10^{18} flop/s, Zflop/s = 10^{21} flop/s. Un FLOP représente toute opération mathématique entre des nombres à virgule flottante (addition, soustraction, multiplication, division).

Le site Web http://www.top500.org présente la liste des leaders mondiaux actuels du super-calcul, classés selon le nombre d'opérations en virgule flottante traitées par seconde. Le classement est effectué 2 fois par an. Les superordinateurs sont comparables aux voitures de Formule 1 qui nécessitent d'énormes sommes d'argent et des compétences spécialisées pour les concevoir, bien qu'ils soient des ordinateurs généralistes.

La notion de calcul à haute performance (ou HPC, l'acronyme fréquemment utilisé pour *High Performance Computing* en anglais) a été défini depuis un bon moment par Hey (Hey 1997) pour le JISC (*Joint Information Systems Committee*) du Royaume-Uni d'une manière très compréhensible. Selon cette définition, une machine HPC a pour spécificité de fournir une puissance de calcul supérieure de plus d'un ordre de grandeur à celle des ordinateurs de bureau. Cette définition reflète bien le fait que la notion de HPC est mobile. C'est-à-dire, qu'un système HPC aujourd'hui, se retrouvera un jour dans votre ordinateur de bureau, votre tablette ou votre téléphone.

Computing resources which provide more than an order of magnitude more computing power than is normally available on one's desktop (Hey 1997).

L'institut national pour les sciences du calcul des États-Unis (NICS 2016) a également défini le calcul à haute performance comme une application des supercalculateurs à des problèmes de calcul qui

dépassent la capacité des ordinateurs standards ou demandent un temps de calcul trop long. La définition du NICS oppose l'ordinateur de bureau possédant un seul processeur central aux systèmes composés de plusieurs nœuds de calcul, chaque nœud lui-même composé d'un ou plusieurs processeurs.

"High-Performance Computing," or HPC, is the application of "supercomputers" to computational problems that are either too large for standard computers or would take too long. A desktop computer generally has a single processing chip, commonly called a CPU. A HPC system, on the other hand, is essentially a network of nodes, each of which contains one or more processing chips, as well as its own memory (NICS 2016).

Si la définition d'Hey se limite à la puissance du calcul en comparant un ordinateur de bureau à un système HPC, la définition du NICS prend en compte l'architecture du système HPC considéré comme un réseau des ordinateurs interconnectés. Les définitions s'accordent sur des cas d'utilisation, les problèmes du calcul « trop grands » ou « trop longs » qu'un ordinateur de bureau ne peut réaliser.

Le partenariat pour le calcul avancé en Europe (PRACE 2012) a souligné le rôle essentiel du HPC pour les industriels qui comptent sur sa précision et sa rapidité afin d'aider à l'innovation dans de nombreux secteurs. Selon le rapport de Transtec (Transtec 2013), les ordinateurs à haute performance ont été construits pour résoudre les nombreux problèmes que les ordinateurs classiques ne pouvaient gérer.

On utilise depuis 2008 la notion de calcul hybride lorsque l'environnement de calcul est hétérogène, par exemple lorsque le système de calcul est composé d'un processeur classique CPU et de processeurs de calcul graphiques (GPUs). Il peut aussi s'agir de calculs qui sont effectués avec des composants sur la grille de calcul et d'autres dans le cloud, ou encore lorsque des modèles de programmation différents sont utilisés pour résoudre le problème considéré (par exemple : le modèle de programmation MPI + OpenMP, etc).

Avec l'évolution actuelle des performances, le prochain niveau dans l'échelle des performances à atteindre est dit exaflopique, et on parle couramment en anglais de « l'Exascale ». Une machine exascale est capable d'effectuer 1 milliard de milliards d'opérations en virgule flottante par seconde (Hsu 2014b) soit 10^{18} FLOPS. Nous sommes aujourd'hui dans la phase où les constructeurs conçoivent des machines de ce type tout en respectant une contrainte de 20 MW (megawatt) de consommation électrique. Cette course des constructeurs apporte de nombreuses opportunités et pose aussi de nombreux défis pour la communauté du calcul scientifique (Ashby et al. 2010) (PRACE

2012) (Hsu 2014b). Dans un futur proche, la prochaine génération de supercalculateur visera des performances « zettaflopique » (10^{21} flops) ou « quintiflopique » (10^{30} flops).

Dans la course actuelle, les architectures d'ordinateur suivent encore celle initialement proposée par John Von Neumann (Barney 2015). Selon cette architecture, une machine se définit comme « un ordinateur à programme enregistré en mémoire ». En effet, dans cette architecture, les instructions des programmes ainsi que les données sont enregistrés dans la mémoire dite centrale. Il y a 4 composants principaux dans ce type d'architecture : la mémoire, l'unité de contrôle, l'unité d'arithmétique logique et l'unité de gestion des entrées/sorties. Les unités de contrôle et d'arithmétique logique sont dans un bloc qui s'appelle l'unité centrale de traitement (« *Central Processing Unit* » ou CPU en anglais). Selon Barney, les ordinateurs parallèles actuels suivent encore cette architecture ; les différentes unités ou composants sont multipliés, mais l'architecture fondamentale de base reste la même.

Les lois théoriques fixant les limites du calcul parallèle

A la fin des années 60, Amdahl (Amdahl 1967) a proposé des limites théoriques à la performance des codes parallèles, basées sur le ratio entre le pourcentage de temps passé pour des traitements parallèles et celui pour le traitement séquentiel. L'accélération du traitement de la fraction de code qui peut être traité en parallèle est dépendante du nombre de processeurs ou de cœurs utilisés.

> *…the effort expended on achieving high parallel processing rates is wasted unless it is accompanied by achievement in sequential processing rates of very nearly the same magnitude (Amdahl 1967).*

Avec la loi d'Amdahl, viennent un certain nombre de concepts que nous définirons ci-après. La notion de « *speedup* » ou de facteur d'accélération est présentée comme suit :

$$speedup = \frac{1}{r_s + \frac{r_p}{n}}$$

Equation 3. 1. Le facteur d'accélération ou « speedup » calculé selon la loi d'Amdahl (Amdahl 1967).

Dans cette équation 3.1, r_s représente la part séquentielle dans un programme et r_p la part parallèlisée du code, de telle sorte que $r_s + r_p = 1$; n est le nombre de processeurs ou de cœurs utilisés. Si $r_p = 0$ alors le *speedup* est égal à 1. Par contre, si $r_p = 1$ alors le *speedup* est égal à n. Si le

code du programme est complètement parallélisé (ce n'est que théorique), le speedup est maximum.

Gustafson (Gustafson 1988) a proposé une réévaluation de la loi d'Amdahl. Une des limites qu'il a identifié dans la formulation de la loi d'Amdahl, est que celle-ci est spécifiée pour des problèmes de taille fixe.

Times for vector start-up, program loading, serial bottlenecks, and I/O that make up the components of the run do ***not*** *grow with problem size (Gustafson 1988).*

La loi de « Gustafson-Barsis » s'écrit, en reprenant les notations utilisées précédemment:

$$Speedup(n) = n - r_s\,(n - 1)$$

Equation 3. 2. L'accélération selon la loi de Gustafson-Barsis (Gustafson 1988).

On observe que si la proportion du programme qui s'exécute de façon séquentielle est à 1, alors le speedup est limité à 1. Par contre si ce ratio est à 0, alors le speedup est maximal (= *n).* Nous retrouvons pour ces 2 cas les mêmes conclusions que pour la loi d'Amdahl. Le changement vient du fait que l'équation de la loi de Gustafson-Barsis (souvent nommée loi de Gustafson) passe à l'échelle avec le nombre de processeurs qui sont capables de prendre en compte un jeu de données plus lourd même si la partie séquentielle est fixe.

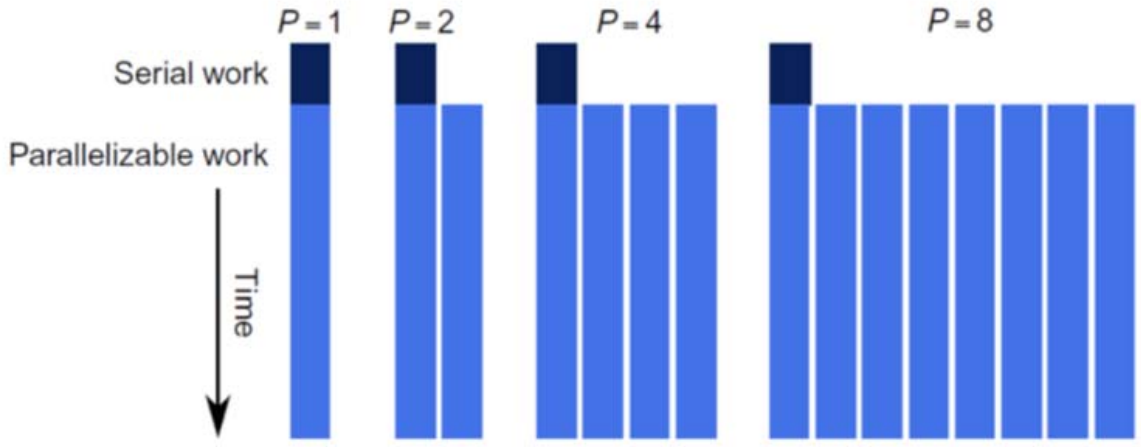


Figure 3.2. Dans la loi de Gustasfon-Barsis, lorsque la taille du problème à traiter augmente avec le nombre de processeurs, alors même que la partie séquentielle reste fixe, le speedup peut continuer à augmenter avec le nombre de processeurs (pages Internet de Barney 2015 et http://www.drdobbs.com/parallel/amdahls-law-vs-gustafson-barsis-law/240162980?pgno=2).

Taxonomie des ordinateurs parallèles

Outre la classification connue de Flynn, nous trouvons sur le site du Lawrence Livermore National Laboratory aux États-Unis, les différentes approches de classification des machines parallèles (Barney 2015). Nous avons identifié 4 approches de classifications possibles :

- ***Classification basée sur les flux d'instructions et de données***: cette classification est la plus répandue, étudiée et proposée par Michael Flynn en 1972. Elle présente 4 types d'approches possibles qui sont rappelées dans la Figure 18.
- ***Classification basée sur la structure interne des ordinateurs***: Wolfgang Handler en 1977 a proposé trois niveaux distincts de cette classification:
- Unité de commande des processeurs (« *Processor Control Unit* » - PCU en anglais).
- Unité d'arithmétique logique (« *Arithmetic and Logic Unit* » - ALU en anglais).
- Circuit au niveau de bit (« *Bit-Level Circuit* » - BLC en anglais).

Handler caractérise un ordinateur par un triplet de couples d'entiers prenant en compte les règles et les opérateurs qui sont utilisés dans la relation entre les différents éléments d'un ordinateur:

$$Computer = (p * p', a * a', b * b')$$

Pour cette définition, l'opérateur « * » est utilisé pour indiquer que les unités sont chaînées ou « pipelinées » avec un flux de données traversant toutes les unités, p est le nombre de PCUs, p' est le nombre de PCUs qui peuvent être pipelinés, a est le nombre d'ALUs contrôlé par chaque PCU, a' est le nombre d'ALUs qui peuvent être pipelinés, b est le nombre de bits dans l'ALU ou l'élément de traitement (PE – processing element en anglais), b' est le nombre des segments de pipeline sur tous les ALUs. Les autres opérateurs « +, v, ~ » sont définis sur le lien https://computing.llnl.gov/tutorials/parallel_comp/parallelClassifications.pdf.

La classification d'Handler est nettement plus encombrante que la classification de Flynn et n'a pas connu de véritable succès. Par contre, une autre proposition de Handler a retenu l'attention des chercheurs. Il s'agit de la distinction entre les machines dites fortement couplées (« *tighly coupled systems* » en anglais) et les systèmes dits faiblement couplés (« *loosely coupled system* » en anglais).

- ***Classification basée sur la façon d'accéder à la mémoire.*** Avec cette classification, nous avons deux types de systèmes différents : (i) les systèmes à mémoire partagée (« *shared memory system* » en anglais) fortement couplés comme ceux qui proposent un accès mémoire uniforme (UMA : *Uniform Memory Access*), ou ceux qui proposent un modèle d'accès non-uniforme à la mémoire (NUMA : *Non-Uniform Memory Acces*), ou encore les

modèles assez répandus avec une cohérence des caches pour les accès non uniformes (ccNUMA : *cache-consistent Non-Uniform Memory Access*); (ii) les systèmes à mémoire distribuée qui possèdent un mécanisme de transmission des messages dans le réseau d'interconnexion lorsqu' on souhaite accéder à une donnée qui n'est pas sur le nœud de calcul courant.

- ***Classification basée sur la granularité du parallélisme***: L'idée de cette classification est basée sur la reconnaissance du niveau de parallélisme dans un programme destiné à être exécuté sur un système multi-processeurs. Il s'agit d'identifier si le parallélisme va travailler au niveau des tâches ou de sous-tâches (fonctions), ou encore au niveau des instructions. La parallélisation va dépendre de plusieurs facteurs comme le graphe d'exécution du programme, le nombre et le type de processeurs disponibles, l'organisation de mémoire ou la dépendance de données.

Nous détaillons ci-après un peu de la classification de Flynn, la plus utilisée en expliquant chaque catégorie de machines suivant les types identifiés par Flynn.

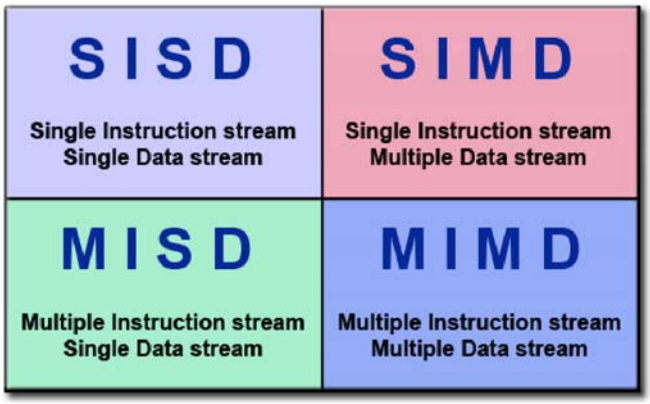


Figure 3.3. Classification des ordinateurs proposée par Flynn (image provenant des pages Internet de Barney 2015).

Le SISD (*Single Instruction Single Data*) est l'architecture d'une machine séquentielle : un seul flux d'instruction et un seul flux de données sont actionnés pendant un cycle d'horloge. Il suit l'architecture classique d'un ordinateur tel que décrite par John Von Neumann. Dans l'histoire des ordinateurs, on peut retracer des machines de ce type comme par exemple : l'UNIVAC1, l'IBM 360, le PDP1, les anciens PC de bureau ou portables avec des processeurs ne disposant que d'un seul cœur et pas d'unités d'hyper-threading (avant 2003).

Les machines de type MISD (*Multiple Instruction Single Data*) sont présentes dans la classification de Flynn. Cependant, ce type de machine n'a pas dépassé le stade de prototypes de laboratoire. Son intérêt n'est plus que limité et il n'y a pas de machines de ce type disponible dans le commerce.

Les deux autres catégories sont utilisées pour résoudre des problèmes complexes en utilisant le calcul parallèle et/ou vectoriel. Les machines de type SIMD (*Single Instruction Multiple Data*) utilisent une approche vectorielle qui est une forme de calcul parallèle. Toutes les unités de traitements exécutent la même instruction à chaque cycle d'horloge, mais chacune fonctionne sur des données différentes. Cette catégorie de machines a été développée avec deux approches, celle des tableaux de processeurs (« *processor arrays* » en anglais) et celle des machines vectorielles utilisant des pipelines. La plupart des ordinateurs modernes, et pour ainsi dire la majorité des micro-processeurs modernes, en particulier les processeurs graphiques utilisent l'approche vectorielle.

Les architectures dites MIMD (*Multiple Instruction Multiple Data*) sont vraiment l'archétype des ordinateurs parallèles. Dans cette catégorie, les ordinateurs peuvent exécuter des flux d'instructions différents; chacun travaille avec un flux de données différent. Actuellement, la plupart des systèmes parallèles modernes sont de ce type. Dans cette catégorie, on distingue les ordinateurs multiprocesseurs à mémoire partagée dits SMP (pour « *Symmetrical Multi-Processing* » en anglais) et les machines à mémoire distribuées.

Même s'il existe 4 types différents dans cette classification de Flynn, nous rappelons que seules les architectures SIMD et MIMD sont plébiscitées et les processeurs modernes peuvent être considérés comme des hybrides car ils utilisent en fait les deux approches vectorielles et parallèles.

Analyse de la performance parallèle

Dans cette rubrique, nous présentons quelques éléments permettant l'analyse de la performance des machines parallèles. L'évaluation de la performance des machines avec des programmes parallèles adaptés est très importante pour déterminer les meilleurs compromis d'architecture en fonction des algorithmes. Il existe de nombreux métriques ; Foster (Foster 1995) a proposé 3 métriques de base : le temps d'exécution, l'efficacité et l'accélération ou « *speedup* » (en anglais). Gramma (Gramma et al. 2003) a ajouté 2 autres métriques élémentaires à la proposition de Foster pour prendre en compte le coût des machines et les surcoûts engendrés par une solution parallèle. Nous présentons ci-après ces 5 critères élémentaires :

Le temps d'exécution: Foster précise que le temps d'exécution total est la somme des 3 temps suivants: le temps de calcul, le temps passé en communications et en entrées-sorties, et le temps d'inactivé (en attente de synchronisation par exemple).

- Le temps de calcul est le temps pour effectuer le calcul proprement dit. Il dépend normalement de la taille du problème, du nombre de tâches et du nombre de processeurs

pour traiter ces tâches. Dans un système parallèle hétérogène, le temps de calcul peut varier en fonction des différentes performances des processeurs. Il dépend aussi des caractéristiques de performance d'accès à la mémoire par les processeurs.

- Le temps de communication est pour un programme parallèle le temps d'envoi et de réception des messages. Il y a deux types distincts de communication, les communications « inter-processeurs » et « intra-processeur ». Le temps de communication global peut, suivant les applications, intégrer les coûts d'entrées-sorties.
- Le temps d'inactivé: c'est un composant du temps d'exécution global, néanmoins, il n'est pas facile à déterminer. Il dépend en effet de l'ordre d'exécution des opérations. Un processeur peut être inactif en raison de l'absence de calcul ou de l'indisponibilité des données à traiter ou encore à cause des attentes nécessaires à la synchronisation et aux communications des différents processus parallèles.

Selon l'idée de Gramma (Gramma et al. 2003), le temps d'exécution global correspond au temps écoulé depuis le moment où un calcul parallèle a commencé, jusqu'à l'instant où le dernier élément a été traité.

Le total des surcoûts parallèles: Gramma a défini le total des surcoûts (« *total overhead* » en anglais) d'un système parallèle comme le total du temps passé par tous les éléments de traitement par rapport au temps requis par l'algorithme séquentiel le plus rapide connu pour résoudre le même problème sur un seul élément de traitement. En effet, tout ce qui concerne la synchronisation d'une activité parallèle, tel que le lancement de plusieurs tâches, et la phase de réduction pour collecter les résultats parallèles et fournir le résultat final, constitue un surcoût de la solution parallèle.

Gramma a aussi donné une formule simple pour calculer cette métrique.

$$T_o = pT_p - T_s$$

Equation 3. 3. Formule permettant d'obtenir le surcoût parallèle ou temps d'« *overhead* » (Gramma et al. 2003).

Dans l'Equation 3.3, T_o représente le temps d'« overhead », T_s est le temps d'exécution en séquentiel, T_p le temps d'exécution d'un seul élément par le programme parallèle, p le nombre d'éléments à traiter et pT_p le total du temps passé à traiter tous les éléments.

L'efficacité: l'efficacité est la fraction du temps pendant laquelle les processeurs parallèles font un travail utile. Foster (Foster 1995) a proposé la formule donnée par l'équation 4 pour évaluer cette métrique.

$$E_{relative} = \frac{T_1}{PT_P}$$

Equation 3. 4. L'efficacité relative de la performance parallèle (Foster 1995).

Dans cette Equation 3.4, T_1 est le temps d'exécution sur un processeur et T_P est le temps d'exécution avec P processeurs, *P* est le nombre de processeurs utilisés. L'efficacité relative donne un taux. Si ce taux est à 1, on a 100% d'efficacité, ce qui est rare car nous nous avons vu qu'il existe toujours un surcoût pour réaliser un traitement parallèle. Cette efficacité relative est liée au *speedup*. Elle s'exprime en rapportant le temps d'exécution en séquentiel au temps d'exécution en parallèle par *P* processeurs. Pour ne pas amplifier la performance parallèle, il faudrait comparer la performance du meilleur algorithme séquentiel (qui souvent ne peut pas être parallélisé) avec celui du meilleur algorithme parallèle.

Gramma (Gramma et al. 2003) a aussi proposé une formule pour calculer l'efficacité (voir Equation 3.5). Dans cette formule, *S* représente l'accélération (ou « speedup »), que nous présentons ci-après et *P* le nombre de processeurs.

$$E = \frac{S}{P}$$

Equation 3. 5. L'efficacité selon Gramma (Gramma et al. 2003).

Les équations de Foster et de Gramma sont équivalentes.

L'accélération: l'accélération ou « speedup » est une mesure très utilisée pour présenter l'efficacité d'un programme (notamment sur la fraction de code parallélisée) par rapport à une exécution séquentielle. Cette métrique est souvent préférée à l'efficacité car elle donne une mesure plus optimiste (certains diront commerciale) de la même réalité. Il s'agit juste d'une autre façon de présenter l'efficacité d'un ordinateur parallèle. D'autres lois existent plus marginalement dans le domaine de calcul à haute performance, avec notamment une loi qui spécifie que la performance des systèmes croît avec la racine carrée du coût de système ; une autre précise que la performance des programmes communicants est liée au log en base 2 du nombre de processeurs.

Foster (Foster 1995) a aussi donné une formule directement reliée avec l'efficacité relative que nous avons décrite dans l'équation 3.4 pour cette métrique :

$$S_{relative} = P.E_{relative}$$

Equation 3. 6. Définition de l'accélération de la performance selon Foster (Foster 1995).

Avec $S_{relative}$ est l'accélération relative, E est l'efficacité relative et P le nombre de processeurs.

Le coût: le coût est lié au temps d'exécution tel qu'il a été défini par Gramma (Gramma et al. 2003), et il est fonction du nombre d'éléments de traitement utilisés et du coût global du système. On peut

avoir d'excellentes performances avec des coûts systèmes exceptionnels, ou accepter d'avoir des temps d'exécution plus long mais un budget maîtrisé.

Toutes les métriques ci-dessus sont construites en mettant en relation les temps de calcul en séquentiel et en parallèle ainsi que le nombre de processeurs ou d'éléments de traitement.

Les tests de référence pour la performance

A côté des indicateurs qui nous permettent d'analyser la performance parallèle, il est aussi nécessaire d'évaluer le profil des performances d'un système de calcul. Comme nous l'avons présenté dans la partie dédiée aux notions utilisées dans le domaine, les supercalculateurs internationaux sont évalués bi-annuellement par la batterie de tests HPL (*High Performance LinPack*) qui aboutit à un chiffre unique exprimé en nombre d'opérations flottantes par secondes (FLOPS – Floating Operations Per Seconds). Cette mesure calculée en FLOPs théorique est utile pour une seule catégorie d'applications (algèbre linéaire) et elle est de plus en plus critiquée car elle ne constitue pas une mesure fiable pour bon nombres d'applications réelles qui ne reposent pas sur l'algèbre linéaire.

Dans le tableau 3.1, nous présentons brièvement quelques suites de tests de référence (« *benchmark* » en anglais) conçus pour évaluer divers aspects du calcul à haute performance. Nous présentons notamment la suite de tests de performance pour les systèmes de calcul « pétaflopiques » qui s'appelle la « *HPC Challenge Benchmark Suite* » (ou HPC CBS). Elle se compose des 7 tests de performance suivants: HPL, STREAM, RandomAccess, PTRANS, FFT, DGEMM et b_eff (Luszczek et al. 2006). Tous ces tests opèrent sur des matrices ou sur des vecteurs.

Benchmark	Description	Opération mathématique	Comptage du nombre d'opérations	Résidus mis à l'échelle
HPL	Le HPL est le « *High Performance LINPACK* ». Il a été introduit par Jack Dongarra (Meuer and Gietl 2013) (Luszczek et al. 2006). Il s'agit d'une implémentation du Linpack TPP (*Toward Peak Performance*). Il est basé sur la résolution de systèmes d'équations linéaires par une	Ax = b Avec $A \in R^{nxn}$; $x, b \in R^{n}$	$\frac{2n^3}{3} + \frac{3n^2}{2}$	$\frac{\| Ax - b \|_\infty}{\varepsilon \| A \|_1 n}$ $\frac{\| Ax - b \|_\infty}{\varepsilon \| A \|_1 \| x \|_1}$ et $\frac{\| Ax - b \|_\infty}{\varepsilon \| A \|_\infty \| x \|_\infty}$

	méthode d'élimination Gaussienne avec un pivot partiel.			
DGEMM	Le DGEMM représente la performance pour la multiplication de matrices en double précision (« *Double precision General Matrix-Matrix multiplication* » en anglais).	$C \leftarrow \beta C + \alpha AB$ Avec : A, B, C $\in R^{nxn}$, $\alpha, \beta \in R^n$	$2n^3$	$\frac{\mathrm{V}\, C - \hat{\mathrm{C}}\, \mathrm{V}}{\varepsilon n\, \mathrm{V}\, C\, \mathrm{V}_F}$ $\hat{\mathrm{C}}$ est un résultat de référence pour la multiplication de matrices.
STREAM	Il s'agit d'un programme simple d'évaluation de la performance de la bande passante mémoire durable avec un taux de calcul pour 4 types d'opérations vectorielles (dont ce qui correspond aux instructions FMA – Fused Multiple Add).	COPY: c ←a SCALE: b ← αc ADD: c ← a + b TRIAD: a ← b + αc Avec: a, b, c $\in R^m$ $\alpha \in R$	2m or 3m (en fonction de l'opération)	$\mathrm{V}\, x - \hat{x}\, \mathrm{V}$
PTRANS	Le PTRANS représente l'évaluation de la performance de transposition de matrices en parallèle (« *Parallel Matrix Transpose* » en anglais). Ce test mesure également le taux de transfert de données.	$A \leftarrow A^T + B$ Avec : A, B $\in R^{nxn}$	n^2	$\frac{\mathrm{V}\, A - \hat{\mathrm{A}}\, \mathrm{V}}{\varepsilon n}$
RandomAccess	Il mesure le taux de performance de mises à jour de variables à des emplacements de mémoire aléatoires.	$x \leftarrow f(x)$ $f: x \rightarrow (x \oplus a_i)$ Avec : a_i – flux de nombres pseudo-aléatoires $f: Z^m \rightarrow Z^m$; $x \in Z^m$	4m	La procédure permet d'évaluer les performances pour des mises à jour de mémoire simultanées sur des architectures à mémoire partagée.

FFT	Ce test évalue la performance pour le calcul de transformées de Fourrier rapides (« *Fast Fourier Transform* » en anglais). Il s'agit d'une mesure du taux d'opérations en virgule flottante pour l'exécution de transformées de Fourrier discrètes unidimensionnelles complexes en double précision pour taille m.	$Z_k \leftarrow \sum_j^m z_j e^{-2\pi i \frac{jk}{m}}$ $1 \leq k \leq m$ Avec : z, Z ∈ C^m	$5m\log_2 m$	$\frac{\mathrm{V}\, x - \hat{x}\, \mathrm{V}}{\varepsilon log m}$ $\hat{x}$ est le résultat
b_eff	On utilise ce test pour mesurer la bande passante et la latence des communications (évaluation du réseau d'interconnexions). Il représente la notion de bande passante efficace (« *effective bandwidth benchmark* » en anglais).	On a 8 octets (pour la latence) et 2000000 octets (pour la bande passante)		La communication est mise en œuvre avec le standard MPI synchrone. (envois et réceptions bloquants).

Table 3.1. Tests de performance de la « HPC Challenge Benchmark Suite » (Luszczek et al. 2006).

Récemment, Dongarra (Dongarra et al. 2015) a proposé une nouvelle métrique pour évaluer les performances des systèmes à haute performance. Il s'agit du HPCG (*High Performance Conjugate Gradient*). Cette méthode se base sur le calcul d'un gradient conjugué avec une pré-condition de Gaussien-Seidel symétrique. Ce test est disponible sur le site web http://hpcg-benchmark.org/. En comparaison avec la suite de test *Challenge Benchmark suite*, le HPCG évalue bien les aspects multidimensionnels, mais il ne considère qu'un seul new facteur de tester la performance du système et de classer la performance du système.

Après avoir présenté l'ensemble des concepts utilisés dans le domaine du calcul à haute performance et employés par la suite dans ce manuscrit, nous allons examiner des perspectives des nouvelles architectures à court, moyen et plus long terme.

II.2 - Vers de nouvelles approches pour le calcul à haute performance

Dans cette section, nous présentons les nouvelles approches pour les systèmes de calcul à haute performance. En effet, les fuites de courants et les surchauffes associées contraignent la capacité à obtenir de meilleures performances en augmentant la fréquence et le nombre de transistors. L'exploration de nouvelles approches de calcul a démarré en 2003 et elle est marquée par une orientation de plus en plus forte vers l'introduction du parallélisme au sein des micro-processeurs par l'introduction commerciale de la technologie d'hyper-threading qui a conduit aux architectures « multicores » et « manycores » actuelles. Nous parlerons aussi des architectures d'accélération de calcul qui en sont inspirées mais reposant sur l'évolution des unités de traitement graphique qui sont maintenant utilisées comme accélérateurs de calcul. Nous aborderons également les puces dites neuro-morphiques qui s'inspirent du cerveau humain et qui commencent à être disponibles. Enfin, nous aborderons le cas des accélérateurs quantiques qui sont étudiés en laboratoires, dont certains sont disponibles commercialement depuis quelques années.

Nous détaillons d'abord les raisons qui ont conduit aux évolutions architecturales actuelles : Vajda (Vajda 2011) a identifié plusieurs raisons pour lesquelles les gains obtenus dans le passé par l'augmentation de la fréquence d'horloge ne peuvent plus être observés. Il a évoqué notamment le problème de la consommation énergétique induit par des fréquences d'horloge plus élevées qui conduit à faire fondre des composants. D'autre part, les retards induits dans les « câbles » ou toutes connexions internes deviennent une question dominante pour chaque cycle d'horloge.

Allalen (Allalen 2015) évoque des causes similaires à celles de Vajda qui conduisent à la nécessité de concevoir des « coprocesseurs » ou « accélérateurs » pour les systèmes de calcul à haute performance. Selon Allalen, ces nouvelles technologies visent à augmenter la performance du calcul tout en minimisant la consommation d'énergie. Il a également proposé un système hétérogène incluant des CPUs et des accélérateurs incluant des instructions vectorielles.

Architecture de technologie Hyper-threading

La technologie dite de l'Hyper-threading a été introduite par Intel. Elle permet l'exécution simultanée de threads au sein des microprocesseurs (« *Simultaneous Multithreading Technology* » - SMT en anglais). Vue du système d'exploitation, la technologie de l'Hyper-threading crée en fait des processeurs logiques pour aider à accélérer la capacité de calcul et de traitement des processeurs physiques. Dans (Marr et al. 2002), nous trouvons une présentation claire du fonctionnement de cette technologie. Le microprocesseur gère des copies de l'état de l'architecture pour chaque processeur logique. Ces processeurs logiques de support des threads partagent un seul ensemble de

ressources d'exécution physiques. Le processeur logique partage presque toutes les autres ressources du processeur physique comme le cache, les unités d'exécution, la prédiction de branchement ainsi que la logique de contrôle des bus.

Hyper-Threading technology makes a single physical processor appear as multiple logical processors. To do this, there is one copy of the architecture state for each logical processor, and the logical processors share a single set of physical execution resources. From a software or architecture perspective, this means operating systems and user programs can schedule processes or threads to logical processors as they would on conventional physical processors in a multiprocessor system. From a microarchitecture perspective, this means that instructions from logical processors will persist and execute simulateously on shared execution resources (Marr et al. 2002).

En fait, un processeur logique est une copie de l'état de l'architecture d'un processeur physique. On peut voir que cette technologie crée des processeurs logiques qui reposent sur un processeur physique. On trouve assez fréquemment deux processeurs « logiques » sur un processeur physique (quatre sur les dernières générations). Le processeur logique a son propre état indépendant de l'architecture qui contient les registres à usage général, ceux de contrôle, ceux du contrôleur d'interruption et les registres d'état de la machine. La technologie d'Hyper-threading fournit donc un parallélisme donnant un support matériel aux threads logiciels.

Architectures multi-cœurs et manycores

La technologie de l'Hyper-threading est actuellement fonctionnelle, l'idée sous-jacente a cependant évolué de façon à ne pas proposer seulement plusieurs threads au sein d'un seul processeur, mais plusieurs processeurs complets au sein d'une puce « CPU ». On introduit la notion de cœurs de calcul au sein d'un seul CPU et les processeurs sont appelés multi-cœurs (multicore en anglais). Barbic (Barbic 2007) a proposé des clés de compréhension des architectures multi-cœurs à partir des éléments de base d'un ordinateur monoprocesseur classique. Une puce complète se compose des éléments de base suivants: l'interface avec le bus externe, le bus système interne, et un cœur (incluant les registres et l'unité arithmétique et logique). Un cœur est vraiment une unité de traitement autonome complète. L'architecture multi-cœurs est une réplication de plusieurs cœurs sur une seule puce CPU, l'ensemble des cœurs partageant le bus interne et l'interface.

Diaz (Diaz et al. 2012) a également présenté l'architecture de multi-cœurs comme étant simplement l'intégration de plusieurs cœurs (actuellement entre deux et seize cœurs) dans un seul microprocesseur. Ce type d'architecture n'est pas seulement utilisé sur les serveurs de calcul, mais on la retrouve depuis des années dans les ordinateurs de bureau, les ordinateurs portables et maintenant depuis quelques années dans les téléphones portables. Vajda (Vajda 2011) a souligné

l'homogénéité de l'architecture multi-cœurs, y compris au niveau des performances ; les cœurs sont capables d'exécuter les mêmes binaires. En d'autres mots, nous sommes en présence d'une architecture classique de type MIMD à mémoire partagée que l'on appelle SMP (*Symetric Multi-Processor*). De ce fait, avec le terme « multi-cœurs », on voit parfois apparaître le terme « multiprocesseurs » qui peut être utilisé de façon logiquement équivalente.

En ce qui concerne les processeurs *manycores*, Diaz (Diaz et al. 2012) a montré que ces architectures se caractérisent par un changement d'échelle avec un grand nombre de cœurs (de quelques dizaines à plusieurs centaines). La différence entre « multi-cœurs » et « manycores »[1] dans l'idée de Diaz réside dans le nombre de cœurs. Clairement, la définition de ces notions dans ce cas-là n'est pas complète mais c'est une bonne distinction. En réalité, ces architectures diffèrent non seulement en nombre de cœurs, mais aussi par l'architecture.

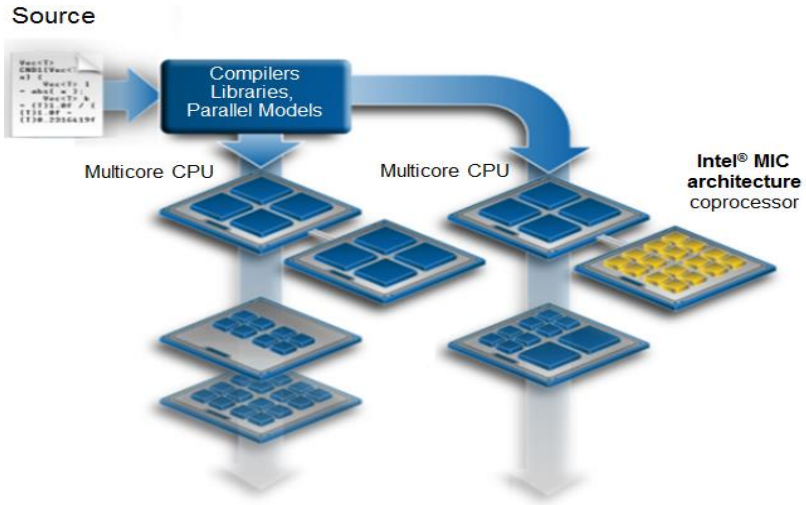


Figure 3.4. Les modèles de programmations, les langages et les bibliothèques communes entre processeurs multi-cœurs et manycores (Reinders 2015).

Reinders (Reinders 2015) a présenté la vision d'Intel qui est d'étendre l'architecture multi-cœurs à l'architecture manycores avec des modèles cohérents de programmation, de langages et de bibliothèques (Figure 3.4). La programmation de processeurs MIC (*Many Integrated Cores* - Intel Xeon Phi) est la même que sur des processeurs Intel Xeon multi-cœurs avec cependant un plus haut degré d'exploitation du parallélisme. On peut utiliser cette figure pour distinguer visuellement les différences entre les architectures multi-cœurs et manycores. Avec l'architecture multi-cœurs, l'utilisateur dispose de 2 à 16 cœurs physiques mais avec une architecture manycores de type Intel dans un système hétérogène, on pourra utiliser jusqu'à 61 cœurs physiques.

1 A cause de la difficulté de traduction du terme *manycore* en français, nous utiliserons ce terme anglais dans la suite de ce manuscrit.

Nous avons introduit dans ce dernier paragraphe la notion de cœur physique. Elle rappelle la notion de cœur logique qui est aujourd'hui supportée par l'Hyper-threading. En effet, les systèmes d'exploitation voient les threads comme des processeurs. Or, il y a une différence (très significative en matière de performance) entre processeurs logiques et physiques. Une puce « CPU » moderne est un multiprocesseur « physique » avec plusieurs cœurs de calcul physique répliquant les unités arithmétiques et logiques ainsi que de la mémoire cache locale. Mais chaque cœur physique propose plusieurs threads matériels (dupliquant uniquement l'état de l'architecture d'un cœur) : ces threads matériels sont les cœurs logiques (processeurs logique) et la puce complète est vue, par la majorité des systèmes d'exploitation, comme un ensemble de N cœurs logiques que le système appelle des processeurs. Cette information imprécise trompe beaucoup d'ingénieurs informaticiens et d'administrateurs systèmes qui n'ont pas la connaissance des architectures de micro-processeurs. Ceci entraîne souvent des déceptions quand on attend des performances linéaires par rapport à un nombre de processeurs logiques. A titre d'exemple, une puce « CPU » de serveur Intel Xeon proposant 16 cœurs physiques, avec chaque cœur physique proposant 2 threads matériels (2 cœurs logiques pour chaque cœur physique) est vue par le système d'exploitation comme disposant de 32 processeurs. En matière de calcul à haute performance, il est important de se rappeler que ce ne sont que des processeurs logiques.

GP-GPU

Le concept de GP-GPU (*General Purpose – Graphical Processing Unit*) vient de l'utilisation d'une unité de traitement graphique (GPU) pour un usage général et plus particulièrement pour réaliser des calculs scientifiques. Une Unité de Traitement Grahique est un dispositif spécialisé conçu pour manipuler rapidement de grandes quantités de pixels graphiques de façon vectorielle. Pour faciliter les affichages graphiques en 3 dimensions, les GPUs sont devenus très efficace dans le domaine du calcul en virgule flottante. Un modèle du calcul hybride consiste à utiliser ensemble un CPU et une ou plusieurs GPUs dans une approche de calcul de co-traitement hétérogène. La partie séquentielle de l'application fonctionne en général sur le CPU et la partie du calcul plus vectorielle qui peut être accélérée significativement est déportée sur une ou plusieurs GPUs. Gray (Gray et al. 2013) montre que l'unité de traitement graphique travaille sur des « flux » du calcul. Une approche de calcul hybride efficace utilise à la fois le parallélisme des CPUs multi-cœurs et tous les éléments de calcul d'une GPU. Dans une présentation commerciale, les sociétés de fabrication des GPUs ont présenté un nombre impressionnant de cœurs de calcul GPUs qui ne correspondaient pas à la notion de cœur physique sur les processeurs classiques. De même, les accélérations affichées étaient elles aussi présentées dans une optique commerciale. En effet, ils comparaient l'ensemble des « cœurs » de calcul d'une GPU, contre un seul cœur physique de CPU. Une comparaison honnête doit mesurer la

performance de l'ensemble des cœurs logiques de calculs d'un processeur moderne avec l'ensemble des cœurs GPU.

L'utilisation des architectures de calcul hybrides répartissant les calculs sur des processeurs à architecture x86 classique et sur des cœurs GPU pose des questions importantes en termes de reproductibilité numériques comme discuté dans (Taüfer 2014) et (Hill 2015b).

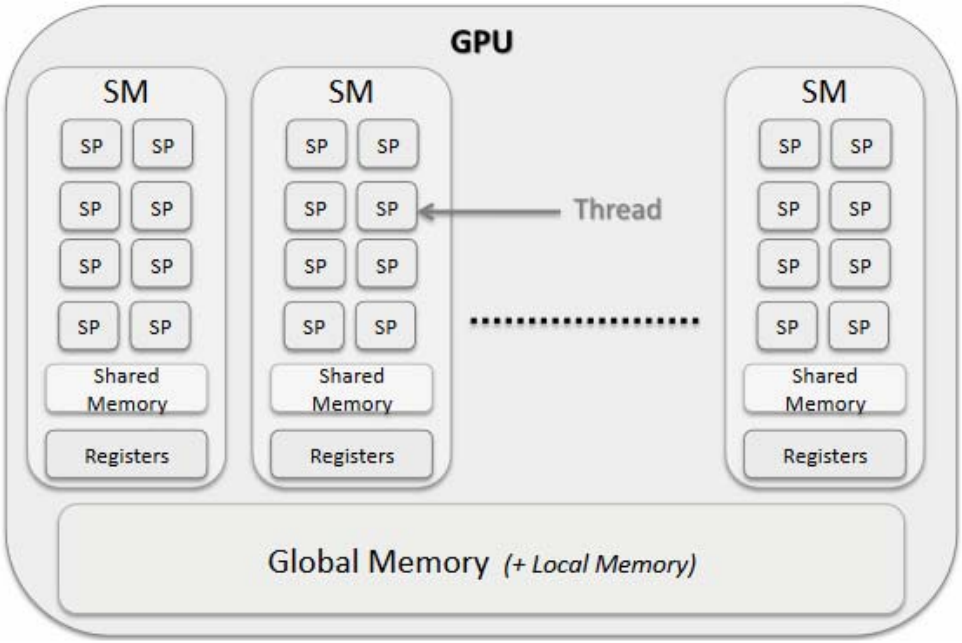


Figure 3.5. Une présentation simple des éléments principaux d'une GPU (Passerat-Palmbach et al. 2012)

La figure 3.5 ci-dessus (Passerat-Palmbach et al. 2012 présente une architecture classique de GP-GPU qui propose un certain nombre de processeurs symétriques (SM – Symmetric Multiprocessor) que l'on peut rapprocher des processeurs vectoriels. En effet, chaque processeur symétrique propose une multiplicité d'unités arithmétiques et logique (notés SP ou *Stream Processor* dans la figure 3.5) qui servent à traiter la même instruction sur plusieurs données différentes suivant l'approche SIMD présentée plus haut. Un processeur GP-GPU combine l'approche MIMD avec plusieurs SM, d'une façon plus massive que l'approche vectorielle avec un grand nombre d'unités arithmétiques et logiques SP.

Jeong et Abram (Jeong and Abram 2016) présentent les GP-GPU comme des unités de traitement graphique servant à « croquer » des masses de données.

L'architecture abstraite des GPUs est présentée par Gray (Gray 2013) comme une grille de blocs de threads selon l'approche préconisée par la société NVIDIA dans son architecture CUDA (*Compute Unified Device Architecture*).

Les Réseaux de portes programmables (FPGA)

Les FPGA (*Field Programmable Gate Array*) ont également été considérés comme des éléments potentiels d'un système de calcul à haute performance (Passerat et Hill 2014). Les FPGA sont utilisés dans certaines des meilleures architectures (comme celle proposée par Cray Inc.) (Craven et Athanas 2007). De nos jours, la communauté du calcul à hautes performances a tendance à privilégier les architectures hybrides à base de GPU pour accélérer les applications, mais selon (Williams et al. 2008), les architectures FPGA offrent de meilleures performances que les GPU face à certaines opérations arithmétiques entières. En plus de ces performances impressionnantes pour certains domaines applicatifs, les approches à base de FPGA ont de nombreux avantages, y compris une faible empreinte environnementale avec des coûts énergétiques très faibles. La plus grande partie de la logique FPGA est configurable, et cette architecture permet d'économiser de l'énergie car la puissance de traitement est limitée par conception aux caractéristiques spécifiques qui seront utilisées. S'il est possible de réutiliser le même silicium pour différentes fonctions, nous avons également l'avantage d'une cartographie facile des différentes opérations permettant ainsi un parallélisme massif. Bien avant le boom du GP-GPU, en 2004, des entreprises se sont jointes à la FPGA High Performance Computing Alliance (FHPCA). Cette alliance est dédiée à l'utilisation des FPGA Xilink pour les applications industrielles haut de gamme du monde réel. Depuis l'avènement du Virtex 4 (et maintenant de dispositifs beaucoup plus grands et puissants), l'utilisation de FPGA pour l'informatique numérique à usage général est devenue une réalité. Même si cela ne se remarque pas encore vraiment dans le Top500, dès 2007, le centre de calcul de l'Université d'Edimbourg a proposé un supercalculateur à base de FPGA nommé "Maxwell".

Avec le genre d'avantages que nous avons décrit précédemment (Passerat et Hill 2014), la question qui se pose est pourquoi n'y a-t-il pas de FPGAs dans le calcul à haute performance ? Leur usage est méconnu et limité pour une série de raisons. Une de ces raisons, est que l'on conçoit une architecture qui n'est absolument pas généraliste et de ce fait, la diffusion restera très limitée. Une autre raison est la limite imposée par les communications entre les processeurs classiques et les FPGA, elle passe actuellement par des bus PCI Express qui restent des goulets d'étranglement sur les architectures à base de processeurs x86. Une autre raison majeure vient du fait que le modèle de programmation commence à peine à être mature. Les développeurs de logiciels dans le monde du calcul à haute performance ne savent pas programmer dans les langages de description matérielle (HDL, tels que VHDL). Le problème ne se limite pas à l'apprentissage d'un nouveau langage car c'est

une approche complètement différente. En effet, nous ne concevons pas des circuits comme nous concevons des programmes. Même si des bibliothèques prédéfinies existantes sont disponibles, cela ne compense pas le fait qu'il n'existe pas de matériel standard réel. Depuis son lancement en juin 2008, la norme OpenCL™ (*Open Computing Language*) a été spécifiquement conçue pour répondre au défi d'exécuter des applications de calcul à haute performance sur des systèmes hétérogènes composés de processeurs multi-cœurs, de GP-GPU et de FPGA. OpenCL est basée sur le langage de programmation C, mais il fournit des fonctionnalités supplémentaires liées à la nature hautement parallèle des FPGA et d'autres périphériques parallèles. OpenCL permet au programmeur de spécifier explicitement et de contrôler le parallélisme. Bien que de nombreux fournisseurs d'accélérateurs aient commencé à prendre en charge OpenCL pour les processeurs et les GPUs, moins d'efforts ont été fournis pour supporter les systèmes hybrides à base de FPGA. De récentes avancées (Altera Corporation) nous permettent de croire que le modèle de programmation OpenCL proposera des solutions avantageuses aux problèmes communs qui ont précédemment empêché l'adoption des FPGA dans le domaine du calcul à haute performance. Une autre porte s'ouvre pour les FPGA avec la nouvelle architecture Power 8+ d'IBM et les prochains processeurs de type IBM Power 9. IBM a en effet fait un choix majeur en permettant à ses nouveaux microprocesseurs de communiquer directement avec des FPGA ou des GP-GPU sans passer par les bus PCI Express.

Les cartes many-cores d'Intel ou « Xeon Phi »

Il est important de détailler l'architecture des coprocesseurs Intel Xeon Phi. Il s'agit d'un accélérateur qui a été très utilisé dans les systèmes de calcul à haute performance durant ces dernières années, notamment en étant à la base de la très haute performance du super calculateur Chinois Tianhe 1-A qui a été le leader du Top 500 pendant 5 benchmark d'affilée. Dès 2012, ce processeur était déjà présent dans 7 superordinateurs parmi les plus rapides du monde dans le Top 500 mais surtout parmi les plus économes en énergie (http://green500.org). Jeffers et Reinder (Jeffers and Reinder 2013) ont souligné les capacités des processeurs Intel Xeon Phi, conçus pour faciliter le passage à échelle de codes conçus pour les processeurs Intel Xeon classiques. De plus ces accélérateurs permettent une exploitation plus large des opérations vectorielles et disposent d'une bien meilleure bande passante mémoire par rapport aux processeurs Intel classiques. Grâce à son architecture supportant très facilement le portage d'applications conçues pour des architectures x86, les Xeon Phi sont pleinement généralistes et permettent de supporter bien les applications qui ne peuvent pas profiter de l'architecture massivement vectorielle des GP-GPUs.

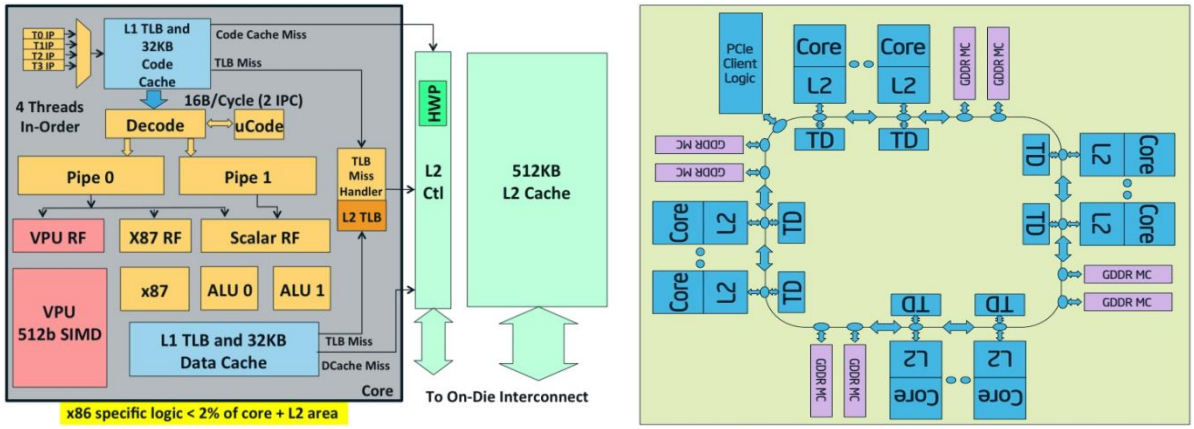
a - Introduction générale à l'architecture MIC ou KNC

Les coprocesseurs ou accélérateurs Intel Xeon Phi sont basés sur l'architecture Knights Corner (KNC) introduite en 2011 qui a évoluée vers l'architecture dite Knights Landing (KNL). Nous sommes avec

ces architectures en présence de multiprocesseurs MIMD many-cores ou MIC pour « *Many Integrated Core* » - MIC en anglais. Voici les principales caractéristiques de ce type de co-processeur (Jeffers and Reinder 2013) (Colfax 2013):

- Il est gravé en 22nm avec des transistors 3-D de type Trigate. La consommation électrique se limite à 255 watts par coprocesseur.
- La première version comportait 50 cœurs x86, mais les 1ères versions commercialisées proposaient 60, les versions actuelles proposent 72 cœurs physiques. Chaque cœur physique propose 4 threads matériels vus comme 4 nouveaux cœurs logiques par un système d'exploitation de type Unix. On disposait de 244 threads matériels sur les puces disponibles entre 2012 et 2015 sur un seul dispositif. Les premiers cœurs physiques étaient cadencés à 1 GHz sur les 5110P, à 1.2 Ghz sur les 7220P et maintenant 1.5 Ghz sur les 7290P avec 16 GB de RAM (introduits en juin 2016). Chaque cœur physique dispose d'un cache local L2 512-KB conformément à la Figure 3.6.a.
- Les cœurs sont connectés par une interconnexion avec une topologie en anneau bidirectionnel comme montré sur la Figure 3.6.b.
- Une caractéristique importante pour la reproductibilité numérique est que les cœurs exécutent leurs instructions dans l'ordre (« *in-order* » en anglais). Ils peuvent traiter jusqu'à deux instruction par cycle avec les pipelines u-pipe et v-pipe.
- Les Xeon Phi n'utilisent pas les anciens jeux d'instructions vectorielles de type Intel MMX, ou SSE. L'unité vectorielle qui traite 512 bits se nomme AVX-512. Ces instructions SIMD pour le jeu d'instructions x86 ont été proposées par Intel et sont prises en charge depuis l'introduction des processeurs Intel Xeon Phi. Cette unité peut ainsi traiter simultanément 16 variables en simple précision ou 8 variables en doubles-précision (Figure 3.6.a).
- Les coprocesseurs Xeon Phi fonctionnent sur un système d'exploitation Linux avec une architecture binaire nommée k1om qui n'est pas exactement l'architecture classique x86 à 64 bits. L'ensemble des langages supportés sont ceux du domaine du calcul à haute performance : C/C++/Fortran. Les outils importants de développement sont fournis principalement par Intel, notamment les compilateurs C/C++/Fortran, le support pour MPI et OpenMP, les bibliothèques mathématiques à haute performance MKL (*Math Kernel Library*), les outils de débogage et d'analyse des performances (*VTune Amplifier XE*).
- Le coprocesseur est en général connecté à un processeur Intel Xeon (ou un autre processeur hôte) via un bus PIC Express (PICe). L'implémentation d'une pile TCP/IP virtualisée permet l'accès au réseau.

On peut voir le coprocesseur comme un « système multiprocesseur » de traitements homogènes. Il s'agit vraiment d'un SMP-on-a-chip fonctionnant avec le système d'exploitation Linux et des binaires très proches du standard x86. Pour mémoire, un SMP (« *Symmetric MultiProcessor* » en anglais) est un système de multiprocesseurs avec de la mémoire partagée et exécutant un seul système d'exploitation.



(a) Architecture d'un cœur du coprocesseur (b) Micro-architecture du coprocesseur

Figure 3.6. Présentation de l'architecture des cœurs en anneaux à droite et détail d'un seul cœur du coprocesseur Intel Xeon Phi (à gauche) (Jeffers and Reinder 2013)

La Figure 3.6.a présente l'architecture d'un cœur dans un coprocesseur de type Xeon Phi. Le VPU (*Vector Processing Unit*) est l'unité de traitement vectorisée qui contient l'unité mathématique étendue pour le calcul en virgule flottante. Le RF (*Register File*) correspond à l'ensemble des registres. Le TLB (*Transaltion Lookaside Buffer*) est une table de gestion des pages et du cache pour accélérer la recherche des adresses de mémoire physique correspondant à des adresses de mémoire virtuelle. Le HWP (*Hardware Prefetching*) est initialisé au premier échec de cache (« *cache miss* » en anglais) dans une page. Le pré-chargement est une fonctionnalité du cache et il est possible de demander que les données soient chargées dans le cache avant que le cœur les utilise afin qu'elles soient directement disponibles pour les calculs. Le coprocesseur Intel Xeon Phi prend en charge le pré-chargement matériel quand une unité de matériel dédiée dans le cache peut apprendre le modèle d'accès aux données et lancer les instructions de chargement automatiquement. Un pré-chargement logiciel est aussi possible lorsque l'instruction de pré-extraction est émise par le code du logiciel en prévision d'une utilisation future des données (Colfax 2013) (Krishnaiyer et al. 2013).

La Figure 3.6.b présente l'architecture d'un coprocesseur Xeon Phi. Le TD est un répertoire divisé en 64 sous-répertoires. Les caches de niveau 2 (L2) sont maintenus parfaitement cohérents les uns avec les autres par le TD. Un TD contient les adresses, les états et l'identité du propriétaire de la ligne de cache. La partie notée GDDR MC correspond au contrôleur de mémoire GDDR5.

b - Modèles de programmation avec un processeur Intel Xeon Phi

La programmation avec un coprocesseur Intel Xeon Phi est une approche de calcul hybride. En effet, on se retrouve avec un système de calcul hétérogène qui propose une exécution sur des processeurs Intel Xeon classiques (les CPUs avec l'architecture multi-cœurs) et l'exécution sur l'Intel Xeon Phi (une carte proposant un coprocesseur avec l'architecture many-cores). On a donc des approches et des modes de programmation différents : des exécutions natives sur le processeur hôte classique, des exécutions basées sur une décharge des calculs (« *offload* » en anglais) qui correspond à un transfert des données et des codes au coprocesseur (Jeffers and Reinders 2013). Un calcul qui fera usage des deux types de processeurs (multi-cœurs et many-cores) est dit hybride. La figure 3.7 ci-dessous donne un panel des approches possibles.

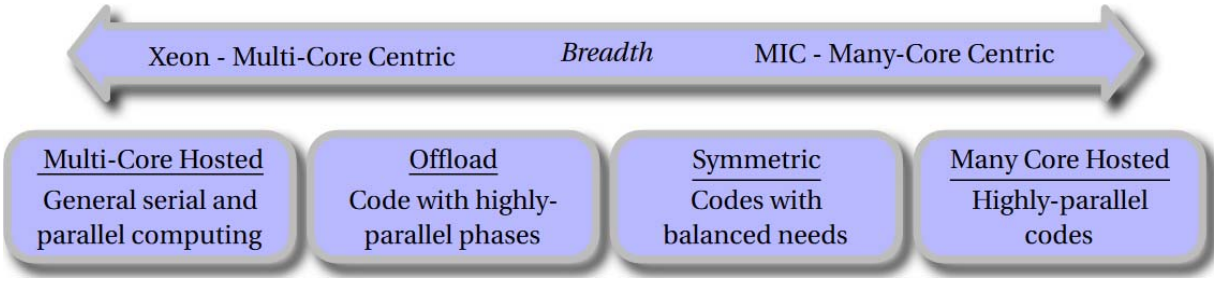


Figure 3.7. Les options de développement d'application sur l'Intel Xeon Phi coprocesseur (Colfax 2013).

Dans (Jeffers and Reinders 2013) (Colfax 2013) (Intel SDG 2014), nous voyons que ces approches permettent aux développeurs de concevoir une gamme de modes de programmation qui correspondent aux types du calcul que l'on peut rencontrer dans une vaste gamme d'applications :

- Exécution sur l'hôte multi-cœurs: les codes travaillent seulement sur le CPU. C'est l'exécution native sur l'hôte.
- Exécution multi-cœurs-centrique: l'exécution sur le système hôte avec certaines opérations réalisées sur le coprocesseur. Il s'agit d'une exécution déchargée. Dans ce cas, l'ensemble des deux architectures multi-cœurs et many-cores travaillent. Normalement, le code précise cet état de fait avec la directive « *#pragma offload* ». La partie du code après cette directive va transférer les calculs et données au coprocesseur, ensuite, elle va recevoir les résultats de ce dernier.
- Exécution multi-cœurs-symétrique: les codes utilisent l'hôte et le coprocesseur en essayant d'équilibrer la charge de calcul. C'est l'exécution du même code en même temps sur les deux architectures
- Exécution sur la carte many-cores : les codes s'exécutent exclusivement sur un coprocesseur ou un ensemble de coprocesseurs. C'est l'exécution native sur le MIC.

Le choix d'un mode d'exécution dépend du profil de l'application à paralléliser (en fonction de la charge de travail en séquentiel, en parallèle ou en vectoriel), du type de communication entre les ouvriers, du coût de transport de données, etc. Dans le cas d'une exécution uniquement sur le coprocesseur Intel Xeon Phi, on utilise l'option « ***-mmic*** » pour compiler les codes.

c - Différentes approches d'implémentation du parallélisme sur Intel Xeon Phi

Pour accélérer la performance de calcul sur les coprocesseurs Intel Xeon Phi, il est important de pouvoir utiliser le *multi-threading* sur chaque cœur comme une clé pour faire diminuer les latences inhérentes à une micro-architecture utilisant une exécution dans l'ordre (Jeffers and Reinders 2013). Jeffers et Reinders ont montré (voir Figure 3.8) que pour avoir une efficacité maximale des performances, on a besoin d'implémenter une application en parallèle sur du matériel supportant ce parallélisme. Selon (Colfax 2013), nous avons trois types d'implémentation différentes pour exploiter le parallélisme : parallélisation sur les données avec une approche vectorielle sur un code séquentiel, parallélisation des tâches avec une mémoire partagée, et parallélisation des tâches avec de la mémoire distribuée.

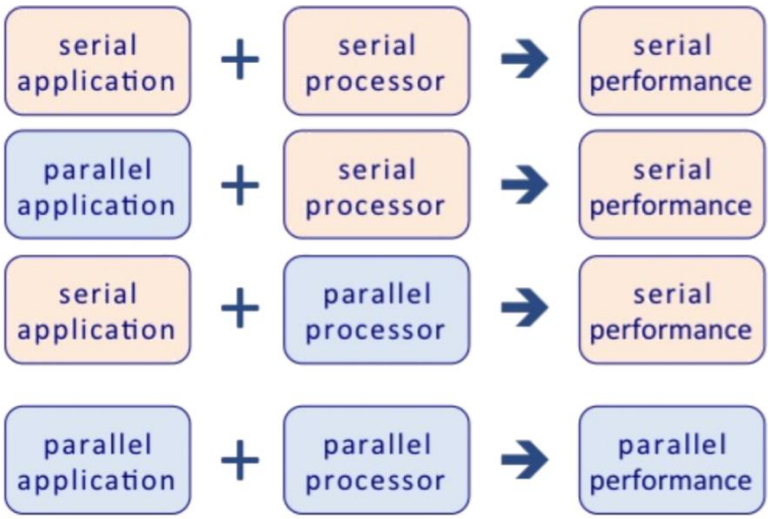


Figure 3.8. L'accès aux hautes performances provient à la fois du matériel parallèle et du logiciel parallèle (Jeffers and Reinders 2013).

La première approche pour une implémentation parallèle utilise la parallélisation des données avec des codes initialement séquentiels. Ceci suppose une utilisation importante des instructions vectorielles et un code qui est adapté à cette approche SIMD (« *Single Instruction Multiple Data* » en anglais). La vectorisation automatique est activée dès que le niveau d'optimisation par défaut est ***–O2***. Ceci suppose que les données puissent circuler pour alimenter efficacement les instructions vectorielles sans surcharge excessive du bus d'accès aux données pour produire des résultats de manière efficace. L'efficacité dans le mouvement des données dépend de la mise en page de

données (*data layout*), de leur alignement, leur pré-chargement (*prefetching*) et également d'un stockage efficace (Jeffers et Reinders 2013).

La deuxième approche utilise la parallélisation des tâches dans une mémoire partagée en utilisant des bibliothèques telles qu'Intel Cilk Plus et Intel Open Muti-Processing (OpenMP). Le choix entre Cilk Plus ou OpenMP pour la parallélisation dépend du profil de l'application et des considérations de performance. Le lecteur intéressé par plus d'informations sur ces deux bibliothèques se reportera à (Colfax 2013). Un programme avec OpenMP utilisant une approche *fork/join* est présenté sur la Figure 3.9.

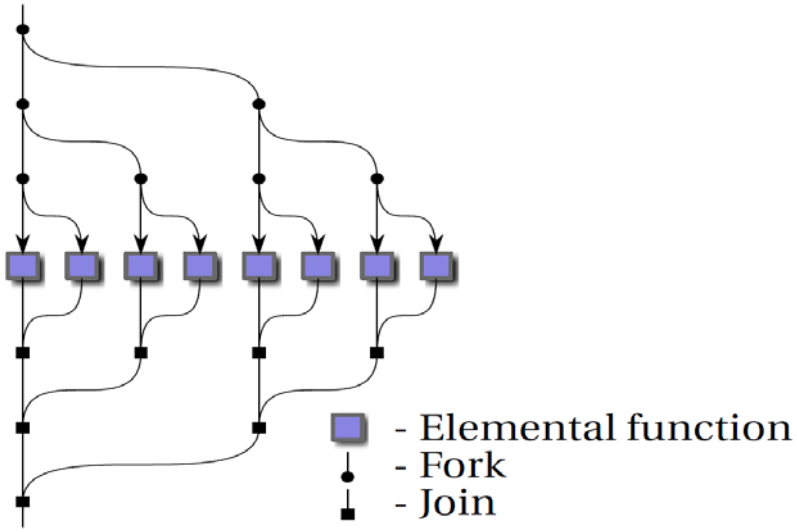


Figure 3.9. Modèle fork-join d'exécution du parallélisme de la mémoire partagée (Colfax 2013).

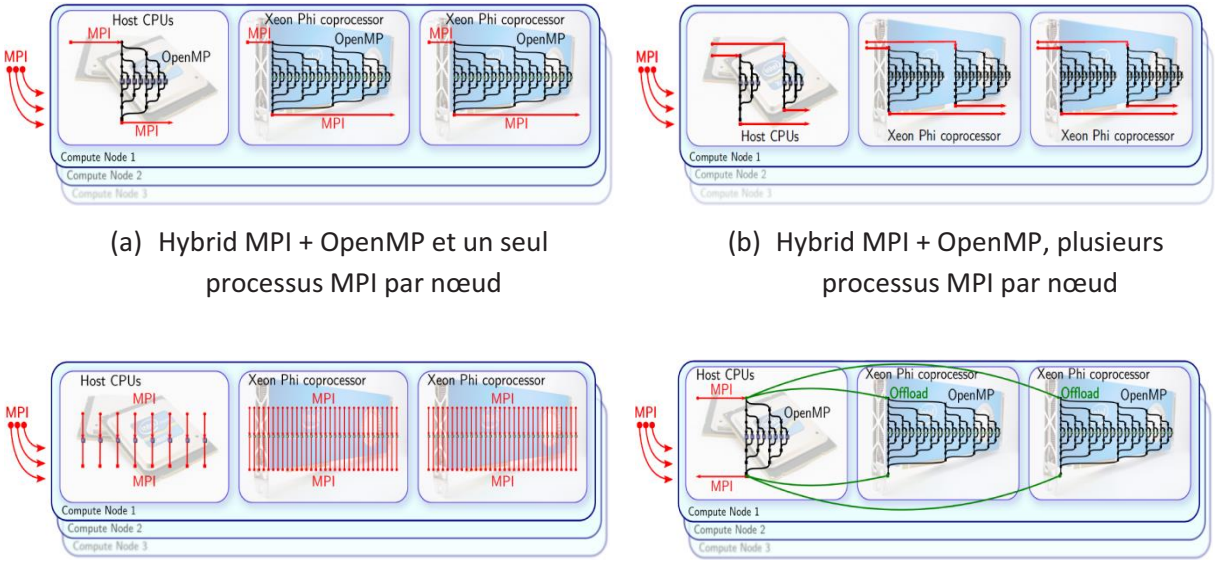
Dans la Figure 3.9, nous avons le modèle classique fork-join avec une tâche fille pouvant être exécutée en parallèle avec la tâche parente. La création se passe au nœud qui se traduit par 2 nouveaux fils d'exécution (c'est l'étape « fourchette »). Cette exécution se déroule en parallèle jusqu'à ce qu'une barrière soit atteinte. Cette barrière met fin à la tâche fille (l'étape « jointure »). Les tâches filles peuvent « enfanter » et créer un arbre de tâches (Colfax 2013). En fait, l'étape de création d'une fille (fourchette) correspond à un lancement de thread quand on se place dans l'espace des processus légers (par opposition aux processus lourds lancés par des « *forks* » du système UNIX) L'étape de jointure finale est assurée par le thread maître. Dans (Colfax 2013), nous trouvons la structure de programme OpenMP bien détaillée et une explication pourquoi les threads POSIX (ou pthread) ne sont pas un bon choix pour du calcul à haute performance.

La troisième approche consiste à effectuer une parallélisation des tâches avec de la mémoire distribuée en utilisant des échanges de message et la bibliothèque MPI (« *Message Passing*

Interface » - MPI). MPI repose sur un protocole de communication avec 3 aspects différents qui sont la synchronisation, le mouvement de données (de type: *broadcast*, *scatter* et *gather* et *all-to-all communication*) et la réduction (calcul à partir des résultats collectifs obtenus après un « *mapping* » des calculs aux différents threads). L'utilisation de la bibliothèque MPI permet de multiples traitements sans partage d'une mémoire commune. Il s'agit de mettre en place un réseau d'éléments de calcul effectuant des calculs parallèles avec la communication entre les éléments de traitements par échange de messages. L'article que nous utilisons en référence pour présenter le coprocesseur Xeon Phi (Colfax 2013) montre 4 façons d'utiliser la bibliothèque MPI, dont 3 sont hybrides avec Open MP (Figure 3.10) :

(a) chaque élément de calcul présente un seul processus MPI et exploite le parallélisme de chaque machine avec un cadre parallèle à mémoire partagée d'OpenMP.

(b) chaque élément de calcul exécute OpenMP et plusieurs processus MPI.



(a) Hybrid MPI + OpenMP et un seul processus MPI par nœud

(b) Hybrid MPI + OpenMP, plusieurs processus MPI par nœud

(c) MPI pur

(d) Hybrid MPI + OpenMP avec « *offload* » de l'hôte vers les coprocesseurs

Figure 3.10. Différents modèles simples ou hybrides utilisés pour MPI dans un système du calcul avec multi-cœurs et many-cores (Colfax 2013).

(c) chaque élément de calcul (cœur physique et logique) des différentes architectures utilise un thread MPI.

(d) le processus MPI maître sur l'ordinateur hôte, utilise d'une part plusieurs processus légers avec OpenMp, mais il se décharge également de certaines portions de calcul (*offload*) d'autre part.

Les cas (a) (b) et (c) font des exécutions natives sur les hôtes et les coprocesseurs. Dans le cas (d) par contre, un mécanisme de décharge de calculs est utilisé sur 2 architectures différentes.

Les langages de programmation usuels n'ont pas été conçus pour le parallélisme. Il arrive dont fréquemment qu'on leur ajoute des nouvelles fonctionnalités, soit par des bibliothèques, soit par des directives ou des constructions supplémentaires (telles que « DO CONCURRENT ») pour faire de la programmation parallèle (Jeffers et Reinders 2013). Les coprocesseurs Xeon Phi permettent aux développeurs parallèles d'utiliser les mêmes outils, les mêmes langages de programmation et les mêmes modèles de programmation que ceux à disposition sur des Intel Xeon classiques. Les 3 approches précédemment présentées sont disponibles dans un environnement à base de Xeon Phi. Le choix d'une approche plutôt qu'une autre dépend de l'objet de chaque programme.

d - Compilation et exécution sur Xeon Phi

Avec un coprocesseur Intel Xeon Phi, au moins deux compilateurs sont disponibles: Intel C/C++ Compiler (*icc* et *icpc*) pour la programmation en C et C++, et Intel Fortran Compiler XE (*ifort*) pour la programmation en Fortran (Intel SDG 2014). La syntaxe en ligne de commande pour compiler un programme en C est classique (Intel ICC 2016):

icc [options] [@response_file] file1 [file2…]

où ***options*** représente pas ou plusieurs options et ***file*** *1 à N* les fichiers à compiler dans les différents formats acceptés (.C, .c, .cc, .cpp, .cxx, .c++, .i, .ii, .s, .S, .o, .a). Le fichier ***response_file*** optionnel liste les options du compilateur. Pour que le code fonctionne sur les cœurs d'un coprocesseur Intel Xeon Phi, il faut utiliser l'option ***–mmic*** sur un système Linux et ***/Qmic*** sur Windows. Il s'agit alors d'une compilation croisée avec la compilation sur le processeur hôte x86 et la génération de code pour le processeur cible many-core en architecture k10m. Les compilateurs Intel utilisent des variables d'environnement pour fonctionner correctement. Sur de nombreuses plates-formes, il convient d'exécuter auparavant un script pour configurer l'environnement d'exécution (par exemple sous Linux: **source /$path/compilervars.sh mode_compile**).

Avec ce type de script, nous avons le chemin par défaut pour les compilateurs 32 et 64 bits (**/opt/intel/composer_xe_2013_sp1/bin** sous Linux). Nous présentons ci-après un mini guide pour compiler et lancer un programme sur Intel Xeon Phi avec la production d'un code exécutable sur l'architecture k10m (code 3.1). Le coprocesseur se nomme **mic0** et on s'y connecte par **ssh**.

```
source  /opt/intel/composer_xe_2013_sp1/bin/compilervars.sh  intel64

icc  -mmic  hello.c

sudo scp  a.out  mic0:~/

ssh mic0
```

```
./a.out
```

Code 3. 1. Exemple de compilation et d'exécution d'une application sur MIC (Colfax2013).

Le code ci-dessus présente le processus de compilation et d'exécution d'un programme en C qui va s'exécuter sur un Xeon Phi rattaché à un Xeon classique. On peut enlever l'option « **-mmic** » pour compiler et lancer directement l'exécution sur l'hôte sans se connecter au co-processeur. Ici la sortie par défaut pour la compilation est classiquement dans le fichier « **a.out** ».

Pour une application MPI, on utilise la même approche que le Code 3.1 avec quelques changements dans la configuration de l'environnement d'exécution (voir Code 3.2).

```
source  /opt/intel/impi/4.1.0/intel64/bin/mpivars.sh

##sur l'hôte

mpiicc  -o  HelloMPI.XEON  HelloMPI.c

mpirun  -host localhost  -np 2 ./HelloMPI.XEON

##sur le MIC

mpiicc  -mmic -o  HelloMPI.MIC  helloMPI.c

sudo scp HelloMPI.MIC mic0:~/

export I_MPI_MIC=1

mpirun –host  mic0 –np 2 ~/HelloMPI.MIC
```

Code 3. 2. Un exemple de compilation et d'exécution d'une application MPI (Colfax 2013).

Ce Code 3.2 présente deux cas de compilation, l'un pour la machine hôte et l'autre pour le MIC. La différence entre eux est l'argument « **-host XXYY**». Il est nécessaire de configurer l'environnement « **I_MPI_MIC=1** » pour utiliser les fonctionnalités de MPI sur le MIC. L'option « **-np XX** » présente le nombre de cœurs utilisés.

Pour une application où l'on décharge une partie des calculs sur un Xeon Phi, on utilise la directive ***#pragma offload <clauses> <statement>***. Il y a beaucoup d'options différentes qui dépendent du type de décharge. Par exemple, si l'on veut contrôler le trafic des données entre l'hôte et le coprocesseur, il est possible d'utiliser ***#pragma offload_transfer*** ou ***#pragma offload*** ou encore un transfert de données asynchrones ***#pragma offload_wait***, ***#pragma omp***, etc (Jeffers and Reinder 2013).

Pour une application qui utilise les instructions vectorielles, on utilisera une directive ***#pragma simd [clause[[,] clause]…]*** ou ***#pragma vector [clause[[,] clause]…]*** dans le code.

e - Options de précision et d'exactitude

Dans cette partie, nous choisissions de présenter plus particulièrement les options impactant le calcul en virgule flottante. Leur impact peut être très significatif quant aux résultats obtenus, notamment dans le contexte d'un système de calcul hétérogène à base de Xeons de la famille Intel. (Tableau 3.2).

Argument	Effect
-fp-model strict	Only value-safe optimizations.
-fp-model precise	Only value-safe optimizations, exception control is disabled (but may be enabled using *–fp-model except*). **Serial floating-point calculations are reproducible from run to run.**
-fp-model fast=1	Value-unsafe optimizations are allowed, exceptions are not enforced, contractions are enabled.
-fp-model fast=2	Enables more aggressive optimizations than *fast=1*, possibly leading to better performance at the cost of lower accuracy.
-fp-model source	Intermediate arithmetic results are rounded to the precision defined in the source code. **Using *source* also assumes precise**, unless overridden by *strict* or *fast*.
-fp-model double	Intermediate arithmetic results are rounded to 53-bit (double) precision. Using *double* also assumes *precise*, unless overridden by *strict* or *fast*.
-fp-model extended	Intermediate arithmetic results are rounded to 64-bit (extended) precision. Using *extended* also assumes *precise*, unless overridden by *strict* or *fast*.
-fp-model [no-]execpt	*except* enables, *no-except* disables the floating-point exception semantics.
-fimf-precision= value[:funclist]	Defines the precision for math library functions. Here, value is one of: *high*, *medium* or *low*.
-fimf-max-error= ulps[:funclist]	The maximum allowable error expressed in ulps (units in last place)
-fimf-accuracy-bits= n[:funclist]	The number of correct bits required for mathematical function accuracy.
-fimf-domain-exclusion= n[:funclist]	Defines a list of special-value numbers that do not need to be handled by the functions. Here, n is an integer derived by the bitwise OR of the

	following values: extremes: 1, NaNs: 2, infinites: 4, denormals: 8, zeroes: 16.

Table 3.2. Extrait d'une table des arguments pour contrôler la sémantique des calculs en virgule flottante pour un haut niveau de précision des fonctions mathématiques avec le compilateur Intel C++(Colfax 2013) (Intel ICC 2016).

Nous présentons dans cette table un extrait des arguments pour contrôler la sémantique des opérations en virgule flottante avec ***-fp-model*** et les fonctions mathématiques avec ***-fimf-***. Il s'agit de notre synthèse des options d'optimisation utilisées pour le calcul en virgule flottante avec un compilateur Intel C++. Le lecteur intéressé trouvera plus de détails dans les documents (Colfax 2013) et (Intel ICC 2016). Ces arguments influencent significativement les résultats obtenus et peuvent créer des résultats non-reproductibles sur des plate-formes différentes de la famille Intel (Xeon classique vs. Xeon Phi notamment) (Corden et Kreitzer 2010).

Quand on maîtrise ces options de compilation et d'optimisation, les accélérateurs many-cores de type Intel Xeon Phi sont un bon choix pour accélérer la performance de tous types d'applications parallèles scientifiques écrites en C/C++ ou Fortran. La facilité de portage des applications et le spectre large des domaines applicatifs ont fait que ce type d'accélérateur généraliste a rapidement été très utilisé sur de nombreux supercalculateurs (notamment sur le supercalculateur chinois Tianhe-1A qui a été 5 fois de suite en tête du Top 500). On peut utiliser tous les modèles de programmation parallèle et profiter d'une large bande passante d'accès à la mémoire. Cependant, pour certaines applications avec des besoins vectoriels forts, les cartes accélératrices à base de GP-GPU ont encore de meilleures performances. De même, pour des applications très spécifiques, il est possible de développer des FPGA dédiés pour obtenir la meilleure performance crête.

Les puces neuromorphiques

Une nouvelle tendance est apparue récemment dans la conception des architectures d'ordinateur. Elle vise à concevoir des machines avec un fonctionnement similaire à celui d'un cerveau biologique. Au lieu de simuler le cerveau sur un ordinateur classique, on fabrique des puces qui ont le comportement de neurones biologiques. L'objectif est de proposer des simulateurs de cerveaux avec un matériel structurellement proche du fonctionnement biologique. Ce type d'architecture implémente des réseaux de neurones artificiels avec du matériel électronique (Hsu 2014a) (Schuller and Stevens 2015) (Modha 2016).

Les unités fonctionnelles ou blocs sont constituées de neurones, d'axones, de synapses et de dendrites. Schuller et Stevens ont également présenté différentes fonctions que l'on retrouve dans les zones cérébrales.

> *"**Somata/Neuristor** (also known as **neuron bodies**), which function as integrators and threshold spiking devices…These have two important properties: switching and plasticity.*
>
> ***Synapses/Memristor**, which provide dynamical interconnections between neurons*
>
> ***Axons/Long wire**, which provide long-distance output connection between a presynaptic to a postsynaptic neron*
>
> ***Dendrites/ Short wire**, which provide multiple, distributed inputs into the neurons"*
> *(Schuller and Stevens 2015).*

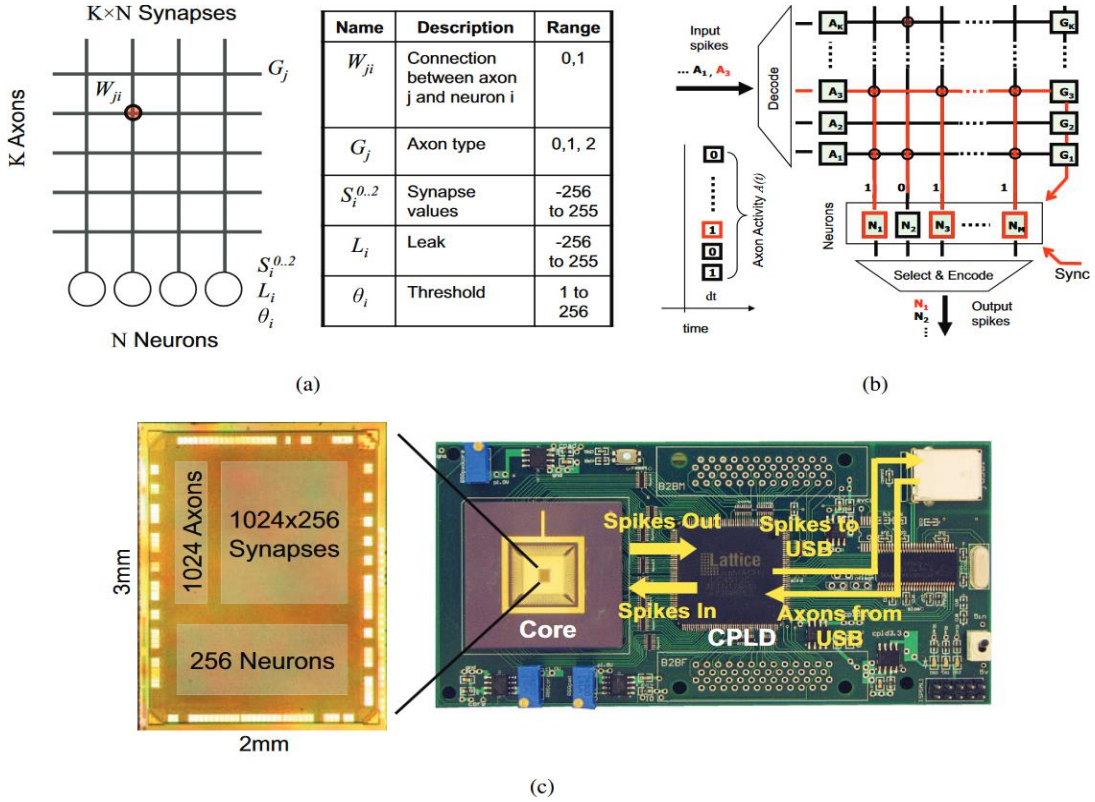
Les synapses sont les interconnexions entre neurones : elles se souviennent de l'état précédant, mettent à jour un nouvel état et maintiennent le poids de la connexion. Un axone est vu comme un bus. A chaque fonction correspond un type de matériel électronique.

Dans Arthur et al. 2012, nous trouvons (Figure 3.11) une proposition de puce neuro-synaptique et son implémentation dans une carte de test. Dans la Figure 3.11.a, on voit une grille contenant K = 1024 axones qui connectent N = 256 neurones via un réseau de K*N synapses aux valeurs binaires. On a aussi une table de présentation des symboles pour cette structure: W_{ji} est une connexion entre axone *j* et neurone *i* ; un axone j est statiquement spécifié comme étant d'un des trois types G_j = {0, 1, 2} pour différencier la nature des connexions, par exemple excitateurs ou inhibiteurs ; les $S_i^{G_j}$ donnent les valeurs des synapses entre neurone i et axone j, etc.

La Figure 3.11.b présente l'opération détaillée d'un cœur sur deux phases. Dans la première, un événement entrant active l'axone 3 (A_3), qui lit les connexions et met à jour les résultats pour les neurones (N_1, N_3 et N_M). La deuxième phase met en œuvre une synchronisation temporelle. Dans cette dernière, un événement de synchronisation (Sync) est envoyé à tous les neurones. En recevant cette synchronisation, chaque neurone vérifie si son potentiel de membrane est au-dessus du seuil, et si oui, il produit une impulsion « *spike* » et réinitialise le potentiel de la membrane à 0. Ces *spikes* sont codés et envoyés comme des événements d'une manière séquentielle.

La Figure 3.11.c montre la taille d'un cœur élémentaire proposant 256 neurones. Ce cœur est fabriqué dans un processeur gravé en 45nm avec 3.8 millions transistors dans 4.2mm^2 de silicium. A droite, nous voyons une carte de test qui s'interface avec la puce via une liaison USB 2.0: les

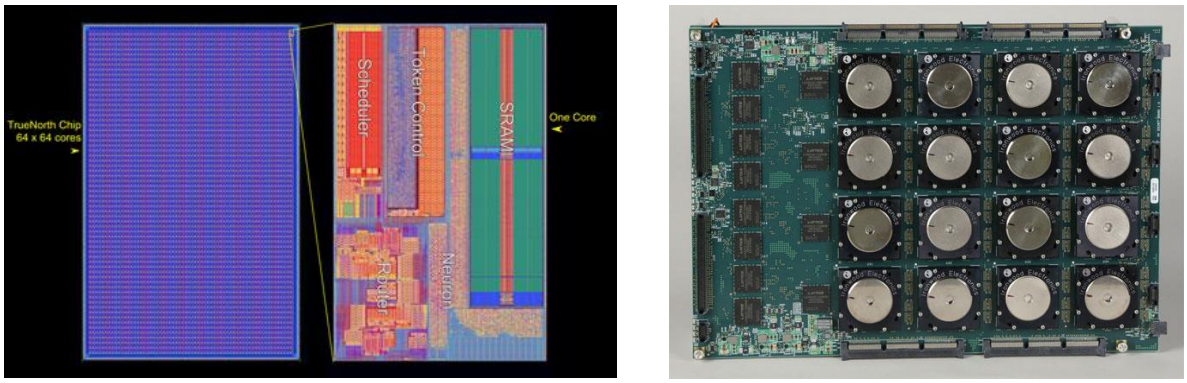
événements de type « *spike* » sont envoyés à un PC pour collecter les données, et ils sont également acheminés via les interfaces d'entrée/sortie de la puce pour permettre la connectivité avec d'autres réseaux de neurones programmables.



Name	Description	Range
W_{ji}	Connection between axon j and neuron i	0,1
G_j	Axon type	0,1, 2
$S_i^{0..2}$	Synapse values	-256 to 255
L_i	Leak	-256 to 255
θ_i	Threshold	1 to 256

Figure 3.11. Cœur d'une architecture neuro-synaptique avec 256 neurones et les interconnexions pour passer à l'échelle de grands réseaux de neurones (Arthur et al. 2012)

Récemment, Modha (Modha 2016) a proposé une nouvelle puce neuronale nommée TrueNorth (Figure 3.12.a). Cette puce dispose de 4096 cœurs via un réseau *on-chip* pour créer 1 millions neurones et 256 millions synapses. Une carte neuromorphique IBM (figure 3.12.b) propose 16 puces TrueNorth, soit 16 millions de neurones et 4 billions synapses.



(a) Tableau du cœur d'une puce TrueNorth (b) Chipboard de 16 puces

Figure 3.12. Présentation d'une puce et d'une carte de puces neuromorphiques d'IBM (Modha 2016).

Cette approche de développement de puces neuromorphiques crée de nouvelles architectures d'ordinateur. Elle suppose une nouvelle approche de la programmation, même s'il existe des API (*Application Programming Interface*) qui permettent le travail avec des langages de programmation classiques. Par exemple, PyNN (prononcé «pine») est un langage indépendant du simulateur matériel pour construire des modèles de réseaux neuronaux. Dans le projet Européen *Human Brain project*, le langage permet de lancer des exécutions sur des plate-formes matérielles différentes telles que BrainScale ou Spinnaker.

Les accélérateurs quantiques

D-wave (D-wave 2015) propose un nouveau type d'accélérateur de calcul basé sur la mécanique quantique. L'ordinateur utilise les bits quantiques (« *quantum bits* » ou qubits en anglais) qui en plus des valeurs habituelles 1 ou 0, peuvent exister simultanément dans les états 1 et 0. C'est une différence fondamentale avec l'ordinateur classique. Sur la base de la théorie des probabilités, des dispositifs quantiques sont en mesure d'effectuer plusieurs calculs à la fois en explorant la combinatoire imposée par les qubits.

Nous présentons des éléments de l'architecture d'un processeur quantique à la Figure 3.13 que nous avons modifié à partir d'une image originale de l'article (D-wave 2015). Un processeur quantique de ce type contient un treillis de circuits supraconducteurs minuscules (qubits) fabriqués à partir de niobium métallique. Il présente des comportements quantiques à très basses températures. Les qubits sont les éléments de base que le système utilise pour résoudre les problèmes. Le processeur quantique est entouré par l'électronique utilisée pour programmer les processeurs et lire les résultats.

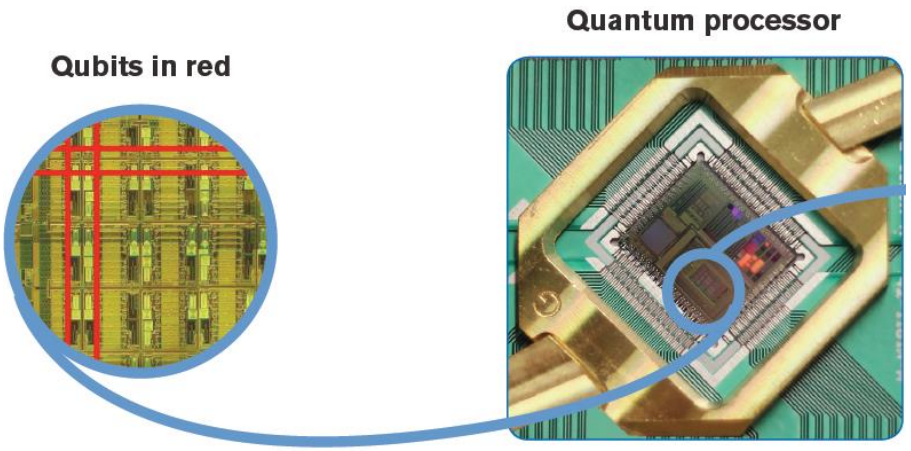


Figure 3.13. Une présentation extraite d'un processeur quantique (D-wave 2015).

Hey (Hey 1999) a donné une présentation simple d'un ordinateur quantique avec les fondamentaux pour avoir des portes logiques réversibles à partir de qubits et de registres quantiques. Il a proposé

aussi un processus de compilation quantique qui traduit un algorithme quantique en une suite d'opérations physiques réelles sur un système quantique.

Comparing the performance of the device on random spin glass instances with limited precision to simulated classical and quantum annealers, we find no evidence of quantum speedup when the entire data set is considered, and obtain inconclusive results when comparing subsets of instances on an instance-by-instance basis (Rønnow et al. 2014).

L'ordinateur quantique semble actuellement bien adapté à l'optimisation discrète, ainsi qu'à la cryptographie. Cependant, les systèmes de calcul à haute performance classiques sont beaucoup mieux adaptés aux simulations numériques à grande échelle qui restent des programmes généralistes. Dans une étude récente réalisée par l'équipe de Rønnow (Rønnow et al. 2014), l'accélération du dispositif D-Wave Two avec 503 qubits n'a pas mis en évidence d'accélérations notables. A l'inverse, avec le même matériel en 2015, Google a dit avoir résolu certains problèmes 100 millions de fois plus vite qu'avec un ordinateur ordinaire. Toujours selon Google, ce résultat pourrait amener d'énormes améliorations dans le domaine de l'intelligence artificielle. Des experts mettent en doute cette annonce. Ces études sont commentées en précisant que l'accélération quantique est insaisissable et peut dépendre de la question posée. De son côté, IBM a mis au point des accélérateurs quantiques et propose même depuis 2015 de réaliser des expériences quantiques sur un petit nombre de qubits – 5 précisément. (http://www.research.ibm.com/quantum/).

II.3 - Modèles de programmation parallèle

Dans son article, Diaz (Diaz et al. 2012) a montré que la plupart des applications logicielles sont encore développées selon un modèle d'exécution séquentielle. Il est ainsi difficile, selon (Jeffers et Reinders 2013), de parler de performance parallèle même si ces applications marchent souvent sur des systèmes parallèles élémentaires (microprocesseurs multi-cœurs) ou encore lorsque ces applications tournent sur des systèmes de calcul à haute performance (avec une approche *Single Program Multiple Data*), on peut le voir dans la Figure 3.8. Même si de nombreux programmes, y compris pour la bureautique, utilisent plusieurs threads, une grande majorité des applications restent encore séquentielles et n'utilisent pas toutes les ressources de calcul à leur disposition. Par contre, pour un programme parallèle, il est vraiment nécessaire de disposer d'un système de traitement parallèle pour avoir une exécution efficace. Quand on ne souhaite pas revisiter le code des applications séquentielles, mais que l'on souhaite exploiter les ressources matérielles parallèles,

il existe des outils de parallélisation automatiques. Tout d'abord, les programmes séquentiels sont parallélisés automatiquement quand cela est possible en utilisant l'ILP (« *Instruction level parallelism* » en anglais) des processeurs modernes, mais aussi en utilisant les options des compilateurs qui génèrent par exemple, des exécutions en parallèles de codes qui peuvent être vectorisés simplement (ne serait-ce qu'en effectuant une optimisation au niveau des boucles). Cependant, ces approches ne sont pas suffisamment efficaces. Il faut donc bien penser aux approches de programmation parallèle pour exploiter au mieux les ressources matérielles à disposition. Dans le reste de cette section, nous présenterons les différents modèles de programmation qui permettent de paralléliser une application en général, aussi bien pour des systèmes hétérogènes ou hybrides que pour des systèmes classiques.

Diaz (Diaz et al. 2012) a réalisé une enquête sur les modèles de programmation parallèle et proposé une classification en 4 catégories :

- modèle parallèle pur avec mémoire partagée utilisant OpenMP ou Pthread, ou avec mémoire distribuée en utilisant MPI;
- modèle de programmation parallèle hétérogène ou hybride pour les nouvelles architectures avec un ou plusieurs CPU multi-cœurs et un ou plusieurs GP-GPU, ou encore un ou plusieurs CPU multi-cœurs et un ou plusieurs processeurs manycore de type Intel Xeon Phi;
- modèle de parallélisme partitionnant un espace d'adresse global (« *Partitioned global address space* » - PGAS en anglais);
- modèle hybride avec une combinaison de mémoire partagée et de mémoire distribuée.

L'article montre aussi des exemples de programmes distribués et indique les langages de programmation qui supportent la programmation parallèle.

Barney (Barney 2015) a également proposé une autre synthèse des modèles de programmation parallèle : modèle de mémoire partagée sans ou avec threads ; modèle à base de mémoire distribuée ; modèle de parallélisation sur les données ; et modèle hybride. Même si cette classification a des similitudes avec celle de Diaz, elle ajoute deux modèles SPMD et MPMD (*Single Program Multiple Data* – déjà évoqué et *Multiple Program Multiple Data*). Barney a également mis en relief le fait que ces modèles de programmation existent en tant qu'abstraction au-dessus des architectures matérielles. Il n'existe pas à proprement parler de « meilleur » modèle, bien qu'il y ait certainement de meilleures implémentations en fonction des applications que l'on souhaite

paralléliser. A partir des classifications de Diaz et de Barney, nous proposons ci-dessous une table (Table 3.3) récapitulant les principaux modèles de programmation parallèle :

Modèle	Description	Implémentation
A mémoire partagée	Dans ce modèle, les tâches/processus partagent un espace d'adresse commun qu'ils peuvent lire et écrire de manière asynchrone. Différents mécanismes permettent de contrôler ces accès à la mémoire partagée (ex: les verrous – ou *locks* en anglais et sémaphores).	On utilise une bibliothèque Pthreads.
A base de Threads	C'est un type de modèle également capable d'utiliser une mémoire partagée. Un processus « poids lourd » peut avoir de multiples « processus légers », qui s'exécutent ensemble.	On utilise une bibliothèque Pthreads et/ou un compilateur OpenMP.
A mémoire distribuée	Les tâches échangent les données via une communication avec envoi et réception de messages. Elles utilisent des mémoires locales au calcul en cours.	On utilise une bibliothèque telle que MPI qui est le standard "de facto" de l'industrie pour le passage de messages.
Parallélisation de données (PGAS)	Ce modèle est identifié comme étant le modèle PGAS. Il s'agit d'une approche de DSM (« *Distributed shared memory* » en anglais) qui combine les avantages de la mémoire partagée et du passage de messages. Elle supporte une notion de mémoire partagée dans une architecture distribuée. Le modèle PGAS fournit un espace d'adresse global avec un modèle de contrôle de SPMD explicite.	Les implémentations les plus répandues sont : -Co-array Fortran avec un compilateur spécifique -Unified Parallel C - UPC, une extension du langage C avec un compilateur spécifique. -Global arrays avec des bibliothèques C et Fortran. -X10 basé sur un langage de programmation parallèle proposé par l'IBM -Chapel, un projet de langage de programmation parallèle « open source » proposé par Cray.

Hybride	C'est une combinaison des modèles de programmation présentés ci-dessus.	MPI + OpenMP, MPI et sur un système CPU-GPU, MPI + Pthreads, etc.
SPMD et MPMD	-SPMD est un programme qui s'exécute sur de multiple jeux de données, c'est vraiment un modèle de haut niveau d'abstraction. -MPMD correspond au fait d'être capable d'exécuter de multiples programmes avec de multiple données. Comme le ferait un système multitâche classique, c'est aussi un modèle de programmation de haut niveau.	-Le modèle SPMD peut combiner des modèles d'implémentation plus fins tels que ceux décrits ci-dessus. -Tout comme le modèle SPMD, le modèle MPMD peut utiliser tous types d'implémentations sous-jacentes.

Table 3.3. Modèles de programmation parallèle (Diaz et al. 2012) (Barney 2015).

Dans ce tableau, on peut distinguer les modèles de programmation parallèle qui utilisent de la mémoire partagée et ceux qui ont besoin de la mémoire distribuée. Les combinaisons donnent des approches hybrides et il est toujours possible de se placer à un plus haut niveau d'abstraction avec les modèles SPMD et MPMD. Le choix d'un modèle de programmation pour paralléliser un programme dépend principalement du type d'applications.

III - Conclusion

Dans ce chapitre, nous avons présenté l'essentiel des notions utiles à la compréhension des architectures matérielles parallèles et à leur programmation. Dans la première partie, nous nous sommes concentrés sur les concepts et la terminologie du calcul parallèle. Après avoir donné les lois théoriques utilisées pour fixer les limites du calcul parallèle, nous avons présenté la taxonomie classique des architectures parallèles, les méthodes et outils d'analyse des performances, ainsi que les tests qui font actuellement référence pour classer les systèmes parallèles sur la scène internationale. Dans la deuxième partie de ce chapitre, nous avons évoqué les nouvelles approches utilisées dans le domaine du calcul à haute performance en passant en revue, d'une part, les architectures actuellement utilisées massivement (processeurs multi-cœurs, many-cores et GP-GPU), et d'autre part, les accélérateurs d'un nouveau type tels que les FPGA et à plus long terme, les accélérateurs quantiques. Même si cela s'éloigne du cadre de notre thèse, nous avons également évoqué les puces neuro-morphiques qui servent à accélérer les simulations du cerveau. Dans cette partie, nous avons aussi fait une étude plus poussée de l'architecture d'accélérateur à base de processeurs many-cores de type Intel Xeon PHI (ou MIC : *many integrated cores*). Une bonne compréhension des aspects matériels est essentielle pour réaliser des implémentations parallèles efficaces mais, elle est encore plus essentielle à la reproductibilité numérique de l'application.

Chapter 4 - Propositions

I - Introduction

Dans les chapitres précédents, nous avons présenté le contexte de travail de notre thèse. Dans ce chapitre, nous analysons les définitions du domaine et nous évaluons les solutions proposées sous différents aspects : logiciel, matériel, conception des programmes et problèmes mathématiques liées aux opérations en virgule flottante et aux optimisations sur des plateformes différentes de la famille Intel Xeon. Ensuite nous exposons nos propositions pour répondre aux principales problématiques que nous avons identifiées. Tout d'abord, nous proposons une méthode pour détecter la non-reproductibilité et la non-portabilité des générateurs de nombres pseudo-aléatoires dans différents contextes d'exécution. Ensuite, nous présentons et évaluons une méthode statistique pour tester la corrélation entre des générateurs de nombres pseudo-aléatoires utilisés en parallèle. Enfin, nous proposons une procédure pour développer une simulation de Monte Carlo parallèle qui soit numériquement reproductible. Les propositions exposées dans ce chapitre seront mises en pratique dans le chapitre suivant.

II - Approches existantes pour la reproductibilité numérique

II.1 - Des propositions logicielles

Nous présentons des propositions concernant la recherche computationnelle reproductible. Dans ce contexte, il existe des propositions populaires dans la communauté scientifique. On peut citer notament CDE, ReproZip, IPOL, Lepton, ReScience, REP, Docker, Umbrella, plateforme reproductibilité à l'échelle, etc.

Reproductibilité d'expérimentale

Tout d'abord, nous présentons des approches existantes qui sont basées sur la technique de mise en paquet. Guo (Guo 2011) a créé un outil logiciel ouvert qui empaquette automatiquement les codes, les données et l'environnement d'un programme. Cet outil s'appelle CDE (« Code, Data and Environment » en anglais) facilite les installations et exécutions dans des contextes matériels différents sous Linux car il crée un environnement d'exécution indépendant sur les différentes versions du système d'exploitation Linux reposant sur une architecture matérielle x86. CDE capture toutes les données, les codes et l'environnement d'exécution d'un programme, ensuite, il créer un

« package » Linux. On peut utiliser ce package sur autre machine Linux (voir la Figure 4.1). CDE facilite la portabilité des programmes sur divers systèmes d'exploitation Linux. CDE aide à réaliser la reproductibilité d'expérimentale même s'il se heurte à des limitations, notamment sur le partage de bibliothèques statiques. De plus CDE n'offre pas la garantie d'être complet, mais il est une aide réelle dans bien des contextes.

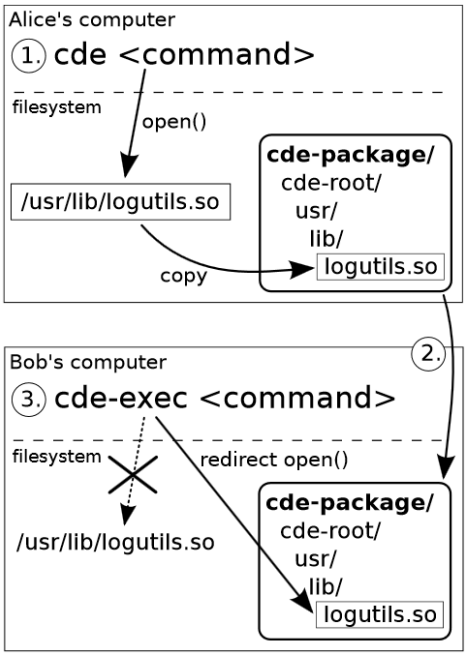


Figure 4.1. Un exemple avec CDE (Guo 2011).

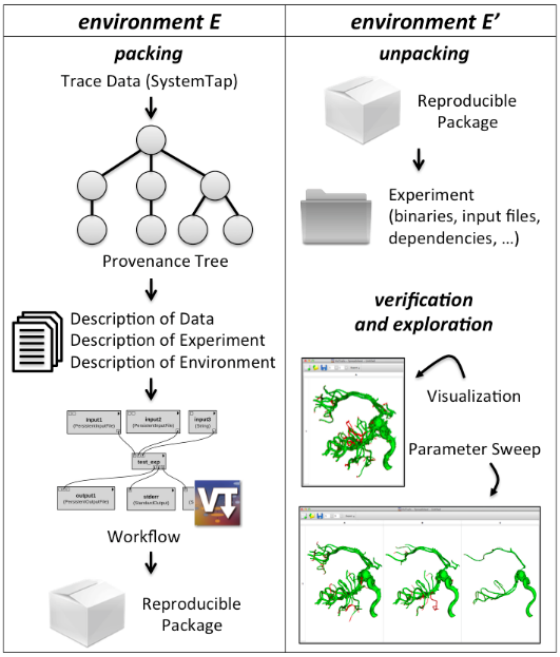


Figure 4.2. Création d'expérience reproductible avec ReproZip (Chirigati et al. 2013)

Chirigati (Chirigati et al. 2013) a également proposé un outil qui se nomme « ReproZip » pour aider à la reproductibilité des expériences. Cet outil vise à capturer systématiquement les détails des expériences existantes, y compris les dépendances de données, les dépendances aux bibliothèques utilisées et les paramètres de configuration. Ces informations sont combinées dans un paquet qui peut être installé et exécuté sur un environnement différent comme indiqué dans la Figure 4.2. Il utilise deux outils libres en « open source » SystemTap et MongoDB pour capturer et stocker les informations nécessaires. Cet outil aide les auteurs à publier des résultats reproductibles et pour des rapporteurs, c'est une aide pour valider les résultats publiés. Il y a cependant des limites annoncées par les auteurs:

In the packing step, ReproZip captures what happens on a run, and therefore, the experiment will be reproducible if the process is « deterministic » - in a case there is a non-deterministic step (e.g., race conditions on the code that may produce different outputs depending on processor speed and system

overhead), it is not possible to guarantee that the original results will be reproduced (Chirigati et al. 2013).

En effet, l'outil ReproZip n'assure pas une reproductibilité numérique des exécutions dans tous les cas. Les principaux écueils sont liés aux aspects non-déterministes, intrinsèques au programme, aux ressources partagés par des processeurs, à la charge de calcul, à la variabilité de la vitesse d'exécution des processeurs etc.

Une autre approche vise à utiliser la technologie de virtualisation. Tout d'abord, nous présentons Docker, projet dont le code est libre et ouvert. Il s'appuie sur de nombreuses technologies familières comme celle des conteneurs Linux, de la virtualisation des processeurs des systèmes d'exploitation, de la gestion des versions du code, etc. (Boettiger 2015). La façon de fonctionner de Docker est similaire à celle d'une image de machine virtuelle. Néanmoins, Docker partage le noyau Linux avec la machine hôte et, on obtient donc une indépendance vis-à-vis de l'environnement d'exécution. Son image en binaire peut inclure tous les logiciels installés, configurés et testés.

Docker has the potential to address shortcomings of certain existing approaches to reproducible research challenges that stem from recreating complex computational environments (Boettiger 2015).

Selon Boettiger (Boettiger 2015), Docker a des atouts pour la recherche de reproductibilité notamment dans sa capacité à gérer les aspects d'environnement d'exécution. Cependant, Boettiger a aussi présenté les limitations de Docker comme l'absence de virtualisation complète, les limites de gestions des machines hôtes de 64 bits et parfois une hétérogénéité observée des résultats en fonction du système d'exploitation.

A partir des obstacles observés au niveau de l'environnement d'exécution, des limitations des machines virtuelles et des containeurs systèmes, Meng et Thain ont proposé Umbrella (Meng and Thain 2015). Selon eux, les approches basées sur les machines virtuelles et des containeurs systèmes résolvent une partie du problème lié à l'environnement d'exécution. Umbrella est un outil qui permet une spécification et une matérialisation des environnements d'exécution du matériel jusqu'aux logiciels en incluant les données.

The organized Umbrella specification is light-weight, and makes an application portable and reproducible. (Meng and Thain 2015).

Umbrella permet de fournir un environnement d'exécution à partir d'une description déclarative fournie par l'utilisateur. Ces informations comprennent : le matériel, le système d'exploitation, le logiciel, etc. De plus, Umbrella est capable de travailler dans des environnements de grille de calcul et/ou d'informatique en nuage. Son architecture est celle d'une plateforme combinant des technologies récentes et d'autres plus anciennes (comme le « sandboxing »), les technologies autour de Docker, chroot, Parrot, Condor et Amazon EC2 comme illustré sur la Figure 4.3.

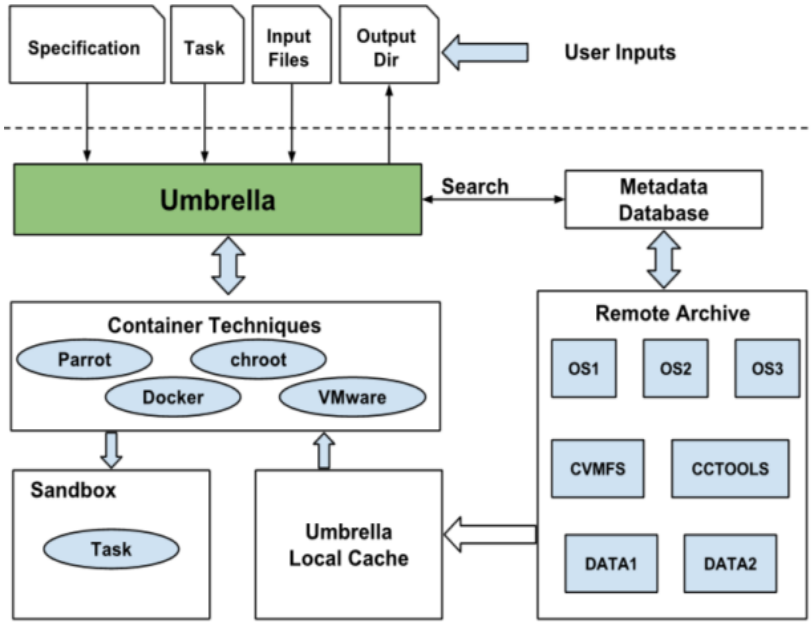


Figure 4.3. L'architecture d'Umbrella (Meng and Thain 2015)

La troisième d'approche identifiée et celle des plateformes conçues pour faire de la reproductibilité. Tout d'abord, nous mentionnons la plateforme REP qui a été proposée par Likhomanenko (Likhomanenko et al. 2015). Selon lui, c'est une infrastructure logicielle complète qui supporte tout l'écosystème collaboratif pour la science computationnelle. Tout comme les outils Docker et Umbrella, cette plateforme est construite en Python.

It is a Python based solution for research teams that allows running computational experiments on shared datasets, obtaining repeatable results, and consistent comparisons of the obtained results (Likhomanenko et al. 2015).

La Figure 4.4 présente l'infrastructure REP, dans cette image on trouve différents composants de cette plateforme, la prise en charge de différentes bibliothèques, la possibilité de calcul distribué sur grille, la gestion des versions des données et du code et des résultats avec « git » et l'utilisation de Docker pour capturer toutes les dépendances.

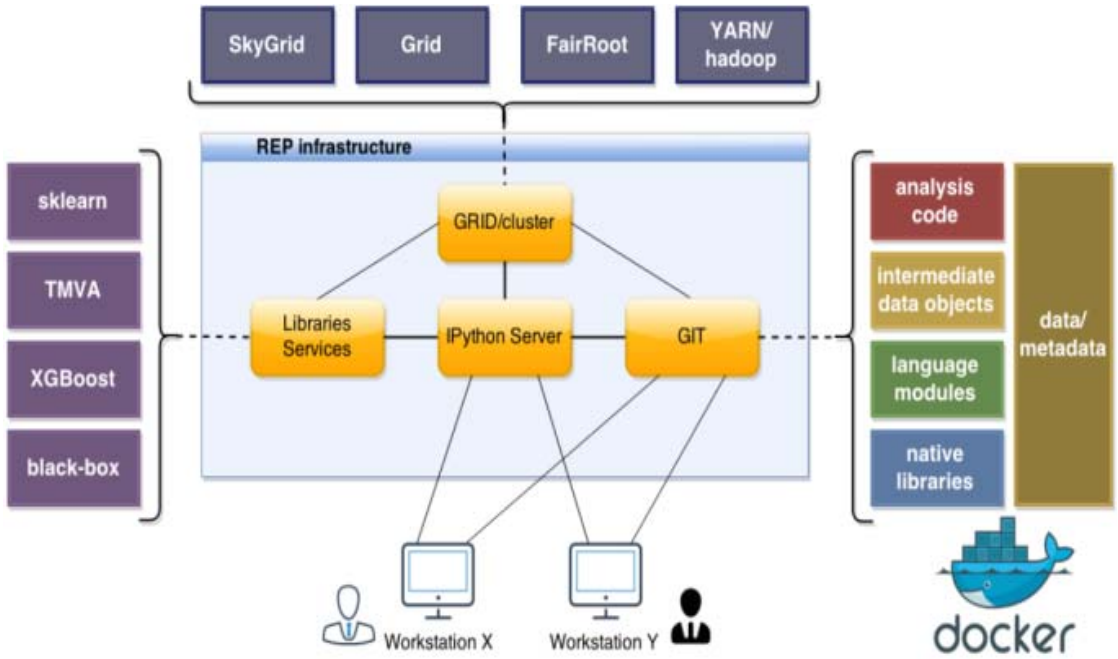


Figure 4.4. L'infrastructure de REP (Likhomanenko et al. 2015)

Une autre plateforme a fait récemment son apparition, il s'agit de Redo, elle est dédiée au calcul à haute performance et a été présentée par Jimenez (Jimenez et al. 2014). L'idée de cette plateforme est basée sur l'évaluation des résultats obtenus qui ont besoin d'une méthode de comparaison.

> *...we present the design of Redo, our reproducibility framework for capturing an experiment and its results, and automating job comparison (Jimenez et al. 2014).*

La plateforme Redo utilise des technologies comme les systèmes de contrôle des versions (avec git), la virtualisation (avec Docker), l'inspection des performances (avec snapshots), etc. Redo utilise le terme « job » ou « job de calcul », le résultat d'un job est non seulement défini par sa sortie, mais aussi par d'autre paramètres tels que: la précision, la performance, la portabilité, la longévité. Redo essaie de capturer l'état de l'environnement scientifique utilisé par agrégation des dépôts individuels et des versions associées. Au sein de cette plateforme, la reproductibilité d'un résultat se compose de 6 étapes différentes :

- Vérification (« checking out » en anglais) un commit de Redo.
- Préparation de l'environnement et des données qui se réfèrent au commit.
- Re-exécution du job.
- La capture et le marquage du résultat généré.
- Le stockage des statistiques de runtime de l'exécution du job.

- L'obtention des métriques qui permettent à l'utilisateur de comparer le résultat obtenu et ses dépendances avec les précédents.

Cette plateforme est automatique dans son processus de capture des données et de comparaison des résultats obtenus.

Nous venons de présenter des solutions pour la reproductibilité des expériences computationnelles utilisant trois approches différentes : la technologie des paquetages, la technologie de la virtualisation et une intégration complète via une plateforme. Ces approches essayent de fournir le plus d'éléments possibles pour permettre de reproduire des expériences computationnelles dans différent environnements d'exécution. Ces approches sont précieuses sur le chemin de la recherche reproductible.

Reproductibilité « des publications »

A côté des outils décrits ci-dessus, nous présentons d'autres outils plus centrés sur la reproductibilité des publications scientifiques.

Parlons tout d'abord de Lepton, un outil créé par Li-Thiao-Té (Li-Thiao-Té 2012) pour fournir un appui dans des tâches quotidiennes de la recherche informatique telles que la programmation ou l'écriture de rapports techniques, le revue et la description des méthodes et des résultats par les collaborateurs et à long terme, la réutilisation des codes sources, des données entrées et / ou des résultats de la recherche.

> *Lepton is an automaton for literate executable papers. It enables researchers to publish their work in the form of a script or program that can generate the research paper along with the corresponding source code, input data and output results. Lepton files do not contain pre-computed results, but the full set of instructions for reproducing the results presented in the manuscript.*
> *Lepton is designed for writing reproducible technical reports during method development, and journal manuscripts when the research is polished (Li-Thiao-Té 2012).*

Cet outil stocke un grand nombre d'informations pour reproduire les résultats. Il fournit aussi des documents exécutables sous la forme d'un manuscrit avec un ensemble d'instructions pour reproduire les résultats scientifiques. On peut voir Lepton comme un README autonome et

automatique, pour un environnement donné. Il apporte un aide à la cohérence et à l'exactitude des résultats ainsi que pour la mise à jour de la documentation.

Limare et Morel ont également présenté un outil pour la reproductibilité de publication en ligne sur le traitement d'image et les algorithmes d'analyse d'images (Limare and Morel 2011). Cet outil s'appelle IPOL (« Image Processing On Line » en anglais) http://www.ipol.im/. Il permet aux scientifiques de vérifier directement les algorithmes publiés en ligne en fournissant une interface d'exécution Web pour une image téléchargée. En d'autres mots, cette approche fournit un environnement d'expérimental direct via le réseau, ensuite, il devient facile de tester et de reproduire les résultats via l'interface Web.

ReScience est un outil similaire à IPOL visant à une utilisation en ligne (Rougier 2015). Néanmoins, la façon de fonctionner de ReScience est différente de celle d'IPOL. En effet, il s'agit d'un journal fonctionnant sur le principe classique de la revue par des pairs http://rescience.github.io/. Il est basé sur Github et permet aux examinateurs de faire une vérification et une validation sur une publication dans un mode collaboratif. Sa philosophie est de promouvoir la reproductibilité et plus encore la répétabilité dans le cadre des expériences computationnelles « Reproducible science is good. Replicated science is better ».

ReScience is a peer-reviewed journal that target any computational research and encourage the explicit reproduction of already published research promoting new and open-source implementations. ReScience lives on github where each new implementation is made available together with explanations (article).
Each submission takes the form of a pull request that is publicly reviewed and tested in order to guarantee that any researcher can re-use it (Rougier 2015).

Cet outil est une nouvelle approche pour des publications scientifiques ouvertes et de qualité dans le domaine de l'informatique computationnelle. Ce type d'outil va devenir très utile pour la communauté scientifique afin d'augmenter la confiance dans les résultats des publications.

Les trois approches différentes ci-dessus visent à obtenir des publications reproductibles pour augmenter la qualité des publications en informatique. Elles cherchent à répondre à la question suivante : « comment reproduit-on une expérience ou une publication d'informatique computationnelle ? ».

II.2 - Bibliothèques mathématiques et reproductibilité

Nous nous intéressons aux principales approches existantes pour prendre en compte les aspects liés au calcul en virgule flottante. La plupart des bibliothèques mathématiques sont basées sur la bibliothèque BLAS (Basic Linear Algebra Subprograms). BLAS fournit un ensemble de fonctions réalisant des opérations basiques d'algèbre linéaire (additions de vecteurs, multiplications de matrices, résolution de systèmes linéaires,…). Les autres bibliothèques ajoutent de nouveaux algorithmes, notamment pour augmenter la précision des calculs et la reproductibilité des calculs numériques en virgule flottante. BLAS est très utilisée dans le domaine du calcul à hautes performances et notamment au sein de LAPACK, de LINPACK). Elle est également retenue par les fondeurs de microprocesseurs comme AMD qui propose la bibliothèque ACML (« AMD Core Math Library » en anglais), et Intel qui propose MKL (« Intel Math Kernel Library » en anglais). Nous allons présenter maintenant plus en détails certaines bibliothèques qui favorisent la reproductibilité numérique.

ReproBLAS

La bibliothèque ReproBLAS a été développée par le groupe d'étude Bebop (« Berkeley benchmarking and optimization group » en anglais) et a été présentée dans l'article (Demmel et Nguyen 2013b). Cette bibliothèque fournit aux utilisateurs une version BLAS séquentielle et parallèle qui garantit la reproductibilité face à différents changements : nombre de processeurs, la partition de données, façon de réaliser les réductions, ordre de calcul des sommes, etc. Les solutions proposées par la bibliothèque ReproBLAS utilisent l'arithmétique exacte, l'arithmétique à virgule fixe (limitant la plage des exposants et la plage des valeurs) et la technique des virgules flottantes indexées. ReproBlas a été développée pour des environnements de calcul allant des ordinateurs portables multi-cœurs à des systèmes hautement parallèles, y compris pétaflopiques, mais elle s'adapte aussi à l'informatique en nuage. ReproBlas a été développée avec 5 principaux objectifs :

1. la reproductibilité d'une exécution à l'autre (dite reproductibilité RUN to RUN) en proposant des calculs indépendant de l'ordre dans lequel les sommes sont effectuées,
2. la précision,
3. un coût arithmétique raisonnable (c'est le coût en temps du calcul et en utilisation de la mémoire),
4. des coûts de communication minimaux (y compris dans le cas d'un grand nombre de processeurs),

5. une facilité d'utilisation pour de nombreux cas applicatifs

Elle est disponible en ligne et on peut la télécharger à partir du site web suivant :
http://bebop.cs.berkeley.edu/reproblas/index.php

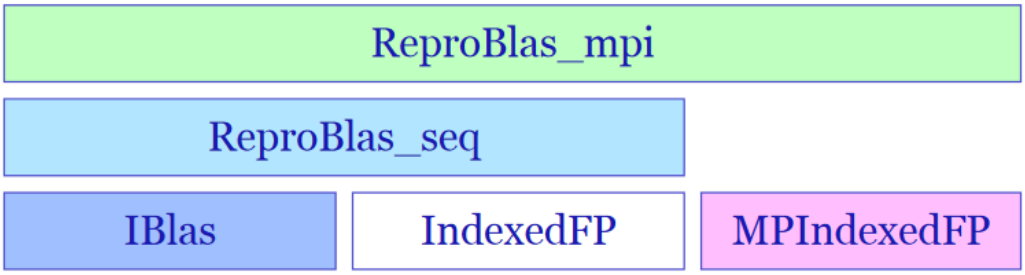


Figure 4.5. La structure de ReproBLAS (Demmel et Nguyen 2013b).

La Figure 4.5 montre que ReproBLAS dispose de 5 modules principaux. Outre le module d'optimisation des performances IBlas, 4 modules sont ajoutés par les auteurs pour l'arithmétique afin d'assurer la reproductibilité numérique pour des applications séquentielles et parallèles avec MPI.

BLAS-1

A partir d'analyses sur les échecs de reproductibilité numériques qui apparaissent dans les calculs massivement parallèles utilisant les virgules flottantes et la norme IEEE-754, Chohra (Chohra et al. 2014) a proposé une solution pour garantir la reproductibilité numérique. Selon lui, il est possible d'étendre l'arrondi correct IEEE-754 pour les grandes séquences de calcul. Cette solution vise à s'appliquer au niveau 1 de la bibliothèque BLAS dont dépend l'ordre des sommes au niveau des fonctions : asum (la somme d'un vecteur en valeur absolue), dot (le produit scalaire) et nrm2 (la norme Euclide vectorisée). Comme la solution proposée touche le 1er niveau des fonctions BLAS, cette version de la bibliothèque s'appelle BLAS-1. En d'autres mots, c'est l'implémentation des algorithmes de sommation, gérant correctement les arrondis dans la bibliothèque BLAS, qui permet d'obtenir des résultats reproductibles (avec les fonctions : Rasum, Rdot, Rnrm2).

ExBLAS

Iakymchuk (Iakymchuk et al. 2015) a proposé des solutions (y compris avec une gestion plus correcte des arrondis et un solveur particulier pour les matrices triangulaires) pour résoudre les problèmes de reproductibilité numérique et de précision des résultats dans le contexte des opérations classiques

d'algèbre linaire comme celles incluses dans la bibliothèque BLAS. L'approche d'ExBLAS intervient à plusieurs niveaux et se déroule en 5 étapes:

1. le filtrage,
2. le super-accumulateur privé,
3. le super-accumulateur local,
4. la réduction parallèle,
5. La gestion des arrondis (Collange et al. 2015).

Les solutions proposées sont appliquées en tant qu'extension des routines BLAS de niveau-1 et de niveau 3. De fait, c'est vraiment une approche multiniveau.

Iakymchuk a aussi présenté une synthèse présentant d'autres solutions et il a analysé leurs limitations pour faire de la reproductibilité numérique. Il cite notamment :

- Les erreurs dues à l'ordre des opérations
- Les erreurs dues à l'arithmétique finie. Néanmoins, cette approche n'est pas portable et ne passe pas bien à l'échelle notamment quand on change le nombre de cœurs pour réaliser le traitement.
- La méthode de Kulisch qui consiste à éviter l'annulation des erreurs d'arrondi utilise une méthode d'accumulation exacte, dite : « super-accumulateur ». Cette solution augmente la précision mais elle se réalise au prix de plusieurs opérations et transferts de mémoire. Elle est trop coûteuse.
- La proposition d'Intel qui réalise de la reproductibilité numérique conditionnelle (« Conditional numerical reproducibility » - CNR en anglais) ne garantit pas une gestion correcte des arrondis. De plus, elle induit selon Iakymchuk de grand coût en termes de performance.
- Les solutions introduites dans l'arithmétique en virgule flottante par Demmel et Nguyen induisent un double ralentissement car les transferts de données et les réductions doivent s'effectuer deux fois.

CADNA

La bibliothèque CADNA (« Control of Accurrancy and Debugging for Numerical Application » en anglais) a été développée au sein du Laboratoire d'Informatique de Paris 6 (Jézéquel et al. 2015). Elle permet de contrôler la précision et le débogage des applications numériques en utilisant l'arithmétique stochastique discrète (« Discrete stochastic arithmetic » - DSA en anglais) en se

basant sur la méthode CESTAC. Cette méthode se base sur une approche probabiliste qui permet d'estimer la propagation des erreurs d'arrondi dans les programmes scientifiques en C++ ou Fortran avec MPI. Cette bibliothèque est disponible en ligne à télécharger http://www.lip6.fr/cadna

When no overflow occurs, the exact result of any non-exact floating-point arithmetic operation is bounded by two consecutive floating-point values R^- and R^+. The basic idea of the method is to perform each arithmetic operation N times, randomly rounding each time, with a probability of 0.5, to R^- or R^+ (Jézéquel et al. 2015).

Cette bibliothèque est utile pour estimer la reproductibilité des programmes numériques qui sont affectés par les erreurs d'arrondi. Ensuite, elle nous aide à expliquer les différences de résultats entre environnements différents. En d'autres mots, elle détecte des instabilités numériques qui se produisent au cours de l'exécution du code. Cette bibliothèque marche bien pour des simulations utilisant aussi bien des CPUs que des GPUs. Néanmoins, il y a des limitations dans l'implémentation séquentielle de la DSA qui font qu'actuellement, celle bibliothèque n'est pas encore assez efficace pour des programmes à très grande échelle. La bibliothèque CADNA a aussi une implémentation adaptée à BLAS pour la multiplication de matrices.

En effet, la plupart de ces bibliothèques visent à améliorer la bibliothèque BLAS en proposant des algorithmes de calcul complémentaires pour mieux gérer l'impact du calcul en virgule flottante pour ce qui concerne la gestion des arrondis afin de garantir la reproductibilité et/ou une meilleure précision numérique. Clairement, ces algorithmes sont actuellement nécessaires et utiles pour les applications de simulations scientifiques en utilisant l'algèbre linéaire

II.3 - Les données scientifiques ouvertes

Une condition préalable pour permettre à quelqu'un de reproduire une simulation est que les données de l'expérimentation computationnelles soient disponibles : les paramètres d'entrée, les données, mais aussi les codes sources, etc. Malheuresement, nous avons vu dans le chapitre 1 que c'est une condition trop souvent insatisfaite.

Dans cette partie, nous présentons quelques approches existantes pour aider à franchir cet obstacle. Nous abordons la notion de norme de reproductibilité, des licences de publication, des sites web en lignes permettant de reproduire des expériences, etc.

Tout d'abord, nous commençons avec une norme pour la recherche de reproductibilité « Reproducible Research Standard – RRS » qui a été proposée par Stodden (Stodden 2009a). Cette norme est un réalignement des droits légaux avec les normes scientifiques et suggère l'utilisation de licences provenant du monde des logiciels libres et ouverts pour changer la culture de publication actuelle. Cette norme aide à la distribution des données pour garantir les droits d'auteurs tout en permettant à l'utilisateur de faire des tests de reproductibilité (comprenant également des droits de copie, de réutilisation, et d'attribution des codes, textes, figures, données, etc.). Cette norme promeut la recherche reproductible de la façon similaire à l'approche utilisée pour la licence publique GNU (GPL) qui est utilisée pour distribuer du logiciel libre.
Parmi les licences recommandées, on peut citer :

- la licence Creative Common d'attribution « CC BY » qui permet d'utiliser librement les textes et les figures, mais qui demande que l'auteur soit cité.
- La licence MIT (ou licence X11 dédiée à la gestion des fenêtres X11) ou similaire pour le code qui permet d'utiliser le code, le copier, le modifier, le fusionner, le publier… mais impose de rappeler le nombre d'auteur et les règles de copyright.
- La licence d'attribution (du type BY dans la licence CC) qui concerne les problèmes de partage de code.
- La licence CC0 (Creative Commons Zero) pour la libération de données qui permet de réutiliser librement ces données, les améliorer, les modifier…dans la limite des lois applicables.

Stodden (Stodden 2009b) a également présenté une licence de recherche ouverte (« Open Research License » - ORL en anglais) pour résoudre les problèmes des droits d'auteur et pour faciliter la recherche reproductible dans le monde scientifique utilisant des approches computationelle. Cette licence mentionne l'article, les données, les expérimentations, les résultats d'expérimentation, les matériaux auxiliaires.

Leveque (Leveque et al. 2012) propose une orientation de la culture de la publication scientifique prenante compte le besoin d'une recherche computationnelle reproductible.

Vandewalle (Vandewalle et al. 2009) a pour sa part proposé des exigences pour une publication reproductible :

- Un référentiel pour chaque publication : c'est une base de données en ligne de publications où les auteurs peuvent télécharger leurs publications en remplissant un formulaire précisant l'ensemble des informations essentielles pour reproduire les résultats de la publication. Vandewalle a aussi mis en avant certains logiciels existants comme : arXiv, le logiciel en source ouverte EPrints, le site web ROMEO (http://www.sherpa.ac.uk/romeo/index.php), etc.
- Dans le cas de références sur des pages web (susceptibles de changer) l'usage des DOI est préconisé (Digital Object Identifier).
- Pour le code, il faut utiliser une licence de code en source ouverte comme les licences GNU GPL, BSD ou similaires. On peut aussi utiliser un site web mettant des informations en « open access » et en utilisant des technologies Open Source comme sur le site de référence PubMed.
- Toutes les données doivent être disponibles.

Vandewalle a souhaité donner un cadre gagnant-gagnant pour la communauté scientifique : avoir accès de plus en plus aux algorithmes et aux codes des publications tout en passant peu de temps à inventer de nouvelles technologies. Il préconise plutôt l'usage des techniques et outils existants.

Il existe aussi des sites et plateformes internet qui permettent de partager les informations, codes et données nécessaires en ligne. On peut citer entre autre, Github (https://github.com/), RunMyCode (http://www.runmycode.org/), Bitbucket (https://bitbucket.org/), SoData (http://sodata.org/), etc.

Il y a de plus en plus de sites web ou d'outils qui aident au partage des données (y compris les codes sources, les données d'expérimentales, les documents, etc.) mais il existe un problème important en ce qui concerne les droits d'auteur. La reproductibilité et le droit d'auteur sont deux notions qui ne cohabitent pas forcément harmonieusement. Néanmoins, pour la recherche de reproductibilité numérique, un partage de toutes les données de publication nous semble vraiment nécessaire et important pour l'avancement de la science.

III - Contributions à la reproductibilité pour la simulation numérique

III.1 - Discussion sur quelques définitions

Tout d'abord, reprenons la définition de la reproductibilité par Demmel et Nguyen. (Demmel and Nguyen 2013) ont indiqué que la reproductibilité numérique consiste en l'obtention de résultats identiques au niveau du bit près (bitwise) d'une exécution à une autre.

Reproducibility i.e. getting bitwise identical results from run to run. (Demmel et Nguyen 2013)

Par rapport à cette définition, Revol et Théveny (Revol and Théveny 2013) ont soulevé la difficulté de définir le résultat et son exactitude. Ils proposent donc une autre définition de la reproductibilité, à savoir l'obtention d'un même résultat lorsque le calcul scientifique est exécuté plusieurs fois, soit sur la même machine, soit sur des machines différentes, avec différents nombres d'unités de traitement, des types, des environnements d'exécution, des charges de calcul, etc. Ces auteurs ont donné des indications sur les notions de reproductibilité et de gestion correcte des arrondis. Dans cette définition, on ajoute le qualificatif numérique, et c'est cette définition que nous retenons depuis le début de ce manuscrit en parlant de « reproductibilité numérique ».

What is called numerical reproducibility is the problem of getting the same result when the scientific computation is run several times, either on the same machine or on different machines, with different numbers of processing units, types, execution environments, computational loads, etc. (Revol et Théveny 2013)

Pour la simulation numérique, il existe différents points de vue :

- Vandewalle (Vandewalle et al. 2009) présente et limite la notion de reproductibilité à la disponibilité de toutes les données d'expérience et à la mise en œuvre par un autre chercheur indépendant capable de reproduire les résultats. Cette idée est similaire à celle initiale de Fomel et Clearbout.

  Bien avant ces travaux, Srinivasan (Srinavasan et al. 1999) présentait la reproductibilité comme la capacité de porter correctement les exécutions des programmes d'une machine sur une

autre. Son point de vue est basé sur la portabilité d'un programme, néanmoins, cette définition ne précise rien sur les résultats obtenus.

- Urbatsch et Evans (Urbatsch and Evans 1999) ont proposé une définition de la reproductibilité dans le contexte du calcul parallèle. Selon eux : un code reproductible est un code de calcul qui donne « exactement » les mêmes résultats avec les mêmes d'entrées quel que soit le nombre de processeurs utilisés pour les calculs. Cette reproductibilité doit donc exister pour les calculs utilisant un nombre différent de processeurs, mais elle doit aussi exister avec le même nombre de processeurs. Cet article insiste également sur le sens du mot « exactement » pour distinguer de l'approche utilisant une « équivalence statistique ».

> *Be reproducibility, we mean that, for the same input, a computer code gives exactly the same answer regardless of the number of processors used for the calculation (Urbatsch and Evan 1999).*

- Récemment, L'Ecuyer a présenté la notion de reproductibilité des résultats comme étant le fait pour une simulation de répéter exactement et de produire exactement les mêmes résultats sur la même machine ou sur des machines différentes. Il a aussi mentionné les types différents d'exécution des simulations, qu'elles soient séquentielles ou parallèles.

> *...reproducibility of the results: it is often required that simulations must be exactly replicable and produce exactly the same results on different computers and architectures, either parallel or purely sequential, and when running the program again on the same computer. (L'Ecuyer et al. 2015a)*

- Hill (Hill 2015) a aussi donné une définition de la reproductibilité dans le domaine du calcul scientifique mais il rappelle l'importante de la notion de « répétabilité numérique ». Le point de vue de cette définition est basé sur la définition d'un programme par Niklaus Wirth. Pour Hill, à la suite de Drummond, la reproductibilité scientifique ne requiert pas la réplication exacte des résultats. L'obtention des mêmes conclusions scientifiques est suffisante en matière de reproductibilité scientifique. Par contre en informatique, il est essentiel d'avoir la répétabilité numérique sans laquelle on ne peut plus mettre au point les programmes. Cette répétabilité fait partie de la reproductibilité au sens large.

In the field of scientific computing, we should focus initially on a subset of reproducibility which is named repeatability – being able to "replicate" the results, i.e. finding the same results with same input data and the same algorithm (Hill 2015).

Pour le contexte de la simulation stochastique parallèle en particulier, Urbatsch et Evans (Urbatsch and Evans 1999) ont présenté une approche pour réaliser ce qu'ils appellent la reproductibilité des codes de Monte Carlo parallèle. Ils insisté sur le mot « exactement » pour préciser qu'il ne s'agit pas d'avoir simplement des résultats statistiquement équivalents. Leur idée de la reproductibilité se résume à la répétabilité numérique : les résultats des codes de Monte Carlo sont indépendants du nombre de processeurs dans leurs versions parallèles. Même si leur définition est vague, c'est la référence la plus ancienne que nous ayons trouvé concernant la reproductibilité numérique pour les simulations parallèles de Monte Carlo. Ils mentionnent déjà les problèmes d'obtention des mêmes résultats d'une exécution à l'autre (dans un même contexte d'exécution).

By reproducibility, we mean that, for the same input, a computer code gives exactly the same answer regardless of the number of processors used for the calculation. We emphasize the word « exactly » to differentiate it from « statistically equivalent ». Determining the statistical equivalence of two samples (which are not exactly the same) requires a statistical study, especially when estimates of variance are unavailable. We also mean to imply that the reproducibility exists not only between calculations of differing number of processors, but also between successive calculations using the same number of processors (Urbatsch and Evans 1999).

Nous présentons aussi notre compréhension de la notion de reproductibilité numérique pour la simulation parallèle de Monte Carlo. Pour nous, il s'agit de répliquer exactement une simulation stochastique d'une exécution à une autre en obtenant des résultats numériques identiques, ceux-ci ne sont pas des résultats statistiquement équivalents, quand on utilise les même entrées (incluant les mêmes flux stochastiques), et ce avec :

- différents environnements d'exécution (systèmes d'exploitation, compilateurs, options ou versions d'un compilateur, matériels, etc.),
- différents types d'exécution et de conception parallèle,
- différents modèles de programmation parallèle,
- différents types d'exécution (séquentiel vs séquentiel, séquentiel vs parallèle, parallèle vs parallèle) qui correspondent aux différentes techniques de parallélisation des flux aléatoires, et ce indépendamment du nombre de processeurs, etc.

- un autre scientifique conduisant l'expérience.

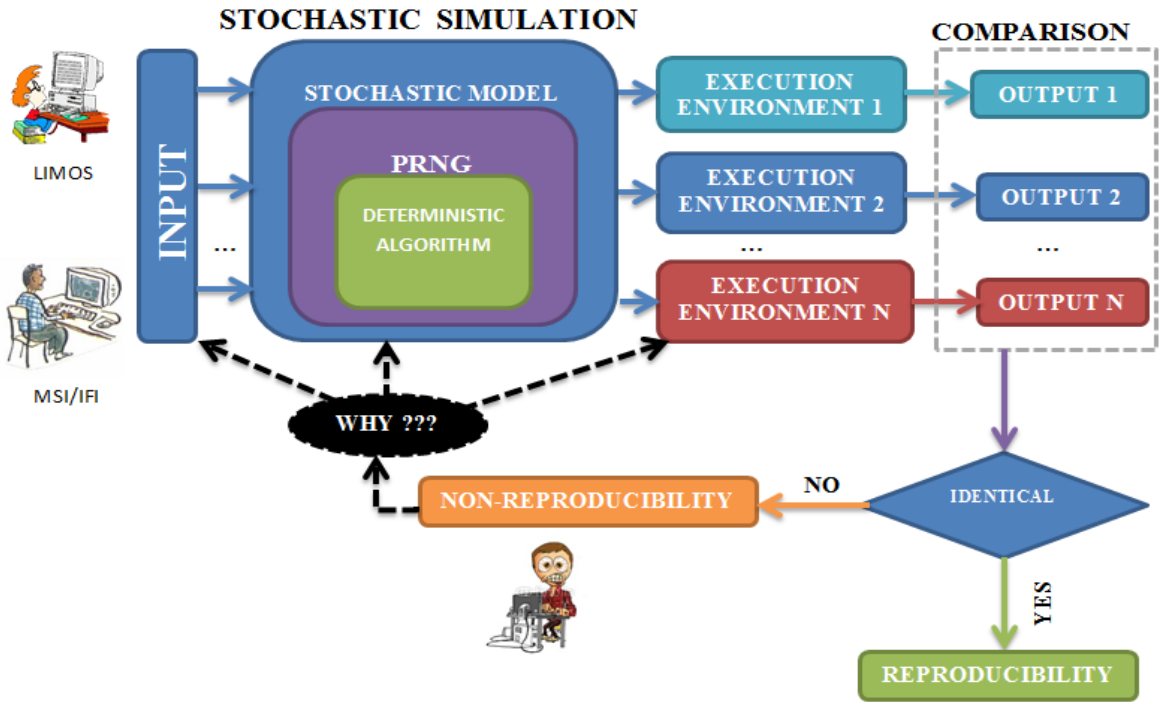


Figure 4.6. Une illustration de la reproductibilité numérique pour la simulation stochastique.

Notre proposition pourrait sembler simple et normale, mais elle est déjà très exigeante dans le contexte des simulations stochastiques parallèles qui utilisent des matériels de calcul à haute performance. Nous présentons une illustration ci-après qui reprend la compréhension que nous avons sur la reproductibilité à partir des définitions ci-dessus.

La figure 4.6 est basée sur toutes les définitions précédentes, néanmoins, dans la pratique, il est complexe d'obtenir les mêmes résultats car les causes de non-reproductibilité évoquées dans le chapitre sont vraiment nombreuses.

III.2 - Distinction entre « reproductibilité » et différents termes associés

La distinction entre la notion de « reproductibilité », et les autres mots cousins ou associés est importante pour mieux comprendre les objectifs de la recherche reproductible dans le domaine computationnel. Ce travail de clarification est nécessaire pour progresser dans ce domaine et améliorer la qualité des résultats produits par la recherche Drummond (Drummond 2009), L'Ecuyer (L'Ecuyer et al. 2015a) et Hill (Hill 2015) ont mentionné la « répétabilité » en lien avec la notion de « reproductibilité ». Fomel et Claerbout (Fomel et Claerbout 2009) ont présenté une relation entre

« reproductibilité » et « réplication ». Srinivasan (Srinavasan et al. 1999) a donné également une connexion entre les notions de « reproductibilité » et la notion plus ancienne de « portabilité ». Néanmoins, tous ces auteurs n'ont pas approfondi la distinction entre ces concepts à l'exception de Drummond.

On trouve également une bonne distinction entre répétabilité, réplication et reproductibilité dans une étude conduit par de Juristo et Gómez (Juristo and Gómez 2012). Dans cette étude, les auteurs ont utilisé la définition de Cartwright et ont également fait une bonne synthèse de la réplication dans le domaine du génie logiciel, mais aussi dans plusieurs autres disciplines.

...Cartwright, for example, suggests that replicability « doing the same experiment again » should be distinguished from reproducibility « doing a new experiment ». For Cartwright, the replication of an experimental protocol used in the previous experiment very closely following the experiment refers to repeating previous experiment, whereas reproduction refers re-examining a previously observed result using a different experimental protocol to what was employed in the previous experiment.
According to Cartwright, replication does not guarantee that the observed result represents the reality under observation. The result can be artefactual, i.e. a product of the materials or the instruments used in the experiment. To guarantee that the result is consistent with the reality under observation, we have to undertake a reproduction using different experimental protocols to ensure that the observed result is independent of the procedure, materials or instruments used in the experiments that arrived at the result.
Based on the different replication types that we have found, replications appear to fall into three groups:
1. Replications that vary little or not at all with respect to the baseline experiment.
2. Replications that do vary but still follow the same experimental protocol as the baseline experiment.
3. Replications that use different experimental protocol to check the baseline experimental results i.e. reproductions. (Juristo, N. and O.S. Gómez 2012).

Cette distinction a bien été reprise par une explication de Nussbaum (Nussbaum 2015). Selon lui, les trois groupes de Juristo et Gomez correspondent aux catégories suivantes:

- Le premier groupe de Juristo et Gómez correspond à la répétabilité: on utilise la même méthode, le même environnement et les mêmes paramètres pour la même expérience. Dans ces conditions, on doit obtenir le même résultat.

- Dans le deuxième groupe: on utilise la même méthode mais, des environnements et des paramètres différents pour la même expérience. Dans ces conditions, on arrive à la même conclusion scientifique.
- Le troisième groupe recouvre la reproductibilité scientifique au sens large : on utilise une méthode différente pour tester la même hypothèse scientifique, et on souhaite la même conclusion scientifique.

Une taxonomie des niveaux de la reproductibilité proposant une distinction entre la reproductibilité et les notions voisines est décrite dans un rapport du workshop ICERM sur la reproductibilité dans les mathématiques computationnelles et expérimentales édité par Stodden (Stodden et al. 2013). La reproductibilité implique différents processus liés à la recherche tels que : la revue par les pairs, la réplication des expériences, la confirmation de résultats, la vérification et la recherche ouverte (avec code, logiciel, données et résultats librement accessibles). Ces niveaux de reproductibilité ont été aussi réexpliqués précisément par Hill pour la reproductibilité des simulations stochastiques parallèles (Hill 2015).

Récemment, Langlois (Langlois et al. 2015) a présenté les notions de reproductibilité, de portabilité et d'exactitude numérique en faisant bien la distinction entre ces notions.

The reproducibility requirement is not a portability issue. Portability problems comes when one source code yields different binaries, i.e. binaires with different numerical properties. In our scope, the main portability parameters are the compilers, their options, the libraries, the few freedom spaces let by the IEEE-754 floating-point standard (rouding modes, double rouding, fused multiply and add operator) and the targeted computing unit. As mentioned before, reproducibility may fail for a given set of these portability parameters.
As already pointed out by [7], the reproducibility requirement is not a claim for accuracy. Numerical reproducibility is getting bitwise identical results for every p-parallel run, $p \geq 1$. Full accuracy, or accuracy up to the computing precision, is getting bitwise exact result. [7] introduces reproducibility but not necessarily accurate summation algorithms. Of course, improving the computed result accuracy up to the IEEE-754 correct rounding guarantees its numerical reproducibility (Langlois et al. 2015).

Selon Langlois, la demande de reproductibilité numérique n'est pas seulement une exigence de portabilité, mais c'est aussi une demande de précision. La ***portabilité*** numérique est associée au fait que le même code source va produire différents binaires dans des contextes matériels différents.

Pour Langlois la portabilité concerne à juste titre les binaires du même code source qui peuvent avoir des propriétés numériques différentes. Il précise que la reproductibilité peut échouer en raison des paramètres de portabilité. Pour Langlois, la ***reproductibilité*** obtiendra des résultats identiques au niveau du bit, mais par contre, **l'exactitude** obtiendra des résultats précis (eux aussi au niveau du bit). Cette distinction entre la reproductibilité et la précision est basée sur l'idée de Demmel et Nguyen (Demmel and Nguyen 2013). On peut effectivement reproduire des résultats faux et imprécis, la bonne nouvelle c'est qu'on pourra au moins déboguer ce « mauvais » programme, mais l'idéal est de pouvoir obtenir et reproduire des résultats les plus précis possibles dans la limite de l'arithmétique flottante par exemple.

It should be noted that reproducibility is not equivalent to accuracy. On one hand, computing a highly accurate result does not guarantee reproducibility, i.e. computed results might be close to the exact result but not identical. On the other hand, a reproducible result might be far away from the exact result (Demmel and Nguyen 2013).

Feitelson (Feitelson 2015) a également présenté des définitions pour le terme « reproductibilité » et d'autres termes proches, ainsi que leurs relations. Il a donné de plus, une définition élargie ou de niveau plus élevé, il s'agit de la notion de « corroboration ».

III.3 - Reprise des différentes notions et de leurs relations

Nous proposons de replacer ces définitions dans le contexte des éléments d'une simulation pour préciser ces notions et leurs relations. Dans notre proposition, une exécution se compose d'un tuple avec les éléments suivants :

- *P* représente un programme, un algorithme ou une méthode implémentée,
- *EE* représente l'ensemble des « entrées » de la simulation,
- *PE* représente l'ensemble des paramètres d'entrée qui présentent tous les problèmes liés à l'environnement d'exécution comme l'architecture de la machine, le nombre de cœurs ou de processeurs, le compilateur, le système d'exploitation, les bibliothèques mathématiques et associés, la version du compilateur, etc.,
- *O* représente les options de compilation,
- *R* est l'ensemble des résultats à la fin des calculs,
- *E* représente une exécution de simulation,
- *C* représente une compilation de simulation,

- *B* représente un binaire compilé, et *RH* est un résultat en haute précision

Nous utilisons ces notions pour clarifier les situations pour mieux tester et/ou déboguer les simulations en vue d'obtenir une reproductibilité numérique.

- ***Reproductibilité ≠ répétabilité***
  - E1 (P, EE, PE, O) → R1 = R2 ← E2 (P, EE, PE, O) : ***répétabilité***
  - E1 (P, EE, **PE1**, **O1**) → R1 = R2 ← E2 (P, EE, **PE2**, **O2**) : ***reproductibilité***
  - Reproductibilité = {répétabilité + changement (PE et/ou O)}

- ***Reproductibilité ≠ réplication***
  - E1 (P, **EE1**, PE, O) → R1 ≠ R2 ← E2 (P, **EE2**, PE, O) : ***réplication***

- ***Reproductibilité ≠ portabilité***
  - C1 (P, EE, **PE1**, **O1**) → B1 ≠ B2 ← C2 (P, EE, **PE2**, **O2**) : ***portabilité***
  - Reproductibilité = {portabilité + (R1 = R2)}

- ***Reproductibilité ≠ précision (« accuracy » en anglais)***
  - R1 = R2 : ***reproductibilité au bit***
  - R1 ~ RH, R2 ~ RH, R1 ≠ R2 : ***précision/exactitude au chiffre***
    - Rouding_correct(R1) = rouding_correct(R2) : ***reproductibilité***

Cette formalisation permet de mieux réaliser les distinctions. En effet, la portabilité et la reproductibilité sont différentes mais pour faire de la reproductibilité numérique d'une simulation, la portabilité est comme une condition nécessaire. Une simulation numérique doit être portable sur plusieurs environnements d'exécution différents et on doit faire une comparaison des résultats obtenus. De même, la reproductibilité numérique et l'exactitude numérique sont aussi des notions différentes mais l'amélioration de la gestion des arrondis est nécessaire pour faire non seulement de la reproductibilité mais pour avoir des résultats précis.

III.4 - Objectifs de la reproductibilité

On peut voir grâce aux sections précédentes qu'il y a beaucoup de définitions différentes pour la notion de reproductibilité, certaines étant plus précises et d'autres plus vagues. Chaque définition comporte un objectif différent qui dépend du domaine de recherche de l'auteur, mais aussi du point

de vue de l'auteur. A partir des définitions précédentes, nous analysons et nous présentons ici les objectifs que nous avons identifiés :

- L'objectif de Fomel et Clearbout est le partage de données afin qu'une autre personne puisse réaliser des tests indépendants et reproductibles.
- L'objectif de Drummond est que les utilisateurs de l'informatique pour leurs recherches fassent la distinction entre la reproductibilité et sa cousine la « répétabilité ».
- L'objectif de Demmel et Nguyen concerne l'amélioration du calcul arithmétique en virgule flottante pour obtenir de la reproductibilité numérique.
- L'objectif de Revol et Nathalide est similaire à celle de Demmel et Nguyen mais il contient une autre idée qui est celle de pouvoir réaliser des tests et de faciliter le débogage.
- L'objectif de Srinivasan est centré sur la portabilité d'une exécution.
- L'objectif d'Urbatsch et Evans est de pouvoir concevoir un code parallèle Monte Carlo qui soit indépendant du nombre de processeurs qui exécute ce code, il souhaite aussi mettre l'accent sur l'obtention de résultats identiques et non pas sur l'obtention de résultats statistiquement similaires qui montre précisément un manque de rigueur.
- L'objectif de L'Ecuyer repose sur un test et un concept de programme reproductible pour la simulation en prenant en considération les cas des codes séquentiels et parallèles.
- Notre objectif est de réaliser la conception d'un programme de simulation stochastique parallèle en intégrant la notion de reproductibilité, pour ensuite être capable de tester cette reproductibilité avec une répétabilité des résultats entre les codes séquentiels et parallèles éventuellement avec des nombres différents de processeurs.

Différentes définitions et objectifs existent mais ils visent ensemble un point commun qui est d'augmenter la qualité et la fiabilité des résultats publiés par des expériences computationnelles.

III.5 - Apports essentiels de la reproductibilité numérique pour le calcul à haute performance

A partir des définitions données ci-dessus et de l'analyse des objectifs présentée dans la section précédente, nous identifions trois apports essentiels de la reproductibilité numérique pour des simulations stochastiques parallèles (Dao et al. 2016).

1. La reproductibilité numérique sert pour déboguer et tester une simulation stochastique parallèle sur différents environnements d'exécution ou différents types d'exécution. Si l'on a les mêmes données en entrée, la même simulation stochastique parallèle et les mêmes conditions de tests alors on doit obtenir des résultats identiques. S'il existe un ou des problèmes alors on doit essayer d'en trouver l'origine pour proposer des corrections. La vérification de cette propriété incombe à l'auteur de la simulation ou de la publication.

2. La reproductibilité numérique constitue un critère pour juger de la pertinence d'une expérience numérique publiée et donc des conclusions qui en découlent. Il appartient aux auteurs de l'expérience de fournir les réponses aux questions suivantes :

   - Est-ce que les simulations permettent la reproductibilité numérique ?
   - Comment reproduire l'expérience de simulation ?
   - Comment évaluer l'exactitude des résultats des simulations, sont-ils crédibles ?
   - Comment évalue-t-on la qualité de résultats de recherche obtenus par le calcul ?
   - Comment peut-on répéter les résultats de ces simulations ?

   Si l'on peut répondre de façon satisfaisante à ces questions, d'autres scientifiques peuvent refaire les expériences et retrouver des résultats identiques à ceux qui sont présentés dans les publications.

3. La reproductibilité numérique permet de promouvoir le développement scientifique. En effet, la reproductibilité n'est pas seulement la répétabilité d'une simulation existante dans différentes conditions d'exécution. Elle implique une nouvelle création. Dans le contexte de la simulation stochastique parallèle, la reproductibilité numérique doit signifier que la simulation peut être refaite avec différents langages de programmation utilisant les mêmes normes de calcul scientifique (notamment IEEE 754, DEC64 ou toute autre norme pour le calcul scientifiques), différents types d'exécution (séquentiels ou parallèles) et éventuellement différents modèles de programmation parallèle, etc. C'est un défi très difficile à relever, dès que la simulation revêt une structure complexe.

En résumé, le premier apport implique la reproductibilité numérique et les autres sont liés aux deux types de la reproductibilité (reproductibilité numérique et reproductibilité d'expérience). Avec les deux premiers apports, on considère que la reproductibilité comprend la répétabilité ou la copie

complète et exacte, mais le troisième inclut l'imitation. Cette capacité d'imitation est très importante pour la simulation stochastique parallèle utilisant des systèmes de calcul à hautes performances car dans la pratique, il est très difficile d'avoir le même système de calcul à haute performance que l'auteur de l'expérience.

Nous constatons qu'il y a plusieurs façons de comprendre le concept de reproductibilité. Néanmoins, dans le domaine de la simulation en général et la simulation stochastique en particulier, on peut voir que la reproductibilité est la répétabilité exacte des résultats numériques, avec des changements éventuels d'environnement d'exécution. Elle suppose actuellement de sérieux efforts, même parfois pour obtenir des résultats identique d'une exécution à l'autre (exécutions aux contextes identiques). La reproductibilité a pour nous les atouts suivants : pouvoir déboguer et tester une simulation (par l'auteur-même); pouvoir juger de la qualité d'une expérience ou d'une publication et avoir une confiance accrue dans les résultats publiés.

III.6 - Proposition pour une exécution reproductible sur les processeurs Intel Xeon multi-cœurs et l'Intel Xeon Phi many-cœurs

Jeffers et Reinder (Jeffers and Reinder 2013) ont présenté des options de compilation pour minimiser la différence entre les résultats obtenus sur les différentes plateformes de famille Intel Xeon. L'utilisation de ces options à un coût sur les performances (Corden et Kreitzer 2010). Sur les deux plateformes, les options suivantes sont préconisées : ***-fp-model precise -fp-model source.*** La 1ère option permet d'arrondir les résultats intermédiaires à la précision définie dans le code source et elle implique également la 2ème option de précision. Cette dernière indique au compilateur de ne garder que les optimisations qui respectent strictement les valeurs (value safe) lors des calculs en virgule flottante.

- Sur l'Intel Xeon Phi, on désactive l'utilisation des instructions Fused Multiply Add (FMA): ***-no-fma***
- Sélectionner l'option de précision pour les fonctions mathématiques (sur les deux architectures) : ***-fimf-precision=high***
- Pour des applications utilisant OpenMP, on effectue des réductions en parallèle **KMP_DETERMINISITIC_REDUCTION=yes**, on utilise l'ordonnancement statique et utilise **OMP_NUM_THREADS** pour fixer le même nombre de threads sur chaque plateforme. Malgré ces options, il se révèle très difficile d'obtenir la répétabilité des résultats avec OpenMp.

- Pour un application C++ en utilisant les Intel Threading Building Blocks (TBB), on utilise la fonction **parallel_deterministic_reduction()** pour obtenir des résultats cohérents sur les deux plateformes, même pour un nombre différent de threads.

Néanmoins, Corden (Corden 2013b) a indiqué une limitation de ces propositions, si une section du code est parfois déchargée sur le MIC et d'autres fois exécutée sur l'hôte Intel Xeon, l'application sera très probablement non-reproductible. Corden a aussi souligné une remarque importante, même quand on prend en compte les conseils ci-dessus.

There is no way to ensure bit-for-bit reproducibility between code executed on Intel Xeon processors and code executed on Intel Xeon Phi coprocessors, even for fixed numbers of threads or for serial code (Corden 2013b).

En effet, les 2 architectures sont différentes, les processeurs Intel Xeon Phi ne sont plus des architectures x86. Même si l'architecture k1Om est très similaire, ces différences sont à la source des limites concernant la répétabilité numérique entre ces 2 architectures. D'autre part, du fait des architectures différentes, certaines implémentations d'optimisations du compilateur ou de fonctions mathématiques sont aussi à l'origine de disparité dans les résutats. A notre avis, il y a également des problèmes qui sont issus du fait que les processeurs Xeon modernes ont un mode d'exécution « out of order » alors qu'actuellement les processeurs Xeon Phi ont conservé des cœurs de calculs qui exécutent les instructions « in order ». Cet état de fait est fortement bénéfique pour une répétabilité numérique entre les différents cœurs d'un ou de plusieurs Xeon Phi, mais elle est certainement une autre raison pour une absence de reproductibilité numérique entre les 2 architectures k1Om et x86.

III.7 - Conception reproductible pour les simulations stochastiques parallèles

Le résultat numérique d'une simulation n'est pas forcément reproductible d'une exécution à l'autre principalement en raison des problèmes de non-associativité dans le calcul en virgule flottante. Cependant, la conception du programme peut améliorer sensiblement ces problèmes de calcul en virgule flottante. Dans cette partie, nous présentons des solutions existantes pour la conception parallèle d'applications stochastiques.

Srinivasan (Srinivasan et al. 1998) a présenté une approche avec le terme de « processeurs virtuels » visant à reproduire des résultats identiques quand on écrit des applications de Monte Carlo parallèle. Selon lui, la reproductibilité n'est pas résolue par celle du générateur de nombre pseudo-

aléatoires. Par ailleurs, la conception paralléle des flux de nombres pseudo-aléatoires en fonction du nombre de processeurs physiques doit aussi être surveillée sans pouvoir à elle seule garantir la reproductibilité des résultats notamment quand le nombre de processeurs change d'une d'exécution à l'autre.

It is difficult to ensure reproducibility if the number of processors changes between one run and the next. This problem cannot be solved by the PPRNG in itself. In order to write a parallel MC applications that gives identical results with a variable number of processors, it is necessary to write in terms of "virtual processors", each virtual processor having its own PRNG. Each physical processor would handle several virtual processors. ***It is unlikely that many programmers would go to this much trouble just to ensure that their code has this degree of portability*** *unless it can be done automatically (Srinivasan et al. 1998).*

Cette approche par « processeurs virtuels » génère des difficultés pour les programmeurs dans la re-conception des codes. L'idée de reproductibilité de Srinivasan est liée à la notion de « portabilité » y compris quand on change le nombre des processeurs physiques qui exécuteront la simulation. Clairement, cette idée est bonne mais elle n'est pas suffisamment pratiquée et Srinivasan se dit lui-même pessimiste sur son application par les programmeurs.

Deux aspects importants sont absents des propositions de Srinivasan pour obtenir des simulations stochastique parallèles reproductibles. Son approche ne mentionne pas la façon de distribuer et d'assigner les flux aux processeurs virtuels (et c'est vraiment un problème à gérer pour maitriser l'état initial de flux dans chaque processeur virtuel), mais elle oublie aussi le problème de l'ordre du calcul en virgule flottantes lors de la phase de réduction.

Dans un rapport technique d'Urbatsch et Evans à Los Alamos (Urbatsch and Evans 1999) concernant la simulation stochastique parallèle de particules, on trouve une proposition d'affectation séquentielle d'un flux de nombres pseudo-aléatoires pour chaque particule. On peut avoir une correspondance avec la proposition de Srinivasan si l'on considère que chaque particule est simulée par un processeur virtuel. Dans le cas de la physique des particules cette affectation d'un flux à chaque particule permet d'aboutir à un calcul identique quel que soit le nombre de processeurs. La phase de réduction des calculs, ne serait-ce que pour le calcul de moyenne, de variance et d'intervalles de confiance, n'est pas mentionnée dans les travaux d'Urbatsch et Evans.

With particle-owned random number streams and an effectively serial assignment of these streams, the foundation is set for reproducible parallel code (Urbatsch and Evans 1999).

Néanmoins, ce rapport évoque les complications qui se présentent pour une reproductibilité des codes parallèles. Ces complications sont liées à l'ensemble échantillonnage, aux erreurs d'arrondi, à la décomposition de domaines, à la répartition des particules sur les différents processeurs.

L'Ecuyer (L'Ecuyer 2015) a également présenté une idée pour faire la reproductibilité d'un programme avec deux versions l'une séquentielle et l'autre parallèle en concevant des flux identiques de nombres pseudo-aléatoires pour les deux cas. Pour L'Ecuyer, il faut savoir combien de nombres pseudo-aléatoires on utilise dans chaque flux et connaître l'ordre d'exécution dans les deux versions séquentielles et parallèles. L'Ecuyer a aussi donné une autre solution plus simple, utilisant les mêmes flux et sous-flux sur les deux versions. A notre avis, cette idée est bonne mais pas complète pour une utilisation réelle parce qu'il n'a pas présenté précisément la manière de distribuer les flux aux codes séquentiels et parallèles. Ensuite, il n'a pas donné la façon de paralléliser le code avec les sous-flux. De plus, même si on connaît ces façons de faire, alors on a besoin de savoir l'état de départ de chaque sous-flux de nombres pseudo-aléatoires, que ce soit pour le code séquentiel ou parallèle. Il faut stocker les états initiaux pour pouvoir les réutiliser dans les versions séquentielles ou parallèles du programme, spécialement dans le cas où le nombre de flux et de sous-flux est très grand.

This is useful for example to ensure that in a parallel version of a program, the streams can be defined in a manner that the random numbers are used in exactly the same way (and the results are exactly the same) as in a sequential version of the same program with a single stream. To do this, one needs to know how many random numbers are used from each stream, and in what order they are used in both the parallel and the sequential versions. (Another (easier) way to obtain the same results on the sequential and parallel computers, is to run the program with the same streams and substreams on both computers, with standard spacings (L'Ecuyer 2015).

Hill (Hill 2015) a proposé une solution en 4 points pour concevoir une version du code séquentiel qui permete une comparaison avec le code parallèle et qui garantit la reproductibilité numérique avec la version du code en séquentiel. D'autre part, cette proposition ne nécessite pas d'avoir à l'avance combien de nombres pseudo-aléatoires seront nécessaires.

- *A process or object oriented approach (particularly for each stochastic objects or process ;)*

- *the use of modern and statistically sound generators (such as those presented previously) ;*
- *the use of a fine parallelization technique adapted to the selected generator,*
- *a parallel design method starting from the design of the sequential program (Hill 2015)*

Dans sa conception, on retrouve le besoin de comparaison entre le code parallèle et le code sériel comme mentionné dans le rapport d'Urbatsch. Hill propose même une conception spécifique du code stochastique séquentiel pour permettre la comparaison avec le code parallèle (4ème point).

Dans la conception de Hill, chaque objet ou processus stochastique possède son unique flux de nombre pseudo-aléatoire avec une identité correspondante. Les flux de nombres pseudo-aléatoire sont générés par de bons générateurs modernes de nombres pseudo-aléatoires qui ont été testés avec l'outil TestU01. Il préconise le choix d'une bonne technique de parallélisation pour distribuer aux éléments de traitement les objets stochastiques avec leurs flux. Ensuite, il préconise une comparaison des résultats obtenus avec le résultat en séquentiel. Clairement, il a proposé un tuple de multi-exigence {*affectation de flux aux objets stochastiques définis, bon générateur de nombres pseudo-aléatoires moderne, bonne technique de distribution, bonne technique d'émulation*}. Néanmoins, cet article n'a pas précisé d'exigence dans l'ordre des calculs pour l'étape de réduction finale afin d'obtenir un résultat identique avec le calcul séquentiel. En d'autres mots, l'ordre du calcul doit être imposé pour obtenir les mêmes résultats avec les deux versions différentes (séquentielle et parallèle). Cette précision a été apportée dans un article à paraître (Hill et al 2017).

Dans cette section, nous avons passé en revue différentes approches existantes pour faire la reproductibilité numérique d'une simulation parallèle stochastique. On peut identifier des similitudes, entre ces approches. Par exemple, un processeur virtuel dans l'idée de Srinivasan pourrait se comprendre comme un objet ou un processus stochastique dans l'idée de Hill, mais l'idée de Hill ne se limite pas aux éléments de calcul ; une loi de distribution ou sa reproduction est considérée comme un objet stochastique tandis que les différentes instances de générateurs (avec leurs initialisations) sont considérés comme des objets stochastiques et ce sans avoir besoin de les affecter à des éléments de calcul. Les idées d'Urbatsch et Evans, de L'Ecuyer ont un point commun avec l'émulation du code séquentiel dans le code parallèle et la façon de distribuer ou d'affecter les flux de nombres pseudo-aléatoires dans l'exécution parallèle. Hill propose pour sa part une conception parallèle du code séquentiel en vue d'avoir sur ce dernier autant de flux stochastiques que dans le code parallèle en se basant sur tous les objets stochastiques. Ces idées sont au cœur de la reproductibilité numérique pour les applications parallèles stochastiques.

IV - Une méthode de détection de la non-reproductibilité et de la non-portabilité des générateurs de nombres pseudo-aléatoires

Comme nous l'avons mentionné dans les chapitres précédents, le générateur de nombres pseudo-aléatoires (PRNG dans la suite) est un module très important pour une simulation stochastique de type Monte Carlo. La non-reproductibilité et la non-portabilité d'un PRNG entrainera de facto la non-reproductibilité de la simulation stochastique. En conséquence, la reproductibilité et la portabilité sont deux caractéristiques exigées pour un PRNG de qualité. Aussi, pour chaque type de PRNG étudié, nous nous sommes posé les questions suivantes :

- Est-il reproductible ?
- Est-il portable ?
- Comment peut-on vérifier ces deux propriétés ?

Pour répondre à ces questions, nous avons comparé les résultats obtenus sur un environnement de référence avec ceux obtenus en lançant le PRNG sur différents environnements d'exécution (en utilisant, bien sûr, les mêmes statuts initiaux que sur l'environnement de référence). Dans le cas où il existait déjà des résultats publiés sur ce PRNG, ceux-ci ont servi de référence. Sinon, l'environnement de référence que nous avons choisi est une machine Intel Core i7 4800MQ, utilisant le système d'exploitation Windows 7, 64 bits, avec le compilateur gcc-4.8.3 64 bits sous Cygwin. Ce choix est arbitraire et un autre environnement de référence aurait pu être choisi.

Avec cette méthode, nous avons considéré la reproductibilité numérique d'un PRNG comme la répétabilité exacte des flux de nombres pseudo-aléatoires dans des contextes différents. Cette répétabilité est directement liée au problème de la portabilité du PRNG. Notre méthode qui sera mise en œuvre dans le chapitre suivant vise à faire varier :

- le processeur ;
- le compilateur ;
- le système d'exploitation ;
- et le fait d'avoir un ou non un environnement virtuel.

V - Évaluation d'une solution statistique de C. Ismay pour tester la corrélation entre PRNGs exécutés en parallèle

Il y a encore seulement quelques années, les seuls tests qui permettaient de détecter des corrélations entre flux internes étaient mono-variés (Srinivasan *et al.* 2003 ; Salmon *et al*. 2011). Ceci signifie que, pour effectuer le test sur plusieurs flux, ceux-ci devaient être assemblés en un flux unique selon diverses méthodes, et c'est la séquence obtenue par cet assemblage qui était soumise à des tests de qualité statistique. La manière dont ces flux étaient assemblés pour construire la séquence finalement soumise aux tests statistiques s'inspirait des multiples techniques de distribution de flux utilisées en pratique pour les simulations parallèles. Cependant, quelle que soit la manière dont ces flux étaient assemblés, cette technique faisait disparaître la multi-dimensionnalité inhérente au parallélisme.

Dans notre état de l'art, nous avons mentionné une nouvelle approche, proposée par Chester Ismay dans l'article (Ismay, 2013). Ce dernier a introduit de nouveaux tests qui sont multivariés, ce qui signifie que s'il y a $p > 1$ flux à tester, ce n'est pas un vecteur de nombres pseudo-aléatoires construit avec ces p flux qui sont soumis aux tests, mais une matrice à p colonnes, où chaque colonne représente l'un de ces flux. C. Ismay a exécuté ces tests avec des flux générés par des PRNGs d'excellente qualité, en l'occurrence Mersenne-Twister (Matsumoto et Nishimura, 1998) et MRG32k3a (L'Ecuyer, 1999), mais aucune corrélation n'a été détectée entre les flux générés. Pour arriver à rejeter l'hypothèse nulle d'une absence de corrélation, C. Ismay a donc généré des flux dans lesquels des corrélations sont introduites de manière explicite. Au vu de ces résultats, nous nous sommes interrogés sur les capacités de ces tests à détecter des corrélations, c'est-à-dire sur leur puissance statistique, qui est la capacité à rejeter l'hypothèse nulle d'une absence de corrélation. Dans cette partie, nous présentons et évaluons donc cette puissance statistique en appliquant les tests de C. Ismay à des PRNGs de diverses qualités statistiques.

C. Ismay a intégré dans l'outil TestU01 plusieurs tests de corrélations « *inter-streams* » (Ismay, 2013) qui sont explicitement multivariés. La structure de données utilisée par tous ces tests est une matrice avec n lignes et p colonnes. Les p colonnes, notées X_1, ..., X_p dans la suite de cette présentation, représentent les p flux de nombres pseudo-aléatoires testés, et le nombre n de lignes de cette matrice correspond à la longueur de ces flux.

Trois catégories de tests ont été mises en œuvre :

1. *Des tests de corrélation par paires* (extension « mcorr ») : ces tests calculent les corrélations par paires entre les colonnes de la matrice. Ils créent trois matrices de corrélation différentes, contenant respectivement le coefficient de corrélation linéaire de Pearson, le coefficient de corrélation des rangs de Spearman et le coefficient de corrélation des rangs de Kendall. Dans chacun des trois cas, des statistiques (*z*-scores) sont calculées avec les $m = p\,(p-1)\,/\,2$ valeurs de corrélations. Ces valeurs suivent approximativement une distribution normale sous l'hypothèse nulle d'indépendance statistique. Les *p-values* sont alors calculées et le test est effectué en utilisant la procédure de Benjamini-Hochberg-Yekutieli (Benjamini and Yekutieli, 2001).

2. *Des tests sur la matrice de corrélation* (extension « mmult ») : Pour ce test, les colonnes X_1, ..., X_p sont préalablement centrées et réduites. Selon l'hypothèse nulle d'indépendance statistique, la matrice de corrélation devrait alors être la matrice d'identité. Une statistique sur le rapport de vraisemblance est calculée et cette statistique converge en distribution vers une loi du chi-deux avec $p\,(p+1)/2$ degrés de liberté.

3. *Des tests sur des séries temporelles multivariées* (extension « mport ») : Les trois tests programmés, Hosking, Li-McLeod et Mahdi-McLeod, sont des tests du porte manteau qui permettent de tester l'hypothèse nulle d'indépendance statistique par rapport à toutes les alternatives qui sont sous la forme d'un modèle VARMA (*Vector Autoregressive Moving-Average*) (Lutkepohl 2005). Cela signifie que les colonnes de la matrice $(X_t)_{1\le t\le n}$ étant considérées comme les vecteurs successifs d'une série temporelle multivariée, l'hypothèse nulle d'indépendance statistique, selon laquelle $(X_t)_{1\le t\le n}$ correspond à du bruit blanc, est testée avec l'hypothèse alternative selon laquelle $(X_t)_{1\le t\le n}$ suit un modèle multidimensionnel (ou vectoriel) autorégressif à moyenne mobile (VARMA). Les colonnes de la matrice sont centrées, donc les vecteurs de colonnes X_t apparaissent comme les résidus du modèle d'ajustement. La statistique de test est ensuite exprimée en fonction de la matrice d'auto-covariance de ces résidus (la matrice de covariance entre X_t et X_{t-l}, avec $1 \le l \le$ lagOrder). Le test de Hosking utilise cette statistique (Hosking 1980) tandis que le test de Li-McLeod utilise une version modifiée de celle-ci (Li and McLeod 1981). La statistique de Mahdi McLeod est une fonction du déterminant d'une matrice par blocs, les blocs étant les matrices d'auto-covariance des résidus (Mahdi and McLeod 2012). Il faut bien noter que, dans le programme

implémenté dans TestU01, la valeur maximale de lagOrder est par défaut le nombre de colonnes de la matrice.

Pour mesurer le niveau de qualité statistique d'un test, nous nous sommes référés aux résultats de TestU01 sur des PRNGs de qualités variables, qui sont consignés dans la table I, pages 28 et 29, de (L'Ecuyer and Simard 2007).

VI - Une méthode pour assurer la reproductibilité des simulations stochastiques parallèles

L'objectif principal de notre thèse est de proposer une stratégie pour développer des simulations stochastiques parallèles capables d'obtenir les mêmes résultats, non seulement sur une configuration matérielle et logicielle donnée, mais aussi capables de reproduire ces résultats sur d'autres configurations. Dans cette partie, nous énonçons des pratiques, présentées sous la forme de quatre recommandations qui permettent de concevoir une simulation stochastique parallèle dont les résultats seront reproductibles. Ensuite, nous proposons une implémentation pratique de cette "**conception reproductible**", en basant sur l'idée présentée dans (Hill 2015). Nous présentons enfin un outil qui permet de lancer une simulation stochastique en parallèle en garantissant la répétabilité numérique des résultats obtenus par rapport à ceux qui seraient obtenus avec une exécution séquentielle, ainsi qu'entre ceux obtenus avec différentes configurations matérielles et logicielles.

VI.1 - Quelques bonnes pratiques

Une simulation stochastique peut être considérée comme une expérience scientifique *in silico*. Ainsi, lorsque l'on conçoit une simulation stochastique parallèle, nous devons nous assurer de la répétabilité et de la reproductibilité numérique de cette expérience. La première condition qui doit être satisfaite par un travail scientifique, en termes de reproductibilité, est de permettre de réaliser exactement l'expérience publiée par l'auteur de ce travail. Si celui-ci met à la disposition de la communauté ses codes sources, ses paramètres d'entrées et les données utilisées, une telle répétabilité peut être envisagée.

Nous énonçons ci-après quatre recommandations :

Recommandation 1 : Pour être en mesure de répéter une expérience de simulation stochastique parallèle, outre la disponibilité d'une configuration informatique identique, nous devons disposer des données fournies par l'auteur. Ces données au sens large se composent :

– des codes sources de la version finale qui a été utilisée pour générer les résultats publiés ;

– des données utilisées (si elles existent) ;

– des paramètres d'entrée ;

– des scripts supplémentaires ;

– des informations sur les ressources utilisées (le matériel, le nombre de processeurs, le logiciel, le système d'exploitation, le compilateur, la bibliothèque de programmation, la machine virtuelle, les paquets supplémentaires) ;

– d'un guide qui explique, étape par étape, le lancement de l'expérience, la description de celle-ci, la correction des bogues et la version du code ;

– des contraintes et dépendances associées à la technique informatique utilisée ;

– des enregistrements de traitement des données (l'outil de calcul, les tableaux de données obtenus, les outils pour créer des images ou des graphiques, s'ils existent) ;

– du modèle de simulation (implémenté en version séquentielle et parallèle), des entrées et sorties du modèle, de la structure des classes, des algorithmes, etc. ;

– d'un descriptif du problème de simulation traité et également de sa mise en œuvre.

Ces données sont nécessaires à une bonne publication scientifique. En d'autres mots, la reproductibilité des expériences numériques doit permettre à une personne du domaine de télécharger toutes les données disponibles en ligne, ainsi que les informations et documents complets concernant l'expérience computationnelle. Dans l'état actuel de la recherche en informatique, cela suppose un changement dans la culture de communication des résultats scientifiques.

Avant la distribution des données et des informations, l'auteur devrait appliquer une norme de recherche reproductible – telle que RRS (Stodden 2009) – pour garantir son droit d'auteur et aussi faciliter la reproductibilité, la réutilisation et la redistribution par des collègues. RRS n'englobe pas seulement les logiciels, mais aussi les procédures et la description des matériels nécessaires pour la réplication des expériences computationnelles. Cette norme assure également que la reproduction peut être effectuée sans faute par les pairs. Il est possible d'utiliser des outils supplémentaires comme : le partage de données en ligne avec des sites internet tels que github.com, bitbucket.org, RunMyCode.org, sodata.org ou d'autres sites similaires. RSS préconise aussi la réplication d'une expérience avec des outils du type de CDE ou ReproZip (Stodden *et al*. 2013).

Après avoir répété une expérience, si l'on obtient les mêmes résultats que l'auteur, les résultats publiés sont considérés comme significativement plus fiables.

Recommandation 2 : Cette recommandation porte sur les techniques nécessaires pour reproduire exactement les résultats d'une simulation stochastique. Elle suppose que les auteurs doivent toujours penser à la reproductibilité lors du développement de simulations.

Les points de vigilance sont :

– la conception du programme pour une comparaison et une reproductibilité des résultats entre les calculs séquentiels et les calculs parallélisés, comme nous le verrons en section 5 ;

– le choix de bonnes bibliothèques et de types de données mathématiques qui supportent une haute précision des calculs, en conformité avec la norme IEEE 754 (Robey 2013) ;

– une attention à l'ordre d'exécution des calculs en virgule flottante ;

– l'utilisation des techniques pour augmenter la reproductibilité de la sommation en virgule flottante, comme celle proposée par Kahn avec la précision composite (Taufer *et al*. 2015) ;

– l'utilisation des meilleurs générateurs de nombres pseudo-aléatoires ;

– l'utilisation de bonnes méthodes d'initialisation de ces générateurs en séquentiel et aussi en parallèle (PRAND, clRNG, SPRNG, etc.) (L'Ecuyer *et al*. 2015) ;

– une assignation rigoureuse des flux de nombres pseudo-aléatoires aux éléments de calcul ;

– la conception d'un programme séquentiel en pensant au parallélisme et à l'indépendance des flux stochastiques pour assurer la comparaison des résultats entre exécutions séquentielle et parallèle. Hormis les phases de réductions qui requièrent un soin particulier, cette approche assure une reproductibilité numérique, même en cas de changement du nombre de processeurs ;

Recommandation 3 : Avant la publication de résultats, il faut être vigilant sur l'environnement d'exécution d'une simulation. Il faut donc lancer son application si possible sur au minimum deux environnements d'exécution différents pour évaluer l'impact de :

– la différence liée au matériel (GP-GPUs, MPPA, Xeon Phi Manycore, etc.) ;

– la différence liée aux compilateurs et aux versions de ces compilateurs;

– la différence liée aux systèmes d'exploitation

– la différence liée aux exécutions sur machines réelles ou virtuelles ;

– la différence liée au nombre de cœurs d'une exécution à l'autre ;

– la différence liée aux options de compilation. On se doit d'utiliser les options du compilateur garantissant l'obtention du maximum de reproductibilité numérique. Par exemple, avec un processeur et un compilateur Intel, l'utilisation des options standards peut fréquemment conduire à des résultats non reproductibles ;

– contrôler autant que faire se peut, les possibilités de calcul "*out of order*". Il est parfois possible de programmer ce contrôle au niveau du jeu d'instruction en assembleur. Par exemple sur les architectures IBM de type 'Power PC', il existe l'instruction machine EIOIO – (*Enforce In Order Input Output*) qui permet de veiller à ce que les accès et le stockage en cache soient effectuées dans la mémoire principale dans l'ordre spécifié par le programme.

Recommandation 4 : Éviter d'utiliser trop de ressources matérielles spécifiques.

Dans le cas où le développement d'une application parallèle nécessite des matériels spécifiques, un dernier point de progression concerne l'aspect de corroboration introduit précédemment. Cela peut supposer :

– d'utiliser des matériels différents et donc des codes logiciels différents ;

– de comparer les résultats pour différentes exécutions parallèles par rapport à une référence séquentielle ;

– d'utiliser différents modèles de programmation ;

– d'utiliser différents langages de programmation.

En résumé, la première recommandation suggère un changement de culture scientifique. La deuxième donne un certain nombre de conseils techniques pour concevoir une application informatique reproductible. La troisième conseille de tester les applications sur au moins deux environnements d'exécution différents et de comparer les résultats obtenus. Enfin, la quatrième recommandation donne des conseils concernant les cas où nous sommes en présence d'une hétérogénéité des ressources matérielles.

VI.2 - Implémentation de simulations stochastiques parallèles reproductibles

L'implémentation proposée ici est basée sur les idées de conception présentées dans (Hill 2015). Elle permet de garantir la répétabilité des résultats issus des simulations stochastiques parallèles de type Monte Carlo.

Cette implémentation se compose de trois étapes principales et d'une étape supplémentaire de tests de la reproductibilité numérique :

- Identifier les objets stochastiques et la manière dont le programme séquentiel utilise le PRNG pour répartir les flux de nombres pseudo-aléatoires entre ces objets, de manière à pouvoir faire une parallélisation du PRNG qui reproduise cette répartition ;
- Choisir une manière d'implémentation de programmation parallèle selon le modèle d'exécution SPMD:
    - Assigner les flux ou sous-flux de nombres pseudo-aléatoires indépendants aux objets ou aux réplications stochastiques en fonction de leur identificateur;
    - Distribuer les réplications ou objets stochastiques aux éléments de traitements avec une attention sur l'équilibrage de charge ;
    - Gérer les processus et threads d'exécution en parallèle.
- Calculer les résultats retournés par les éléments de traitement :
    - Vérifier les résultats intermédiaires retournés qui sont les mêmes entre un programme séquentiel et parallèle ;

- Réordonner les résultats intermédiaires ;
- Faire la réduction ces résultats intermédiaires avec une attention des opérations en virgules flottantes.

- Tester le programme sur plusieurs environnements d'exécution et avec les nombres d'éléments de traitement différents. Cette étape supplémentaire n'est pas une étape de conception, mais elle garantit la reproductibilité en cas de changement d'environnement.

Nous allons maintenant expliquer plus en détails chaque étape de cette conception reproductible :

Identification des objets stochastiques et du mode d'utilisation du PRNG

L'identification des objets ou processus stochastiques et du mode d'utilisation du PRNG dans un programme séquentiel est très importante pour pouvoir développer un programme parallèle qui garantisse la reproductibilité numérique. En effet, une version parallèle d'un programme consiste à exécuter simultanément des versions multiples des mêmes objets ou processus avec leurs propres flux de nombres pseudo-aléatoires. Il y a deux modes d'utilisation du PRNG :

a.1) Chaque objet ou processus stochastique possède son propre PRNG et le statut d'initialisation correspondant (voir la Figure 4.7) ;

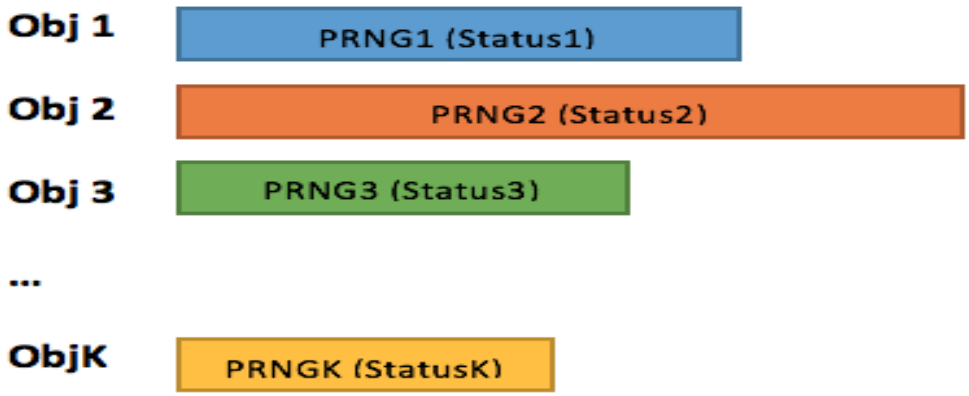


Figure 4.7. Chaque objet ou processus stochastique a son propre PRNG et le statut d'initialisation correspondant. Dans cette illustration, c'est la technique des multiple-flux.

Le « ObjX » avec X=1,2,…k présentent les objets stochastiques, donc Obj1 est le premier objet. De même, le « PRNG1 » est le premier générateur de nombres pseudo-aléatoires avec « Status1 » - le premier statut initial pour le premier objet stochastique. La taille du flux de nombres pseudo-aléatoires utilisée pour un objet est non-déterminée, nous le représentons par un rectangle coloré.

a.2) On utilise un seul PRNG, et chaque objet ou processus possède un statut initial qui lui est propre (voir la Figure 4.8). Il faut faire attention à ce que ce mode d'utilisation du PRNG n'entraîne pas de corrélations entre les sous-flux.

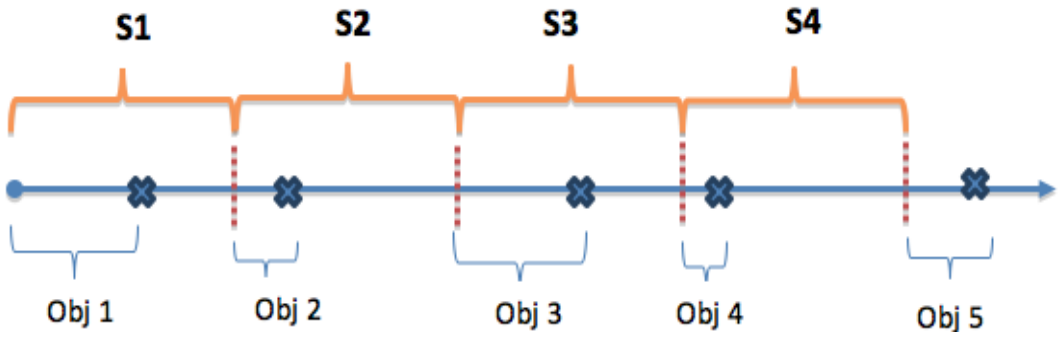


Figure 4.8. Chaque objet ou processus stochastique a son propre statut initial. Dans cette illustration, on utilise la technique de découpage en blocs (*sequence splitting*).

De même que l'explication dans la Figure 0, le « S » présente un sous-flux de nombres pseudo-aléatoires généré à parti d'un statut initial correspondant, donc S1 est le premier sous-flux, SN avec N=1, 2, …k est le Nième sous-flux.

Choix d'un modèle de programmation parallèle

Le modèle d'exécution en parallèle SPMD (*Single Program, Multiple Data*) peut être implémenté en utilisant l'interface de programmation OpenMP, la bibliothèque Pthreads ou la technique MRIP (*Multiple Replications In Parallel*) (Pawlikowski et al. 2002) (Hill *et al*. 2010). En effet, Hill a présenté une même technique parallèle avec la technique MRIP en 1997 mais il n'a pas nommé sa technique (Hill 1997). Le choix d'un modèle de programmation parallèle dépend d'éléments de contexte comme l'implémentation initiale du code en séquentiel, le système matériel utilisé, le temps d'exécution estimé de la simulation, l'équilibrage de charge, etc. L'adéquation entre le modèle de programmation parallèle et le mode d'utilisation du PRNG du programme séquentiel est indiqué dans le tableau 4.1 ci-dessous :

Mode d'utilisation du PRNG	OpenMP/Pthread	MRIP
a.1 – chaque objet ou processus possède son propre PRNG	OUI	FAIBLE
a.2 – chaque objet ou processus possède son propre statut d'un même PRNG	OUI	OUI

Table 4.1. Adéquation entre le modèle de programmation parallèle et le mode d'utilisation du PRNG.

On peut utiliser la technique MRIP pour le mode a.1 mais cela devient vraiment compliqué si le nombre d'objets ou de processus stochastiques est grand. Aussi ne recommandons-nous pas l'utilisation de la technique MRIP avec le mode a.1 d'utilisation de PRNGs. En général, le mode a.1 s'avère un choix difficile à mettre en œuvre.

Les différentes étapes de l'implémentation, en fonction du modèle de programmation parallèle adopté, sont présentées dans le tableau 4.2 ci-dessous :

<table>
<tr><th>Etape</th><th>Description</th></tr>
<tr><td>Assignation</td><td>Les flux ou sous-flux de nombres pseudo-aléatoires sont assignés aux objets ou processus stochastiques en utilisant leur identificateur « IdStatus, idProc ». Ceci garantit le même mode d'utilisation du PRNG entre version séquentielle et parallèle. De plus, les flux de nombres pseudo-aléatoires utilisés par les objets ou processus stochastiques sont les mêmes, quel que soit le nombre d'éléments de traitement.</td></tr>
<tr><td rowspan="3">Distribution</td><td>Les objets ou processus stochastiques sont distribués aux éléments de traitements, en faisant attention à l'équilibrage de charge.</td></tr>
<tr><td>OpenMP ou PThread</td></tr>
<tr><td><u>OpenMP :</u>
La distribution est automatique entre le nombre de threads et le nombre de réplications par OpenMP.
omp_set_dynamic(0);
omp_set_num_threads(NBTHREADS);
…
#pragma omp parallel for reduction(+:total) private(ii)
for(ii = 0; ii < nbRep; ii++)
{
Créer une nouvelle instance stochastique avec une identité précisée ;
Écrire le résultat de son calcul à une variable correspondante par son identité;
total += résultat_instance[ii] ;
}
Supprimer toutes les instances crées;
On peut définir aussi des donnés spécifiques (le nombre de threads, le type d'ordonnancement, etc.) aux plates-formes via une nouvelle variable d'environnement (qui nous utilisons dans le chapitre « Application »).</td></tr>
</table>

	Pthread: Pour transmettre les paramètres d'entrée des threads, on utilise une structure incluant le nombre de réplications de chaque thread (les réplications correspondant au reste de la division nbRep /nbThreads sont ajoutées au thread d'identité 0), l'identité du thread et les paramètres de simulation. Dans un thread du calcul, on a: beginId = threadId * nbRepT ; endId = (threadId + 1) * nbRepT ; avec nbRepT : le nombre de réplications pour un thread. S'il y a le nombre excessif de réplications, threadId = 0, on a : beginId = nbRepT * nbThreads ; endId = beginId + nbRes ; Dans la fonction du calcul principal: Créer une nouvelle instance stochastique avec une identité précisée; écrire son résultat du calcul à une variable correspondant à son identité; faire un calcul en ordre pour chaque thread. Ou, faire un stockage de tous les résultats immédiats. Ensuite, faire une réduction de ces résultats dans le thread principal. pthread_create(threadId[ii],...,&structure[ii]); pthread_join(threadId[ii],&résultat_thread);
	MRIP
	Comme les objets ou processus ont des exécutions indépendantes (simulation de Monte Carlo), ils peuvent, par exemple, être assignés aux unités de traitement avec la technique du tourniquet. S'il y a N_t unités de traitement, les regroupements de N_t objets ou processus traités en parallèle forment des blocs d'exécution. L'exécution des blocs successifs est donc forcément séquentielle : s'il y a N_p objets ou processus stochastiques et si $N_p = N_b \times N_t + T_{res}$, avec $0 < T_{res} < N_t$, il y aura donc N_b blocs de taille N_t et, si $T_{res} \neq 0$, un dernier bloc de taille T_{res}.
Gestion	**OpenMP ou Pthread**
	On fait attention au problème de chevauchement des threads pour l'ordre des opérations en virgule flottante dans le bloc du calcul en parallèle. On fait de l'ordonnancement statique.

	Ou, on peut utiliser aussi des fonctions ***omp_set_lock()*** et ***omp_unset_lock()*** avec OpenMP et ***pthread_mutex_lock()*** et ***pthread_mutex_unlock ()*** avec la bibliothèque Pthreads pour éviter les conflits d'accès aux ressources partagées et faire la synchronisation entre les threads.
	MRIP
	On gère les processus lancés pour éviter un problème dans la gestion des processus (par exemple, le nombre des processus stochastiques créés est égal et supérieur au nombre pid_max par défaut pour un utilisateur du Linux) et la gestion d'allocation de mémoire. En effet, une simulation ne peut pas faire la reproductibilité parce qu'il annule les processus stochastique en raison de problèmes de mémoire ou d'ordonnancement du système d'exploitation.

Table 4.2. Etapes d'implémentation des deux modèles de programmation parallèle.

Réduction des résultats retournés par les éléments de traitement

Avant de faire un calcul de réduction sur les résultats retournés par les éléments de traitement, on doit vérifier les résultats intermédiaires entre un programme séquentiel et parallèle. Ensuite, il convient de fixer l'ordre des calculs pour la réduction des résultats obtenus en parallèle. Pour éviter les problèmes liés à la non-associativité des calculs en arithmétique en virgule flottante, la réduction est exécutée dans un ordre imposé et séquentiel.

Pour l'utilisation d'OpenMP ou de la bibliothèque Pthread, les opérations de réduction se font avec l'option *reduction(+:total)* en OpenMP et avec pthread_join() pour la bibliothèque Pthreads comme nous l'avons présenté dans le Tableau 4.2.

De plus, pour les opérations qui utilisent à la fois des nombres entiers et des nombres réels en virgule flottante, on doit faire attention à la conversion des types de données (en particulier pour la division entre deux entiers ou entre un entier et un nombre réel en virgule flottante). Enfin, comme les résultats des différentes réplications sont écrits dans des fichiers, il faut également faire attention à ce que, au moment des calculs de réduction, les formats en lecture soient concordants avec les formats d'écriture utilisés par les réplications.

Test de la version parallèle créée sur plusieurs environnements d'exécution

Il s'agit d'une étape supplémentaire pour tester la version parallèle qui a été programmée. Elle n'appartient donc pas, à proprement parler, à la procédure de conception reproductible. Elle consiste à vérifier que l'environnement d'exécution et les différentes manières d'utiliser cet environnement n'ont pas d'influence sur les résultats. Pour effectuer ces vérifications, la simulation peut être exécutée :

- Avec deux systèmes d'exploitation différents (architecture 32 bits et 64 bits, Linux et Windows, etc.) ;
- Sur deux architectures matérielles différentes (par exemple, multi-cœurs et many-cores) ;
- Avec deux compilateurs différents ou le même compilateur avec deux versions différentes, ou des options différentes ;
- sur une machine réelle et une machine virtuelle en utilisant les mêmes systèmes d'exploitation et compilateurs.

Commentaires sur les modèles de programmation

Au vu des points soulevés précédemment, nous présentons les points forts et faibles de des modèles de programmation dans le tableau 4.3.

Implémentation	Points forts	Points faibles
OpenMP/Pthread	L'implémentation d'un code en parallèle est plus facile. L'obtention d'une version parallèle se fait avec peu de retouches et de réécritures par rapport au code original séquentiel. Cette approche s'adapte bien aux concepts d'objets ou du processus stochastiques.	La gestion des threads en parallèle sur les calculs en virgule flottante imposent d'être vigilant aux problèmes de « *data race condition* » et de « *deadlock* ».
MRIP	Cette approche est indépendante du code original et ne requiert pas de retoucher le code original.	La gestion des processus peut pénaliser les performances et même empêcher un programme de

	On peut utiliser plusieurs fois cette approche avec plusieurs simulations stochastiques différentes. Elle s'adapte aussi au concept de fonction.	terminer son exécution.

Table 4.3. Une analyse de nos implémentations pour la conception parallèle reproductible d'une simulation stochastique

VI.3 - RSPR : un outil pour lancer une simulation stochastique de type Monte Carlo en parallèle en garantissant la répétabilité des résultats.

RSPR signifie "*Running a stochastic Simulation in Parallel, ensuring numerical Reproducibility*". C'est un outil écrit en C qui permet de lancer une simulation stochastique de type Monte Carlo en parallèle en garantissant la répétabilité numérique des résultats. Nous l'avons conçu en respectant les recommandations d'implémentation qui ont été décrites ci-dessus. La première version de cet outil fonctionne sous le système d'exploitation Linux.

L'architecture de cet outil se compose de trois phases principales (voir la Figure 4.9) :

1. Gestion de tous les flux de nombres pseudo-aléatoires indépendants en appliquant, à l'aide d'un algorithme de *JumpAhead*, la technique de partitionnement en blocs d'un flux du PRNG (*sequence splitting*). Chaque flux correspond à un statut initial précis et est associé à un processus/objet stochastique unique.
2. Parallélisation des processus/objets stochastiques selon le modèle SPMD avec la technique MRIP; le progiciel hwloc (sa description se trouve à la fin de cette section) est utilisé pour l'affinité processeur qui permet de lier un cœur donné (et aussi un élément de traitement ou cœur logique) à une exécution ou processus stochastique. Nous essayons de faire de l'équilibrage de charge, de l'optimisation de performance et un peu de gestion des processus grâce à ce progiciel.
3. Vérification et réduction des résultats obtenus.

Cet outil est appliqué à une simulation stochastique utilisant un seul générateur de nombres pseudo-aléatoires. Chaque objet ou processus stochastique a un statut initial précis. De plus, pour cette première version de l'outil RSPR, nous n'utilisons que le générateur MT19937 auquel David Hill a ajouté deux fonctions, *saveStatus()* et *restoreStatus()*, qui permettent respectivement d'écrire et de lire un statut du PRNG en mémoire. Nous présentons ci-dessous notre algorithme :

Entrée : objet/processus stochastique, N_{rep} - nombre de réplications, N_{co} - nombre d'éléments de traitement, un fichier des paramètres d'entrée.

Sortie : résultat numérique reproductible.

Prérequis : l'objet ou le processus stochastique doit être un programme stochastique séquentiel et le progiciel hwloc doit être installé sur le système avant d'utilisation de RSPR. Dans cette version de l'outil RSPR, nous ne considérons que les éléments de traitement qui sont des cœurs physiques.

Etape1 : Vérifier l'existence des statuts initiaux et/ou les générer

Si les statuts initiaux n'existent pas

Alors on génère N_{rep} statuts initiaux d'un générateur.

Sinon, **si** les statuts initiaux existent **et si** leur nombre est inférieur à N_{rep}

Alors on génère (N_{rep} - le nombre des statuts existants) statuts supplémentaires.

Sinon, passer à l'étape 2.

Etape2 : Effectuer une parallélisation des objets stochastiques

/*Répartir les N_{rep} objets/processus stochastiques entre les N_{co} unités de traitement. N_{co} objets/processus stochastiques exécutés en parallèle forment un bloc d'exécution. Il y a N_{bloc} blocs de taille N_{co} (indicés par une variable IdBloc variant de 0 à N_{bloc}-1), plus un éventuel dernier bloc de taille T_{res} */

$N_{bloc} = N_{rep} / N_{co}$; $T_{res} = N_{rep} \% N_{co}$;

idBloc = 0 ;

/*lancer tous les blocs d'exécution */

Tant que idBloc < Nbloc **Faire**

/* Créer un bloc de N_{co} exécutions en parallèle (voir le code 4.2) */

creerBloc (idBloc) ;

Lancer ce bloc avec son identité idBloc ;

On attend que ce bloc soit terminé ;

idBloc++ ;

Etape3 : Vérifier et réduire les résultats obtenus

Vérifier que le nombre de résultats intermédiaires obtenus est égal au nombre de réplications.
Par exemple pour un calcul de moyenne avant le calcul d'intervalle de confiance

idRep = 0 ;
double average, total = 0. ;
Tant que idRep < N_{rep} **faire** total += total_idRep;
average = total / (double) N_{rep} ;

Algorithme 4.1. Un pseudo code d'implémentation pour l'outil RSPR.

L'algorithme 4.2 consiste à lancer un bloc de processus stochastiques en parallèle. Ce code lance un fichier script exécutable qui contient N_{co} exécutions parallèles en tâches de fond. Nous avons utilisé le progiciel **hwloc** avec l'option par défaut « *logical* ». Cette option assure une assignation de l'exécution des objets stochastiques aux différents cœurs, ce qui équilibre la charge et optimise les performances. L'option « *logical* » est recommandée par l'auteur (Goglin 2015) pour éviter les dépendances au matériel (BIOS, carte mère et ses versions,…) et au système d'exploitation. Ce code permet de prendre en compte des serveurs qui possèdent plusieurs « *sockets* », emplacements où l'on peut mettre des processeurs. Nous avons travaillé principalement sur des serveurs mono socket, bi-sockets mais aussi sur une machine octa-processeur (8 sockets avec 80 cœurs physiques et 160 cœurs logiques).

creerBloc
Entrée : identificateur de bloc (un entier : idBloc) et un fichier des paramètres d'entrée
Sortie : bloc d'exécution des objets/processus stochastiques en parallèle

/* Calculer l'identifiant du 1er processus d'un bloc */
/* Taille de bloc = nb de Coeurs Physiques x nb d'éléments de traitement */
begin = idBloc * Taille de bloc;
increment = 0;
/* N_{PU} = nombre de cœurs logiques */
/* N_{co} = nombre de cœurs physiques */
idPU = idCore = 0 ;

/* lier un processus à un cœur logique */
Si le nombre de « sockets » de la machine est égal à 1

Alors

Tant que idPU < N_{PU} **Faire**

Tant que idCore < N_{co} **Faire**

/*on utilise le progiciel hwloc avec l'option par défaut (« logical »)*/

idRep = begin + increment ;

Lier un processus à un cœur logique précisé avec hwloc (voir le script A.1).

Cette exécution sera faite en arrière-plan.

hwloc socket:0.core:idCore.pu :idPU objet_stochastique(idRep) inFile.dat &

increment ++ ;

Sinon,

IdSoc = 0 ;

Tant que idPU < N_{PU} **Faire**

Tant que idCore < N_{co} **Faire**

Tant que idSoc < N_{soc} **Faire**

/*Calculer l'identité de la réplication dans la relation avec le nombre de sockets*/

idRep = begin + increment ;

Lier une exécution stochastique à un cœur logique précisé avec hwloc.

/*Cette exécution sera faite en arrière-plan*/

hwloc socket:idSoc.core:inCre.pu :idPU objet_stochastique(idRep) inFile.dat &

increment++*;*

Attendre que les processus soient terminés.

Algorithme 4.2. Un pseudo code permettant de créer un bloc d'exécution pour des objets/processus stochastiques en parallèle.

Pour compiler et utiliser cet outil, nous recommandons les commandes suivantes:

```
$ make
$ ./rspr  stochastic_simulation number_cores \ number_replications
input_parametres type_reduction_result
```

où :

- « stochastic_simulation » est une simulation stochastique séquentielle ; son entrée est un identificateur de réplication (idRep) correspondant à un statut précis du flux de nombres pseudo-aléatoires ;

- « number_cores » est le nombre de cœurs physiques ; on peut changer ce nombre mais il doit toujours rester inférieur ou égal au nombre de cœurs physiques dans le système ;
- « number_replications » est le nombre de réplication pour une simulation ; on peut changer ce nombre, mais il doit être égal au nombre de statuts initiaux de flux de nombres pseudo-aléatoires ;
- « input_parameters » est un fichier de tous les paramètres d'entrée pour une simulation.
- « type_reduction_result » permet de faire le calcul final ; on a deux types : « total », par défaut, et « average », pour effectuer une moyenne.

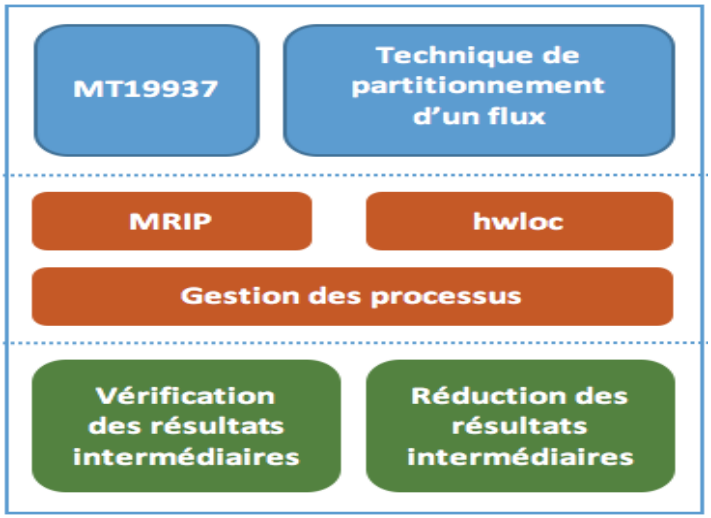


Figure 4.9. L'architecture de l'outil RSPR avec 3 blocs : génération des flux de nombres pseudo-aléatoires indépendants ; parallélisation de la simulation stochastique ; réduction des résultats obtenus.

Le progiciel **hwloc** - « *Hardware Locality* » - est développé dans l'équipe-projet **INRIA Runtime** à Bordeaux en France depuis 2009. C'est un logiciel libre distribué sous la licence BSD. Son objectif est de détecter le matériel et l'exposer de manière portable et générique aux utilisateurs et applications. En d'autres mots, il fournit une abstraction de portabilité de la topologie hiérarchique sophistiquée des architectures modernes incluant les nœuds de mémoire NUMA, sockets, caches partagées, cœurs, threads (Goglin 2014). Il permet de présenter la ressource matérielle via une topologie hiérarchique en mode graphique ou en mode console avec la fonction « lstopo », de calculer le nombre de cœurs, NUMA, threads et socket avec la fonction « hwloc-calc ». Il permet aussi de lier un processus à un ensemble de CPUs avec la fonction « hwloc-bind ». Ce logiciel est meilleur que les autres outils existants dans le système d'exploitation Linux pour lier un processus à un processeur. Le terme « portabilité » signifie qu'il est indépendant du système d'exploitation, du matériel (incluant la carte mère, le BIOS, la version), du vendeur du matériel, etc. Ceci est une des raisons principales pour laquelle nous l'avons intégré dans notre outil « **rspr** ». Dans cette version de l'outil « rspr », nous utilisons la version hwloc-1.11.4.

> Many command-line binding tools exist, including numactl, taskset and schedtool on Linux. But most of them may bind tasks only. Moreover, they only operate on sets of logical processors without any knowledge of processor sockets, caches, etc. Manipulating sets of logical processors unfortunately raises the issue of resource numbering…Applications cannot be portable anymore if they rely on physical resource numbers (Goglin 2014).

Nous avons utilisé l'outil « **rspr** » pour lancer la simulation stochastique démographique dont nous étudierons la pertinence dans le chapitre suivant.

VII - Conclusion

Dans ce chapitre, nous avons présenté la majorité des propositions de cette thèse :

- ✓ Nous avons réalisé une étude des définitions existantes, de leurs nuances et nous avons positionné nos travaux et listé les apports de la reproductibilité numérique que nous avons identifié pour les simulations stochastiques parallèles.
- ✓ Nous avons exposé comment tester, d'une part les propriétés de reproductibilité et de portabilité des PRNGs, et d'autre part la qualité statistique de ceux-ci, lorsqu'ils sont utilisés dans une simulation parallèle. Concernant la qualité statistique, notre proposition est d'évaluer la puissance de tests statistiques multivariés récemment introduits dans TestU01.
- ✓ Après avoir présenté des recommandations pratiques pour assurer la reproductibilité d'une simulation stochastique parallèle qui utilise un système de calcul à haute performance, nous avons proposé une implémentation, détaillée en 3 phases, de la manière de concevoir une simulation stochastique parallèle, en suivant le modèle de programmation parallèle SPMD.
- ✓ Enfin, nous fournissons un outil écrit en C, RSPR, qui permet de lancer une simulation stochastique en parallèle en assurant la répétabilité numérique des résultats. Cet outil montre la faisabilité de la proposition qui est au cœur de cette thèse.

La mise en pratique de ces propositions est présentée dans le chapitre suivant dans lequel nous discutons également de leur efficacité.

Chapter 5 - Réalisations

I - Introduction

Dans le chapitre précédent, nous avons présenté nos propositions en ce qui concerne la reproductibilité numérique des résultats de simulations stochastiques parallèles dans le cadre de l'utilisation de systèmes de calcul à hautes performances (architectures many-cœurs et multi-cœurs). Ces propositions se décomposent en plusieurs étapes : tester la reproductibilité des PRNGs modernes ; évaluer les corrélations entre les séquences générées par des PRNGs parallèles ; concevoir enfin un programme de simulation stochastique parallèle reproductible d'une exécution à l'autre, avec des résultats comparables à l'exécution séquentielle. Dans ce chapitre, nous présentons les résultats obtenus par la mise en pratique de ces propositions.

Processeur	Matériel	OS	Compilateur	Coprocesseurs Intel Xeon Phi
Intel E5-2650v2 Xeon	2.60 GHz 32 CPUs 400 GB Ram	Debian 3.2.39-2 x86_64 GNU/Linux with kernel 3.2.0-4-amd64	- gcc-4.8.2 with 64 bits - Intel C/C++ compiler (Parallel Studio XE) version 14.02	MIC0: 7120P
Intel E5-2687W Xeon	3.10 GHz 32 CPUs 256 GB Ram	Debian 3.2.39-2 x86_64 GNU/Linux with kernel 3.2.0-4-amd64	- gcc-4.8.2 with 64 bits - Intel C/C++ compiler (Parallel Studio XE) version 14.03	MIC0: 5110P MIC1: 5110P
AMD 6272 Opteron	1.40 Ghz 64 CPUs 126 GB Ram	CentOS release 6.4 (Final) Linux version 2.6.32-358.23.2.el6.x86_64	- gcc-4.4.7 (x86_64-redhat-linux) 64 bits - open64-5.0-x_-_64 (gcc version 4.2.0, thread model: posix) - open64-4.1.0-0.i386 (use opencc with gcc version 3.3.1)	non
Intel E7-8870 Xeon	2.40 GHz 160 CPUs (8 nœuds NUMA) 1009 GB Ram	CentOS release 6.7 (Final) Linux version 2.6.32-573.22.1.el6.x86_64	gcc-4.4.7 (x86_64-redhat-linux) 64 bits (thread model: posix)	non
Intel Core Duo T7100	1.80 GHz 2 CPUs 2 GB Ram	Ubuntu Saucy Salamander 13 with kernel 3.8.0-31-generic i686	- gcc-4.8.1 32 bits (i686-linux-gnu) - open64-4.2.1-0.i386 et open64-4.1.0-0.i386 (use opencc with gcc version 3.3.1, thread model: posix)	non
Intel Core i7 4800MQ	2.70 GHz 4 CPUs 16 GB Ram	Windows 7 SP1 OS 64 bits	- Cygwin with gcc-4.8.3 64 bits (thread model : posix) - MinGW with gcc-4.8.2 (thread model win32) - Icc-win64, version 4.1	non

Table 5.1. Les caractéristiques des environnements d'exécution dans (Dao et al. 2014a) et un environnement ajouté Intel E7-8870 Xeon.

La Tableau 5.1 ci-dessus présente les ressources matérielles et logicielles utilisées dans nos expérimentations. Nous avons utilisé deux nœuds de calcul hybrides avec des architectures multi-cœurs Xeon et many-cœurs Xeon Phi : l'un est l'association d'un Xeon Phi 5110P avec deux Xeon E5-2687W standards ; l'autre est constitué d'un Xeon Phi 7120P avec une version assez récente de processeur Intel (Ivy Bridge v2 Xeon E5-2650v2 en 2013). Les drapeaux de compilation sont les mêmes pour les deux architectures, excepté pour le drapeau « *-mmic* » qui est ajouté pour des exécutions sur les processeurs Intel Xeon Phi. Pour le traitement des nombres en virgule flottante, nous avons employé les options de compilation suivantes : « *-fp-model precise* », « *-fp-model source* », « *-no-fma* » et « *-fimf-precision=high* » ; celles-ci garantissent le respect de la norme IEEE 754 pour le calcul en virgule flottante (voir le Chapitre 4) et autorisent de bons niveaux de reproductibilité numérique (Jeffers and Reinders 2013).

Les tests de reproductibilité des PRNGs modernes et les tests de corrélations entre les séquences générées par des PRNGs parallèles vont permettre de comprendre la nécessité d'utiliser des PRNGs de bonne qualité, que ce soit pour une simulation stochastique parallèle ou séquentielle. Ces générateurs influencent fortement la reproductibilité numérique d'une simulation stochastique parallèle exécutée avec un système de calcul à haute performance.

Nos propositions de conception pour obtenir des simulations stochastiques parallèles reproductibles présentées dans le chapitre 4 ont été appliquées à une simulation de Monte Carlo d'un modèle malthusien de croissance d'une population de bactéries. Cette simulation a été testée sur les différents environnements d'exécution présentés dans le Tableau 5.1.

II - Recherche de cas de non-reproductibilité et de non-portabilité avec des PRNGs reconnus comme faisant partie des meilleurs générateurs actuels

Nous avons utilisé des PRNGs actuellement considérés comme faisant partie des meilleurs (statisfaisants les meilleurs tests statistiques de l'outil TestU01, longeur de période, rapidité de génération, germes parallèle, ect.) et mentionnés dans le chapitre 2 : MT19937, avec deux versions 32 bits et 64 bits ; TinyMT avec deux versions 32 bits et 64 bits ; MRG32k3a ; WELL512a ; et MLFG 64 bits. Conformément à nos propositions du chapitre 4, nous avons détecté des cas de non-reproductibilité numérique ou de non-portabilité pour certains de ces générateurs sur plusieurs

environnements d'exécution et avec plusieurs options de compilation. Ces cas sont présentés dans le Tableau 5.2 ci-dessous et ont été publiés dans (Dao et al. 2014a).

Generator	E5-2650v2		E5-2687W		Core 2 Duo T7100		AMD Opteron (TM) 6272		Core i7-4800MQ			
	gcc	lcc	gcc	icc	gcc	open64	gcc	open64	Cygwin	MinGW	lcc	
											lc	lc64
MT19937	Oui	Oui	Oui	Oui	Oui	Oui	Oui	Oui	Oui	Oui	Oui	Oui
MT19937_64	Oui	Oui	Oui	Oui	Oui	Oui	Oui	Oui	Oui	Oui	Oui	Oui
TinyMT_32	Oui	Oui	Oui	Oui	Oui	NON	Oui	Oui	Oui	Oui	Oui	Oui
TinyMT_64	Oui	Oui	Oui	Oui	Oui	Oui	Oui	Oui	Oui	NON	NON	Oui
MRG32K3a	Oui	Oui	Oui	Oui	Oui	Oui	Oui	Oui	Oui	Oui	Oui	Oui
WELL512a	Oui	Oui	Oui	Oui	Oui	Oui	Oui	Oui	Oui	Oui	Oui	Oui
MLFG_64	Oui	Oui	Oui	Oui	N/a	N/a	Oui	Oui	Oui	Oui	Oui	Oui

Table 5.2. Tests de reproductibilité pour 7 PRNGs (MT19937 avec 2 versions, TinyMT avec 2 versions, MRG32k3a, WELL512, MLFG64), sur 5 processeurs différents (Intel E5-2650v2, Intel E5-2687W, Core 2 Duo T7100, AMD 6272 Opteron, Core i7-4800MQ), et avec différents compilateurs (gcc, icc, lcc, open64, MinGW, Cygwin) (Dao et al. 2014a)

Dans le Tableau 5.2, le résultat de chaque test est décrit par un état : « Oui » signifie que les sorties du PRNG testé sont toutes identiques, quel que soit le processeur et le compilateur utilisés ; « Non », qu'un problème de non-reproductibilité est détecté ; et « N/a », dans le cas spécifique du MLFG en version 64 bits, qu'aucun des deux compilateurs gcc et open64 n'a pu générer l'exécutable sur le processeur Core 2 Duo T7100. Dans ce dernier cas, nous pouvons dire que la portabilité du PRNG n'est pas réalisée. Pour notre étude, nous avons envisagé quatre contextes d'exécution différents :

- utilisation de compilateurs différents sur les mêmes architectures matérielles et avec les mêmes systèmes d'exploitation ;
- utilisation de deux versions différentes (32 bits et 64 bits d'un même compilateur), sur les mêmes architectures matérielles et avec les mêmes systèmes d'exploitation ;

- utilisation de deux machines virtuelles avec des systèmes invités différents, ou bien d'une machine virtuelle et d'une machine réelle avec le même système d'exploitation ;
- utilisation du même compilateur, mais sur des architectures différentes.

Une autre situation correspond au cas d'architectures différentes utilisant le même compilateur ; enfin, le dernier contexte d'exécution correspond au cas où l'on utilise une machine « virtuelle » et une machine « réelle ». De plus, nous avons aussi effectué une étude d'évaluation des performances de ces générateurs.

II.1 - Etude avec des compilateurs différents sur les mêmes matériels et systèmes d'exploitations

Ce contexte d'exécution a mis en évidence des problèmes avec les deux versions 32 bits et 64 bits de TinyMT, lorsqu'il est exécuté sur des architectures et avec des systèmes d'exploitation identiques, mais compilé avec des compilateurs différents.

Dans le tableau 5.3 ci-dessous, la colonne de gauche présente les résultats fournis par l'auteur et la colonne de droite ceux que nous avons obtenus avec la version 32 bits du générateur TinyMT, compilé avec open64-i386 sur le processeur Core 2 Duo T7100 sous le système d'exploitation Ubuntu-13.04.

Résultats attendus (CHECK32.OUT.TXT)	Résultats obtenus avec Open64 i386
0.571442**3**	0.571442**2**
0.742153**2**	0.742153**3**
0.663808**5**	0.663808**6**
0.433442**2**	0.433442**1**
0.125419**0**	0.125418**9**
0.468857**8**	0.468857**9**
0.267591**1**	0.267591**0**
0.178412**7**	0.178412**8**

Table 5.3. Résultats de TinyMT_32 compilé avec open64-i386 sur le processeur Core 2 Duo T7100 sous Ubuntu-13.04 (Dao et al. 2014a)

Nos résultats montrent qu'il y a des différences au niveau des 6ème et 7ème décimales entre les résultats obtenus en compilant avec open64 et ceux obtenus par l'auteur : ces différences apparaissent sur les valeurs retournées par les fonctions *tinymt32_generate_float()* et *tinymt32_generate_floatOC()* (qui sont des nombres réels *r* de type « float » avec $0.0 \leq r \leq 1.0$). Il est à noter que lorsqu'on compile avec gcc, il n'y a aucune différence entre les résultats que nous avons obtenus et ceux qui sont proposés comme référence par l'auteur.

Les mêmes problèmes apparaissent avec TinyMT_64. Le tableau 5.4 présente les résultats fournis par TinyMT_64, compilé avec MinGW sur le processeur Core i7-4800MQ sous Windows 7 64 bits. Ceux-ci sont différents des résultats fournis par l'auteur.

Résultats attendus (CHECK64.OUT.TXT)	Résultats obtenus avec MinGW gcc
1.152012609994736	1.152012609994737
1.363201836673650	1.363201836673651
1.218170930629463	1.218170930629464

Table 5.4. Résultats de TinyMT_64 compilé avec MinGW sur Core i7-4800MQ sous Windows 7 64 bits (Dao et al. 2014a)

Le compilateur MinGW, (qui porte deux versions gcc-4.6.4 et gcc-4.8.2) a engendré une erreur sur la 15ème décimale des résultats de la fonction *tinymt64_generate_double12()* qui retourne un nombre réel *r* de type « double » avec $1.0 \leq r \leq 2.0$. Les formats d'affichage ayant été soigneusement sélectionnés à partir des fichiers de sortie, l'affichage différent n'est donc pas une simple erreur d'arrondi. Toujours pour TinyMT_64, nous avons fait le même test avec le compilateur Cygwin, (qui porte gcc-4.8.3) et celui-ci a fourni des résultats reproductibles.

En conclusion, nous avons constaté que, en fonction des compilateurs utilisés, la « répétabilité » de TinyMT, dans ses deux versions 32 bits et 64 bits, n'est pas strictement garantie.

II.2 - Problèmes de compatibilité entre les versions 32 bits et 64 bits d'un même compilateur

Nous avons détecté des problèmes de portabilité pour la version 64 bits de TinyMT, lorsque celle-ci est exécutée sur une machine Core i7-4800MQ sous le système d'exploitation Windows 7 64 bits et

compilée avec lcc-win32, version 32 bits de LCC pour Windows. Ces problèmes sont apparus pour les fonctions *tinymt64_generate_double()*, *tinymt64_generate_doubleOC()*, *tinymt64_generate_doubleOO()* et *tinymt64_generate_double12()* qui retournent des nombres réels *r* de type *double*, avec $0.0 \leq r \leq 1.0$ pour les trois premières et $1.0 \leq r \leq 2.0$ pour la dernière.

Le tableau 5.5 donne, pour chacune des fonctions ci-dessus, un résultat particulier qui montre que les résultats obtenus par l'auteur et les nôtres sont complètement différents.

Résultat attendus (CHECK64.OUT.TXT)	**Résultats obtenus avec le compilateur lcc 32 bits**
0.125567123229521	0.514472427354387
1.437679237017648	1.386730269781771
0.231189305675805	0.112526841009551
0.777528512172794	0.197121666699821

Table 5.5. Les résultats pour le générateur TinyMT_64 sur Core i7-4800MQ utilisant Windows 7 64 bits avec le compilateur lcc 32 bits (Dao et al. 2014a)

Clairement, la compilation avec lcc 32 bits a changé totalement les résultats obtenus normalement par TinyMT_64. La conclusion ici est que la cause de non-reproductibilité numérique pour le générateur TinyMT 64 bits est l'incompatibilité avec le compilateur lcc 32 bits.

II.3 - Utilisation d'une machine réelle et d'une machine virtuelle

Dans cette partie, nous avons trouvé, pour TinyMT_32, des problèmes de reproductibilité entre des machines virtuelles qui exécutent le même type de système d'exploitation Ubuntu mais avec des versions différentes (13.04-i686 et 14.04-i686). Nous avons également comparé les exécutions sur une machine virtuelle et sur une machine réelle avec le même type et la même version de système d'exploitation. Ces machines virtuelles ont été exécutées avec l'outil Virtual Box sur le système hôte Windows 7 et sur une machine hôte Intel (R) Core (TM) i7-4800MQ avec un ou deux cœurs et 2 Go de mémoire vive.

Les résultats obtenus avec ces machines virtuelles diffèrent, en fonction du système invité (Ubuntu 13 ou 14) au niveau de la 7ème décimale (cf. Tableau 5.6). De plus, ces différences sont observées quel que soit le compilateur utilisé, open64-4.2.1 ou open64-4.1.

Résultats attendus (CHECK32.OUT.TXT)	Résultats obtenus sous Ubuntu 13 avec Virtual Box	Résultats obtenus sous Ubuntu 14 avec Virtual Box
0.6455914	0.6455913	0.6455913
0.9415597	0.9415598	0.9415598
0.9034473	0.9034472	0.9034472
0.9348063	0.9348064	0.9348064
0.7581965	0.7581964	0.7581964

Table 5.6. Résultats de TinyMT_32 compilé avec open64-i386 sur des machines virtuelles ayant comme système invité Ubuntu-13.04 et 14.04 (Dao et al. 2014a)

Nous avons ensuite comparé les résultats obtenus entre une machine réelle Intel Core Duo T7200 (voir le tableau 5.3) et ceux obtenus avec les machines virtuelles (voir le tableau 5.6) avec le même système d'exploitation Ubuntu 13.04 et le même compilateur, open64-4.2.1, et nous avons obtenu aussi des résultats différents. Étonnamment, en utilisant la version précédente du compilateur, open64-4.1, les résultats sont identiques.

Un problème similaire arrive lorsque nous exécutons une machine virtuelle avec le système invité Windows 7 et lorsqu'on compare les résultats avec ceux que l'on obtient avec une machine réelle fonctionnant sous Windows 7. Dans ce cas, quel que soit le compilateur utilisé (MinGW et lc 32 bits), les résultats sont différents.

En conclusion, la non-reproductibilité entre machines virtuelles avec différents systèmes invités ou entre une machine virtuelle et une machine réelle utilisant le même système d'exploitation dépend de facteurs divers (système invité, compilateur, ...). En ce qui concerne TinyMT_32, à l'exception d'un seul cas (machines virtuelles sous Ubuntu 13.04 et compilation avec open64-4.1), la reproductibilité n'a jamais été obtenue.

II.4 - Utilisation du même compilateur, mais sur des architectures différentes

Nous avons enfin testé la reproductibilité des différents PRNGs en les exécutant sur des plates-formes différentes, de la famille Intel Xeon, mais en les compilant avec les mêmes compilateurs.

Dans ce contexte, nous avons observé que tous les PRNGs sont reproductibles sur les architectures MIC à notre disposition (7120P et 5110P) et sur les Intel Xeon classiques comme le E5-2650v2 et le E5-2687W (voir les détails de ces architectures dans la section précédente). Nous avons testé les différentes options de compilation (concernant la précision numérique et l'optimisation), et les résultats sont présentés dans le tableau 5.2 : pour toutes les architectures multi-cœurs et many-cœurs, les résultats sont reproductibles.

II.5 - Evaluation des performances

Nous avons mesuré le temps d'exécution nécessaire pour produire 10^{10} nombres pseudo-aléatoires afin d'évaluer les performances des différents PRNGs étudiés sur des machines différentes et des architectures différentes. Nous avons d'abord effectué nos évaluations sur deux architectures Xeon classiques, avec, pour chacune, deux compilateurs différents (gcc et icc) et différentes options. Ensuite, ces évaluations ont été répétées sur deux Intel Xeon Phi avec le compilateur icc et différentes options.

Intel Xeon classiques avec deux compilateurs et différentes options

Nous présentons les temps de génération de 10^{10} nombres pseudo-aléatoires pour différents PRNGs exécutés sur des Intel Xeon E5-2560v2 et E5-2687W avec le compilateur gcc (cf. tableau 5.7) et avec le compilateur icc (cf. tableau 5.8 pour l'Intel Xeon E5-2687W et tableau 9 pour l'Intel Xeon E52687W). A chaque fois, la meilleure performance obtenue apparaît en caractères gras.

PRNGs	Intel E5-2650v2 avec gcc			Intel E5-2687W avec gcc		
	Sans optimisation	**-O2**	**-O3**	**Sans optimisation**	**-O2**	**-O3**
MT19937	2min15.530s	0min54.753s	0min51.465s	2min50.805s	1min0.135s	0min58.117s
MT19937_64	2min24.910s	1min12.881s	1min9.923s	3min0.397s	1min19.869s	1min17.789s

TinyMT_32	3min15.808s	0min51.680s	0min51.674s	4min2.306s	0min58.793s	0min58.778s
TinyMT_64	5min12.615s	1min6.077s	1min5.974s	6min6.762s	1min25.447s	1min25.446s
MRG32K3a	6min11.561s	4min32.556s	3min17.874s	7min9.984s	5min19.870s	4min3.637s
WELL512a	**2min5.294s**	0min53.381s	0min53.160s	**2min31.747s**	1min6.317s	1min6.419s
MLFG_64	2min31.271s	**0min32.969s**	**0min32.790s**	3min1.645s	**0min42.139s**	**0min42.240s**

Table 5.7. Performances de différents PRNGs sur des Intel Xeon E5-2650v2 et E5-2687W avec le compilateur gcc.

PRNGs	**Intel E5-2650v2 avec icc**					
	Sans drapeau de compilation autres que ceux utilisés pour l'optimisation			**Avec des drapeaux de compilation (-fp-model precise et –fp-model source)**		
	-O0	**-O2**	**-O3**	**-O0**	**-O2**	**-O3**
MT19937	2min57.284s	**0min25.237s**	**0min25.471s**	2min57.635s	**0min25.472s**	**0min25.510s**
MT19937_64	3min4.160s	0min57.667s	0min59.536s	3min5.802s	1min2.699s	1min2.354s
TinyMT_32	3min30.050s	0min50.876s	0min50.714s	3min30.364s	0min51.030s	0min50.898s
TinyMT_64	5min23.928s	1min3.735s	1min3.792s	5min25.571s	1min3.459s	1min3.328s
MRG32K3a	6min13.521s	3min6.021s	3min6.449s	6min13.518s	3min5.349s	3min5.061s
WELL512a	2min36.023s	0min55.968s	0min55.135s	2min36.211s	0min55.117s	0min54.832s
MLFG_64	**2min18.396s**	0min40.573s	0min43.543s	**2min19.052s**	0min43.474s	0min43.759s

Table 5.8. Performances de différents PRNGs sur Intel Xeon E5-2650v2 avec le compilateur icc.

PRNGs	Intel E5-2687W avec icc					
	Sans drapeau de compilation autres que ceux pour l'optimisation			Avec des drapeaux de compilation (-fp-model precise et –fp-model source)		
	-O0	-O2	-O3	-O0	-O2	-O3
MT19937	3m42.637s	**0m29.124s**	**0m29.123s**	3m42.653s	**0m29.154s**	**0m29.153s**
MT19937_64	3m46.003s	1m6.198s	1m6.383s	3m44.966s	1m6.202s	1m5.610s
TinyMT_32	4m7.515s	0m58.904s	0m58.903s	4m7.697s	0m58.892s	0m58.893s
TinyMT_64	6m17.400s	1m23.629s	1m23.629s	6m16.740s	1m23.649s	1m23.647s
MRG32K3a	7m25.101s	3m45.354s	3m45.370s	7m3.883s	3m44.058s	3m44.041s
WELL512a	**2m52.199s**	1m8.855s	1m8.854s	**2m51.258s**	1m8.965s	1m8.964s
MLFG_64	2m56.210s	0m44.797s	0m47.378s	2m55.970s	0m49.052s	0m48.186s

Table 5.9. Performances de différents PRNGs sur Intel Xeon E5-2687W avec le compilateur icc.

A partir des tableaux 5.7, 5.8 et 5.9, on peut voir que les performances des PRNGs sur l'Intel Xeon Ivy-bridge E5-2650v2 sont meilleures que sur l'Intel Xeon E5-2687W dans tous les cas. Pour la plupart des PRNGs, lorsqu'il n'y a aucune optimisation, les performances sont meilleures avec le compilateur gcc qu'avec le compilateur icc. Néanmoins, lorsque les optimisations sont utilisées, le compilateur icc est souvent significativement plus rapide avec les optimisations maximales –O2 et –O3. On note notamment les excellentes performances du générateur Mersenne Twister avec le compilateur Intel, avec un speedup atteignant presque un facteur 7 par rapport à celui de L'Ecuyer. Pour des applications de calcul à haute performance, cet écart est très significatif, notamment pour des applications telles que celles de nos collègues en physique où près de 50% du temps de calcul est lié au générateur (travaux de la thèse de Vincent Français à paraître en 2017).

Enfin, comme cela est prévisible, si les drapeaux de compilation nécessaires pour respecter la norme IEEE 754 (*-fp-model precise et –fp-model source*) sont choisis, les performances sont très légèrement moins bonnes.

Intel Xeon Phi avec le compilateur icc et différentes options du compilateur

Pour les architectures Intel Xeon Phi (7120P et 5110P), à options et drapeaux de compilation fixés, nous observons que les temps d'exécution pour le 7120P sont inférieurs à ceux du 5110P, ce qui était prévisible au vu des fréquences de ces processeurs (1.44 GHz et 1.053 GHz, respectivement).

PRNGs	Intel 5110P avec icc					
	Sans drapeau de compilation			Avec des drapeaux de compilation (-fp-model precise et –fp-model source)		
	-O0	-O2	-O3	-O0	-O2	-O3
MT19937	49min43.45s	**7min 38.15s**	**7min 57.28s**	49min43.34s	**7min 38.22s**	**7min 57.28s**
MT19937_64	50min30.82s	16min51.75s	16min51.71s	50min31.33s	16min51.92s	16min51.67s
TinyMT_32	48min46.53s	10min31.14s	10min31.05s	48min46.42s	10min31.05s	10min31.03s
TinyMT_64	49min53.47s	11min47.52s	11min47.51s	49min53.14s	11min47.51s	11min47.81s
MRG32K3a	2h 5min 02s	1h 0min 41s	1h 0min 41s	1h49min 44s	1h13min 16s	1h13min 17s
WELL512a	55min37.32s	15min17.89s	15min18.47s	55min37.96s	15min18.50s	15min17.86s
MLFG_64	**40min57.91s**	14min28.02s	14min27.88s	**40min58.12s**	14min27.89s	14min27.86s

Table 5.10. Performances de divers PRNGs sur une architecture Intel Xeon Phi 5110P avec le compilateur icc.

PRNGs	Intel 7120P avec icc					
	Sans drapeau de compilation			Avec des drapeaux de compilation (-fp-model precise et –fp-model source)		
	-O0	-O2	-O3	-O0	-O2	-O3
MT19937	42min15.09s	**6min29.34s**	**6min45.60s**	42min15.35s	**6min29.33s**	**6min45.58s**
MT19937_64	42min55.69s	14min19.72s	14min19.72s	42min55.60s	14min20.10s	14min19.69s
TinyMT_32	41min26.48s	8min56.46s	8min56.28s	41min26.91s	8min56.39s	8min56.26s

TinyMT_64	42min24.03s	10min1.28s	10min1.37s	42min23.40s	10min1.34s	10min1.31s
MRG32K3a	1h46min 15s	51min34.23s	51min34.41s	1h33min 15s	1h2min 16s	1h2min 16s
WELL512a	47min16.82s	13min0.14s	13min0.04s	47min16.57s	13min0.04s	13min0.03s
MLFG_64	**34min49.06s**	12min17.54s	12min17.54s	**34min48.70s**	12min17.53s	12min17.50s

Table 5.11. Performances de divers PRNGs sur une architecture Intel Xeon Phi 7120P avec le compilateur icc.

Les tableaux 5.10 et 5.11 montrent que les générateurs MT19937 et MLFG_64 sont les plus rapides sur les deux architectures MICs (7120P et 5110P). D'un autre côté, on note comme précédemment que le générateur MRG32k3a n'est pas très efficace sur ces architectures, même quand il utilise les optimisations. L'utilisation des drapeaux de compilation *(–fp-model precise* et *–fp-model source*) ne change que très légèrement les performances. Cette remarque nous sera très utile lorsque nous traiterons de la reproductibilité et de la conformité IEEE 754.

Dans cette partie, nous avons considéré les problèmes de reproductibilité de générateurs de nombres pseudo-aléatoires récents qui ont passé avec succès les tests de la batterie de référence TestU01 (L'Ecuyer et Simard 2007). Nous avons réalisé ces tests avec différentes architectures matérielles, différents compilateurs et différentes options de compilation. Nous avons identifié des problèmes de non-reproductibilité liés à l'utilisation de compilateurs différents sur une même architecture matérielle et avec le même système d'exploitation, ou bien à l'utilisation d'un même compilateur sur différentes architectures. Nous avons également constaté une incompatibilité entre les versions 32 bits et 64 bits des compilateurs et des générateurs. Nous avons enfin observé des problèmes de non-reproductibilité entre des machines virtuelles avec différents systèmes invités ou entre une machine virtuelle et une machine réelle qui utilisent le même système d'exploitation. Bien que les différences observées puissent apparaître très petites, les PRNGs étant la source aléatoire des simulations stochastiques, on ne peut pas exclure que dans certains cas, à cause d'une sensibilité aux conditions initiales (effet papillon), on pourrait obtenir rapidement des résultats très différents.

Les générateurs les plus fiables et les plus efficaces que nous avons identifiés sont le générateur Mersenne Twister MT19937 et le *Multiple Lagged Fibonacci Generator* MLFG_64.

III - Evaluation de la solution de C. Ismay pour tester la corrélation de PRNGs en parallèle

Nous présentons ici l'évaluation de l'approche de C. Ismay pour tester la corrélation des PRNGs en parallèle. L'application développée par C. Ismay se décompose en trois modules (avec les extensions *mcorr*, *mmult* et *mport*) qui utilisent une matrice dans laquelle chaque colonne représente un flux de nombres pseudo-aléatoires. Si la matrice est remplie ligne à ligne à partir d'un flux stochastique unique, cela correspond à la méthode de parallélisation des PRNGs dite du « tourniquet » (ou *leap* frog) que nous avons présentée dans les chapitres 3 et 4. Cette méthode est utilisée par défaut dans les tests de C. Ismay. En remplissant la matrice par colonne d'abord, et en effectuant un saut (avec un algorithme de *jump ahead*) dans le flux stochastique d'origine pour remplir la colonne suivante, nous avons ainsi pu tester la technique de parallélisation de PRNGs dite de *sequence splitting*. Nous avons considéré des matrices à N lignes et P colonnes (correspondant donc à P flux de longueur N) pour les valeurs de (N,P) suivantes : (16384, 2) ; (8192, 4) ; (4096, 8) ; (2048, 16) ; (1024, 32) ; (512, 64) ; (256, 128). Ces tests ont été effectués sur un serveur **Intel E7-8870 Xeon** avec le compilateur gcc**.**

III.1 - Application à des PRNGs de faible qualité statistique

Nous avons choisi, pour ces tests, des PRNGs pour lesquels des corrélations ont été détectées par la batterie de tests « Crush » de l'outil TestU01 de (L'Ecuyer et Simard 2007). Ces générateurs sont pour l'essentiel des *Linear Congruential Generators* (LCGs), notés LCG(*m*, *a*, *b*) pour une fonction de transition définie par la relation de récurrence de type $x_{n+1} = (a\, x_n + b) \bmod m$, et un générateur quadratique, Coveyou-32, dont la fonction de transition est définie par la relation de récurrence :

$$x_{n+1} = x_n\,(x_n + 1) \bmod 2^l.$$

Les LCGs que nous avons testés sont les suivants :

- LCG (2^{64}, 16598013, 12820163) de Microsoft Visual Basic 6.0 ;
- LCG (2^{31}, 65539, 0), le RANDU d'IBM (1968) ;
- LCG (2^{32}, 69069, 1) de Marsaglia (1972) ;
- LCG (2^{32}, 1099087573, 0) de Fishman (1990) ;
- LCG (2^{48}, 25214903917, 11) de Java (dans le package *java.util.Random*) ;

- LCG (2^{15}, 1103515245, 12345) de la norme ANSI C & BSD.

Tous ces PRNGs ont été testés avec des flux stochastiques générés à partir d'un statut initial égal à 1. Les résultats obtenus sont présentés dans les tableaux A.1 et A.2 de l'Annexe.

La technique du « tourniquet » (ou *leap frog*)

L'application de l'extension *mcorr* aux PRNGs testés ne nous a donné qu'une unique réponse : « *Reject none of the null hypotheses* » quels que soient le test et la taille de la matrice. Comme l'hypothèse nulle est une hypothèse d'absence de corrélation (Ismay 2013), cette réponse unique signifie que l'extension *mcorr* n'a détecté aucune corrélation entre les colonnes de la matrice. Les changements de taille de la matrice (variations du nombre et de la longueur des flux) n'ont pas eu d'influence sur les résultats de ces tests.

Par contre, l'extension *mmult* détecte des corrélations entre tous les flux générés par LCG 2^{15} BSD, LCG 2^{32} Marsaglia, LCG 2^{32} Fishman et Coveyou-32. Par ailleurs, pour l'extension *mport,* seul le test de Mahdi-McLeod parvient à détecter des corrélations lorsqu'il y a plus de 32 flux testés (matrices avec 32, 64 ou 128 colonnes).

La technique « sequence splitting »

L'extension *mcorr* détecte des corrélations entre les flux à partir de seulement 2 flux pour le générateur LCG 2^{15} BSD, de 4 flux pour LCG 2^{48} Java, LCG 2^{32} Marsaglia, LCG 2^{32} Fishman et Coveyou-32. En revanche, pour le LCG 2^{64} de Microsoft Visual Basic, il faut quand même au moins 16 flux pour que *mcorr* commence à détecter des corrélations entre ceux-ci. Pour tous les PRNG, à l'exception du LCG 2^{64} de Microsoft Visual Basic, pour lequel il faut encore dans ce cas au moins 16 flux, l'extension *mmult* détecte systématiquement des corrélations, quel que soit le nombre de flux testés. Enfin, en ce qui concerne l'extension *mport*, le test de Mahdi-McLeod détecte aussi des corrélations dès qu'ils sont appliqués avec au moins 32 flux (16 suffisent pour le LCG 2^{64} de Microsoft Visual Basic et pour Coveyou-32, et seulement 4 pour le LCG 2^{15} du ANSI C BSD).

III.2 - L'approche de C. Ismay avec des PRNGs de bonne qualité statistique

Nous avons choisi ici des PRNGs de bonne qualité statistique qui ont été validés par l'outil TestU01 (L'Ecuyer and Simard 2007). Il s'agit de MRG32k3a, MT19937, MLFG, Coveyou64, et ran2 cité dans

l'ouvrage de référence *Numerical Recepies in C : The art of Scientific computing* (Press et al. 2007). Comme nous ne disposions pas d'algorithme de *jump ahead* pour tous ces PRNGs, nous avons restreint nos tests à l'approche du « tourniquet ». Les résultats obtenus sont présentés dans le tableau A.1.

L'extension *mcorr* ne détecte aucune corrélation pour ces PRNGs (« *Reject none of the null hypotheses* »). Pour tous les PRNGs, à l'exception de Coveyou-64, l'extension *mmult* arrive à détecter des corrélations lorsque le test est lancé avec le nombre de flux maximal, c'est-à-dire avec 128 flux (pour Coveyou-64, 32 flux suffisent). Enfin, le test de Mahdi-McLeod de l'extension *mport* détecte, quant à lui, des corrélations pour tous les PRNGs testés dès lors que le test est lancé avec au moins 32 flux.

Les analyses que nous avons effectuées confirment que les tests développés par C.Ismay ont une puissance statistique intéressante, puisqu'ils mettent en évidence des corrélations qui ne sont pas détectées par d'autres méthodes. Cette approche est donc utile pour tester la corrélation des PRNGs parallèles. Malgré les corrélations détectées, ces générateurs sont les meilleurs que l'on puisse utiliser actuellement dans des simulations stochastiques parallèles.

IV - Proposition de méthode de simulation stochastique parallèle en utilisant la technique des réplications en parallèle (MRIP)

Dans le chapitre précédent, nous avons proposé une approche de parallélisation des simulations stochastiques numériquement reproductibles (Hill 2015), et nous présentons ici l'application de celle-ci à une simulation stochastique de croissance démographique, de type Monte Carlo, pour laquelle le nombre de tirages de nombres pseudo-aléatoires dépend des paramètres d'entrée.

L'approche de parallélisation utilisée est la technique MRIP (*Multiple Replication in Parallel*), qui consiste à exécuter de manière indépendante les réplications de cette simulation en attribuant à chacun un flux stochastique qui lui est propre (Hill 1997) (Pawlikowski et al. 1994, 2002).

IV.1 - Description et installation de la simulation

Dans ce modèle, le temps est discrétisé. On considère donc des dates entières, $n \in \mathbb{N}$. Pour chaque date n, l'effectif de la population est noté N_n. La simulation est initialisée à l'instant $n = 0$ par la seule valeur N_0. Nous avons choisi d'initialiser nos simulations avec un individu unique, soit $N_0 = 1$.

Le passage d'une génération à la suivante s'effectue ainsi : pour chacun des N_n individus de la génération n, on détermine le nombre de ses descendants en effectuant un tirage aléatoire suivant une loi de Poisson de paramètre λ (il s'agit du seul paramètre de ce programme). Donc, pour k variant de 1 à N_n, si l'on note $X_k^{(n)}$ la variable aléatoire, distribuée suivant une loi de Poisson de paramètre λ, qui donne le nombre de descendants de l'individu k de la génération n, le nombre total de descendants issus de la génération n est donc $\sum_{k=1}^{N_n} X_k^{(n)}$.

Dans ce modèle, nous avons considéré que les individus de la génération n ne survivent pas à la génération suivante. Ainsi, d'une génération n à une génération $n + 1$, la population est totalement renouvelée, les N_n géniteurs étant remplacés par leurs descendants. On obtient ainsi une relation de récurrence très simple : $N_{n+1} = \sum_{k=1}^{N_n} X_k^{(n)}$, où les variables aléatoires $X_k^{(n)}$ sont indépendantes entre elles et identiquement distribuées suivant une loi de Poisson de paramètre λ. Remarquons en particulier que si, à une génération, il n'y a pas de descendants, alors il y a extinction de la population: Si $\forall k \in \{1,..., N_n\}, X_k^{(n)} = 0$, alors $N_{n+1} = 0$.

L'intérêt de ce modèle est de donner un exemple très simple de simulation stochastique dont le nombre de tirages aléatoires peut être très variable d'une réplication à l'autre. En particulier, il n'est pas exclu (probabilité $e^{-\lambda}$) que $X_1^{(0)} = 0$, c'est-à-dire que l'individu originel n'ait pas de descendance. Dans ce cas, le nombre de tirages aléatoires de la simulation est réduit à 1. Tandis que si $X_1^{(0)} > 0$, pour certaines valeurs de λ, la population peut proliférer, et le nombre de tirages aléatoires être très grand.

Un autre avantage de la simplicité de ce modèle (en dehors de sa simplicité de programmation) est qu'il existe une solution analytique. On démontre facilement, en effet, que l'espérance de N_{n+1} est donné en fonction de l'espérance de N_n par la formule de récurrence $E[N_{n+1}] = \lambda E[N_n]$, ce qui donne immédiatement $E[N_n] = \lambda^n E[N_0]$, c'est à dire $E[N_n] = \lambda^n$, dans notre cas. Nous constatons alors que, lorsque n tend vers l'infini, il y a, en moyenne, prolifération de la population pour $\lambda > 1$, stabilité pour $\lambda = 1$, et extinction pour $\lambda < 1$.

Le Code 5.1 ci-dessous est une implémentation séquentielle de cette simulation en C. Cette implémentation utilise un seul générateur de nombres pseudo-aléatoires, MRG32k3a, qui a un statut initial constitué par 6 nombre réels de type *double*.

```
/* ------------------------------------------------------------------ */
/* simu.c: Calculate the population growth with the Monte Carlo method */
/* Input : Number of replications for an execution and lambda value    */
/* Output: Display the average of the population growth                */
/* Compile & run: gcc simu.c MRG32k3a.c -o simu -lm                    */
/*                    ./simu Number_of_replications                    */
/* ------------------------------------------------------------------ */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "MRG32k3a.h"

/* --------- Poisson law with an implementation on PRNG -------------- */

unsigned long long int poisson(double lambda){
   long long int r = -1;
   double s = lambda;
   while (s >= 0)
   {
      r = r+1;
      s = s + log(MRG32k3a()); /* MRG32k3a() generates U(0,1)random numbers*/
   }
   return r;
}

/* ------ Stochastic simulation with the Poisson distribution law ---- */

unsigned long long int simu(double lambda){
   int iter = 0, iterMax = 10, ind;
   unsigned long long int individu = 1, nbDes;
   while (individu > 0 && iter < iterMax)
   {
      nbDes = 0;
      for(ind = 1; ind <=individu; ind++)
      {
         nbDes = nbDes + poisson(lambda);
      }
      individu = nbDes;
      iter++;
   }
   return individu;
}

/* ----------------------------Main---------------------------------- */

int main (int argc, char **argv){
   int nbRep, iter;
   double lambda, total, average;
   if(argc < 2 || argc > 3) {
      printf("Error: number of args invalid: ./simu nbReplication lambda...\n");
      return -1;
   }
   nbRep = atoi(argv[1]);
   lambda = atof(argv[2]);
   if(nbRep < 1 ) {
      printf("Error: set of pattern is not validated for this simulation\n");
      return -1;
   }
   for (iter = 0; iter < nbRep; iter++) {
      total += simu(lambda);
```

```
    }
    average = total/(double)nbRep;
    printf("average of individus: %14.10f \n", average);
}
```

Code 5. 1. Code de simulation de croissance démographique de type bactérien.

Le code 5.1 est une implémentation simple par fonctions de cette simulation : nous allons l'utiliser afin d'étudier la reproductibilité numérique sur le plan de la précision des calculs en virgule flottante dans la partie suivante.

IV.2 - Impact de la précision des calculs en virgule flottante

Nous avons lancé la simulation stochastique séquentielle de croissance démographique (cf. Code 5.1) sur plusieurs environnements d'exécution avec 10 réplications par exécution. Les résultats sont présentés dans le tableau 2, avec λ = 5 et le nombre de génération est 10. Nous avons détecté des problèmes de reproductibilité numérique liés à la précision des calculs en virgule flottante : pour le calcul de *average = total/replication* (cf. code 5.1). Ils peuvent être :

- Cas 1 : Utilisation de précision single : *int total*; *average = total/replication*
- Cas 2 : Utilisation de précision double : *int total*; *average = (double)(total/replication)*
- Cas3 : Utilisation de précision double : *int total; average= (double)total/(double)replication*
- Cas 4 : Utilisation de la double précision : *double total*; *average = total/ (double)replication*

Ces cas et les résultats obtenus sont présentés dans le tableau 12 ci-dessous.

Matériel	Cas 1	Cas 2	Cas 3	Cas 4
Core i7-4800MQ (Windows 7, 64 bits, utilisant DevC++ avec gcc-core-3.4.2-2004916-1, gcc-g++-3.4.2--2004916-1,	**206654188**	**206654188**	**206654188**	8611007

MinGw-3.7)				
Ubuntu 13.04 virtuelle sur Virtualbox (3.8.0-19-generic i686-linux, gcc-4.7.3)	**-113053965**	**-113261926**	**-113274214.9**	8611007
Ubuntu 13.04 virtuelle sur Virtualbox (3.8.0-35-generic x86_64-linux, gcc-4..7.3)	8611007	8611007	8611007	8611007
AMD Opteron (TM) 6272 (CentOS release 6.4 with kernel 2.6.32-358.23.2.el6.x86_64, gcc-4.4.7)	8611007	8611007	8611007	8611007
E5-2687W (Debian 3.2.39-2 x86_64 GNU/Linux with kernel 3.2.0-4-amd64, gcc-4.8.2, icc-14.0.3)	gcc: 8611007 icc: 8611007 icc_mic_no_precise: 8611007 icc_mic_precise: 8611007	gcc: 8611007 icc: 8611007 icc_mic_no_precise: 8611007 icc_mic_precise: 8611007	gcc: 8611007 icc: 8611007 icc_mic_no_precise: 8611007 icc_mic_precise: 8611007	gcc: 8611007 icc: 8611007 icc_mic_no_precise: 8611007 icc_mic_precise: 8611007
E5-2650v2 (Debian 3.2.39-2 x86_64 GNU/Linux with	gcc:8611007 icc: 8611007 icc_mic_no_precis	gcc:8611007 icc: 8611007 icc_mic_no_precis	gcc:8611007 icc: 8611007 icc_mic_no_precis	gcc: 8611007 icc: 8611007 icc_mic_no_precis

kernel 3.2.0-4-amd64, gcc-4.8.2, icc-14.0.2)	e: 8611007 icc_mic_precise: 8611007	e: 8611007 icc_mic_precise: 8611007	e: 8611007 icc_mic_precise: 8611007	e: 8611007 icc_mic_precise: 8611007

Table 5.12. Impact, sur plusieurs environnements d'exécution, de la précision des calculs en virgule flottante dans la simulation séquentielle de croissance démographique.

Le Tableau 5.12 montre que le « Cas 4 » est correct et donne les résultats reproductibles sur tous les environnements d'exécution testée. Les Cas 2 et Cas 3 ne marchent pas bien dans le système d'exploitation 32 bits i686-Linux et l'ancienne version du compilateur gcc-3.4.2. Le « Cas1 » n'est pas correct mais quelques fois on l'utilise pour le calcul quand on n'a pas d'expérience de codage en C/C++. Les cas 1, 2 et 3 conduisent à une non-reproductibilité numérique dans les opérations en virgule flottante, en particulier, sur la division. Nous recommandons l'utilisation de la double précision (Cas 4).

IV.3 - Détermination du nombre de réplications nécessaires

Il est très important de pouvoir déterminer le nombre de réplications d'une simulation de Monte Carlo qu'il est nécessaire de lancer pour atteindre des intervalles de confiance sur les résultats. Nous présentons donc un algorithme qui permet de déterminer le nombre de réplications nécessaires pour obtenir un intervalle de confiance égal à 1 %. Ceci signifie que le rayon de l'intervalle de confiance doit être inférieur ou égal à 1 % de la valeur moyenne du résultat considéré. Pour l'exemple de notre simulation démographique, l'unique résultat fourni par la simulation est l'effectif final de la population de bactéries.

Entrée : un nombre initial de réplications, nbReplications

Sortie : le nombre de réplications nécessaires pour obtenir un intervalle de confiance égal à 1 %

Etape 0 : On calcule la valeur moyenne et l'écart-type avec le nombre initial de réplications

Etape 1 : On calcule la valeur souhaitée de la précision relative : precisionRelative = 0.01 * moyenne

Etape 2 : On calcule un nouveau nombre de réplications avec la formule :

nbReplicationsNew = ((2,764 * écart-type)/ precisionRelative)^2

Etape 3 : Si nbReplicationsNew < nbReplications, alors on peut arrêter, sinon :

On relance (nbReplicationsNew – nbReplications) réplications

On recalcule les nouvelles valeurs de la moyenne, de l'écart-type et de la précision relative avec :

precisionRelativeEffective = (2.5758 * écart-type)/racine_carrée(nbReplicationsNew)

Si precisionRelativeEffective < 0.01 * moyenne

Alors on peut s'arrêter

sinon :

nbReplications = nbReplicationsNew

retourner à l'étape 1

Algorithme 5. 1. Un algorithme de détection du nombre de réplications nécessaire.

L'algorithme 5.1 a été mis en œuvre dans l'outil RSPR (*Running a stochastic Simulation in Parallel, ensuring numerical Reproducibility*) pour déterminer le nombre de réplications nécessaires pour notre simulation démographique de type Monte Carlo en utilisant le générateur de nombres pseudo-aléatoires MT19937. Nous choisissons la valeur du ratio de reproduction $\lambda = 2$ correspondant dans ce cas à une réalité biologique, la scissiparité, ou la division binaire des bactéries. On obtient ainsi un modèle malthusien de croissance démographique très simple où la population, en moyenne, double à chaque génération, chaque cellule étant remplacée, toujours en

moyenne, par deux cellules filles. Cette simulation a été exécutée sur une machine Intel E5-2687W Xeon avec 16 cœurs physiques.

Paramètres du programme	Nombre de réplications estimé	Moyenne calculée	Ecart-type calculé	Intervalle de confiance calculée
nbIndividuInit = 1 λ = 2 nbReplicationInit = 1000	66493	1024.4772382055	1026.8369195821	10.2571250489
nbIndividuInit = 10 λ = 2 nbReplicationInit = 100	6821	10198.2716610468	3271.5504723040	102.0332544039
nbIndividuInit = 100 λ = 2 nbReplicationInit = 100	671	102795.955290611	10314.6693383670	1025.6665425006
nbIndividuInit = 1000 λ = 2 nbReplicationInit = 10	51	1010221.0588235294	28590.5489227212	10312.1576659379

Table 5.13. Le nombre de réplications détecté et ses valeurs.

On peut voir que, dans tous les cas, la précision obtenue est très proche de 1% de la valeur moyenne.

On peut également observer que le nombre de réplications nécessaires dépend de l'effectif initial de la population. Ceci s'explique par le fait que, plus la population initiale est petite, plus il y a de variabilité dans les scénarios d'évolution de celle-ci, et c'est l'augmentation de la variabilité des résultats qui impose une augmentation du nombre de réplications.

Nous avons aussi utilisé ces valeurs (le nombre de réplications nécessaire, le paramètre d'entrée λ, la valeur moyenne et l'écart type) pour étudier la reproductibilité numérique des résultats sur plusieurs environnements d'exécution. Nous avons utilisé deux architectures de la famille d'Intel, Xeon et Xeon Phi, et nous avons testé la reproductibilité entre code séquentiel et parallèle. Ensuite, nous avons également estimé les performances. Nous présentons les résultats obtenus dans les parties suivantes de ce chapitre.

IV.4 - Impact des conflits d'accès entre les threads stochastiques et de l'utilisation de l'option non-optimisée par défaut de la compilation dans des programmes parallèles utilisant OpenMP ou Pthread.

Pour cette expérience, nous avons, pour assurer la reproductibilité numérique, développé une version en C++ de la simulation démographique en respectant l'approche de parallélisation de simulations stochastiques préconisée dans (Hill 2015). Cette approche se fonde sur la construction d'objets ou processus stochastiques indépendants et aussi garantit la répétabilité numérique. La simulation a été paramétrée avec un nombre initial d'individus égal à 1000, 10 générations et un taux de reproduction λ = 2. Nous avons lancé 51 réplications (-cf. tableau 5.13) sur la machine Intel E5-2687W Xeon et avec 16 cœurs physiques. Enfin, nous avons aussi, au cours de cette expérience, étudié les changements du nombre de threads du code parallèle.

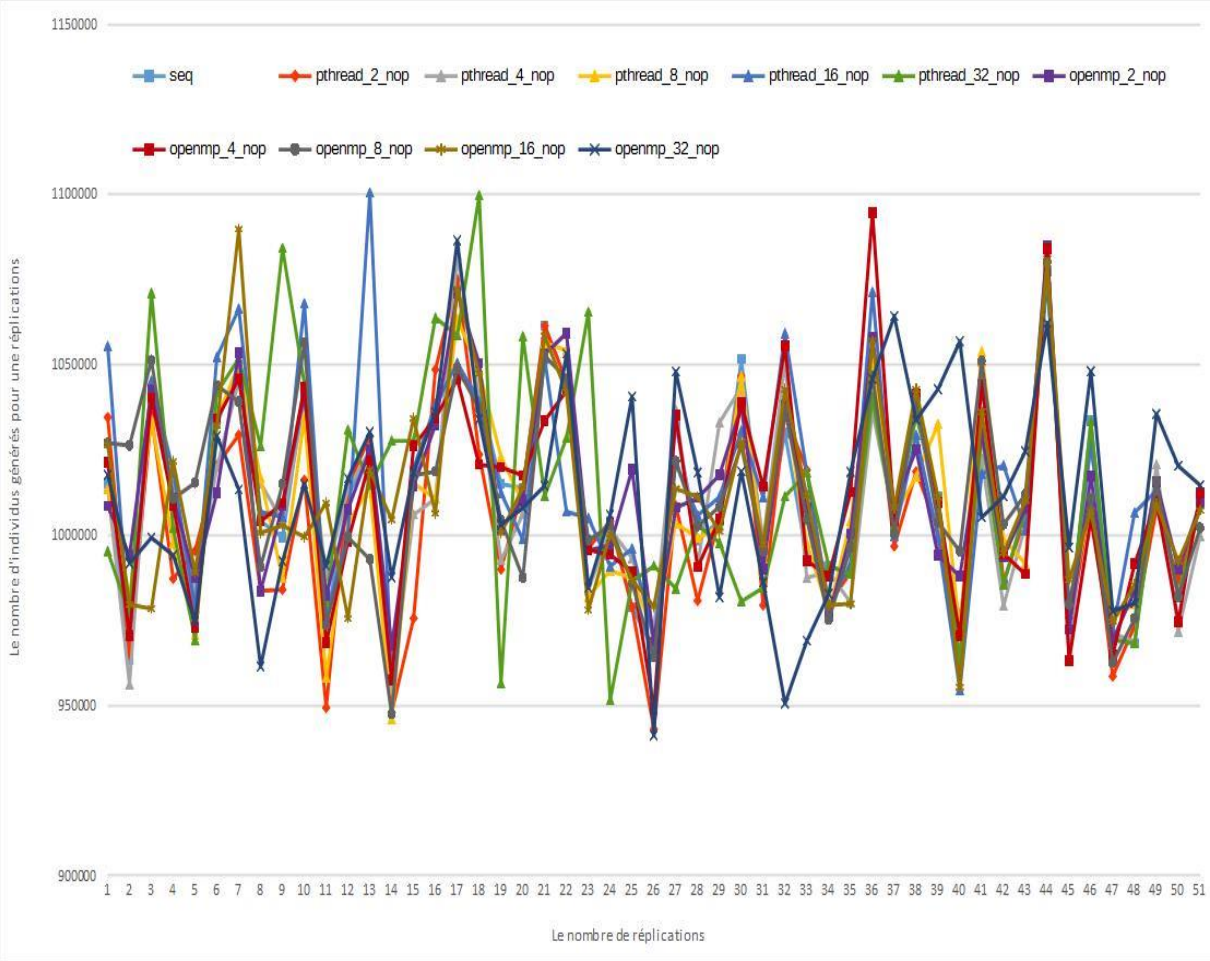


Figure 5.1.Les résultats intermédiaires dans la simulation de croissance bactérienne incluant les résultats séquentiels puis parallèles avec les bibliothèques OpenMP et Pthread pour 51 réplications.

La Figure 5.1 présente toutes les valeurs obtenues dans cette expérience. On voit que les résultats intermédiaires ne sont pas répétables entre l'exécution séquentielle et l'exécution parallèle utilisant OpenMP ou Pthread quel que soit le nombre de threads. Dans cette figure les légendes précisent le nombre de threads utilisés, par exemple, le nom « pthread_2 » signifie que le programme est parallélisé en utilisant la bibliothèque Pthread avec 2 threads, openMP_32 précise que la bibliothèque utilisée est OpenMP avec 32 threads. Bien que l'on utilise les mêmes paramètres d'entrée, le même générateur de nombres pseudo-aléatoires et les mêmes statuts initiaux et que chaque objet stochastique a son statut initial propre, nous obtenons des résultats différents. De plus, les résultats intermédiaires des codes parallèles avec OpenMP et Pthread ne sont pas reproductibles d'une exécution à l'autre. C'est-à-dire, les résultats intermédiaires de ces codes parallèles changent à chaque nouvelle exécution (on parle d'une absence de reproductibilité « run to run » (d'une exécution à l'autre).

De même, les résultats finaux de la simulation sont vraiment différents selon ce qui est montrés dans le tableau 5.14 ci-dessous. En d'autres termes, ces codes parallèles ne sont pas répétables numériquement.

Mode d'exécution	Moyenne	Ecart-type	Intervalle de confiance
Séquentiel	1010221.0588235294	28590.5489227212	10312.1576659379
Pthread_2	1005233.2549019608	32170.3059667417	11603.3192712457
Pthread_4	1009804.4117647059	30355.3300983434	10948.6862536921
Pthread_8	1009215.6862745098	29481.6640466676	10633.5687616631
Pthread_16	1015082.2156862745	32085.6753447087	11572.7943477778
Pthread_32	1013350.3529411765	35159.3643788175	12681.4252461026
OpenMP_2	1012450.2745098040	27114.3251387254	9779.7071539901
OpenMP_4	1010831.3725490196	30733.6387194809	11085.1361748915
OpenMP_8	1011433.3137254902	27527.8617172670	9928.8632408523
OpenMP_16	1011228.4509803922	28472.4406746795	10269.5578935885
OpenMP_32	1012476.9803921569	30030.6282674346	10831.5714517412

Table 5.14. Les résultats non-reproductibles entre le code séquentiel et les codes parallèles utilisant OpenMP ou Pthread avec le compilateur gcc-4.8.2.

Pour obtenir des résultats répétables, nous avons relancé cette expérience en ajoutant l'option d'optimisation (-Ox avec x = 1, 2, 3) du compilateur gcc-4.8.2, ou avec l'utilisation de la fonction de verrou « set_lock() » d'OpenMP et une fonction « mutex_lock() » dans le cas de la bibliothèque Pthread. Nos nouveaux résultats obtenus sont alors répétables et comparables entre l'exécution séquentielle et parallèle, mais aussi entre exécutions parallèles avec les 2 bibliothèques différentes. Le tableau 5.14 montre la difficulté d'obtention de résultats identiques dans ce contexte. Les différences observées peuvent avoir pour origine des conflits d'accès entre les threads stochastiques

et/ou une mauvaise compilation avec le compilateur gcc-4.8.2 (« *miss-compilation* » en anglais) lorsque l'option d'optimisation n'est pas activée.

Nous avons répété aussi cette expérience avec le compilateur icc-14.0.3 incluant l'option d'optimisation (*-O2 –fp-modelprecise –fp-model-source*), puis de non-optimisation sans drapeau sur deux machines à base d'Intel Xeon classique. Nos résultats sont alors également reproductibles entre les programmes séquentiels compilés par le compilateur icc-14.0.3 et le compilateur gcc-4.8.2 (–O2), et entre les programmes séquentiel et parallèles compilés par le compilateur icc-14.0.3.

Nous avons présenté, dans cette partie, l'impact sur la reproductibilité des résultats d'une simulation, parallélisé avec OpenMP ou la bibliothèque Pthreads (ces modèles de programmation ont été présentés dans le chapitre 4 « Proposition ») ainsi que de l'utilisation de l'option par défaut, sans optimisation du code, du compilateur gcc-4.8.2 64 bits. Nous montrons aussi que, dans le cas de codes parallèles utilisant OpenMP ou la bibliothèque Pthreads avec le compilateur gcc, il est nécessaire, pour avoir la reproductibilité, d'utiliser des fonctions d'accès en exclusion mutuelle afin d'éviter les conflits d'accès entre les threads.

IV.5 - Reproductibilité numérique du code parallèle utilisant OpenMPet l'outil RSPR sur deux architectures de la famille Intel Xeon et un seul compilateur icc

Réduction de somme et reproductibilité numérique

Dans cette partie, nous choisissons uniquement OpenMP dans cette comparaison parce que l'implémentation du code parallèle utilisant OpenMP est plus simple que celle avec Pthread. De plus, OpenMP est plus utilisé dans le monde du calcul à haute performance et on peut définir des données spécifiques aux plates-formes via une nouvelle variable d'environnement OMP_PLACES (il faut cependant vérifier que la version OpenMP utilisée est _OPENMP = '201307', qui correspond à la version 4.0). Cette variable d'environnement permet de profiter de l'affinité des threads, c'est-à-dire de spécifier les cœurs sur lesquels les threads d'un programme parallèle peuvent être exécutés. Nous avons donc, dans notre travail, défini un ensemble de variables d'environnement en nous conformant à un document officiel du *libgomp* ainsi qu'à un article portant sur le concept d'affinité des threads avec OpenMP (Eichenberger et al. 2012):

```
#le nombre de 2 a 16 correspond au nombre de coeurs physiques de la machine Intel
E5-2687W Xeon
export OMP_NUM_THREADS=16  ou export OMP_NUM_THREADS=2
```

```
export OMP_PLACES= "threads(16)"  ###16 hardware-threads
ou export OMP_PLACES= "cores(2)"  ###2 cœurs
export OMP_PROC_BIND=spread
```

C'est-à-dire, chaque lieu (« place ») correspond à un seul cœur, chaque cœur correspond à un ou plusieurs hardware-threads. La variable d'environnement `OMP_PLACES` permet d'associer les threads aux cœurs (« *one place per core* » en anglais). Dans notre expérience, il y a 16 cœurs physiques, chaque cœur physique a deux hardware-threads ou processeurs logiques, ce qui est pris en compte par *hwloc-calc --number-of cores all* et utilisé dans l'outil RSPR.

Dans la partie précédente, nous n'avons pas trouvé de problème de non-reproductibilité numérique de la simulation utilisant l'OpenMP et l'option d'optimisation (-O2) sur la machine Intel E5-2687W. Cette simulation a été paramétrée avec un nombre initial d'individus égal à 1000, 10 générations, un taux de reproduction $\lambda = 2$ et 51 réplications. Cependant, quand nous avons effectué une autre simulation pour un taux de reproduction $\lambda = 2$, un nombre de générations fixé à 20, le nombre de réplications nécessaires est de 33467, nous avons trouvé un problème de non-reproductibilité. Nous avons détecté que ce problème provient des opérations en virgule flottante dans la somme des carrées pour notre simulation utilisant OpenMP sur la machine Intel Xeon E5-2687w avec le compilateur gcc-4.8.2 et ce même si on utilise l'option d'optimisation (-O2). Les erreurs sont indiquées sur la table 5.15 ci-dessous. Par contre, sans OpenMP, mais en utilisant notre outil RSPR, nous n'avons pas ce problème de reproductibilité. Nous présentons les valeurs réellesavec 10 chiffres après la virgule (`printf("… %14.10f\n", value)`)

Dans le tableau 5.15, la notion «seq » représente le résultat de la simulation stochastique séquentielle. Les résultats intermédiaires sont reproductibles entre la simulation stochastique de croissance démographique, le nombre des résultats intermédiaries est de 33467 (réplications), donc nous ne présentons dans cette table que les résultats non-reproductibles lors du calcul final.

Cœurs	OpenMP	RSPR
Moyenne		
seq	2092572.2058445632	2092572.2058445632
2	2092572.2058445632	2092572.2058445632
4	2092572.2058445632	2092572.2058445632
6	2092572.2058445632	2092572.2058445632
8	2092572.2058445632	2092572.2058445632
10	2092572.2058445632	2092572.2058445632
12	2092572.2058445632	2092572.2058445632
14	2092572.2058445632	2092572.2058445632
16	2092572.2058445632	2092572.2058445632
Ecart type		
seq	1486210.1625192484	1486210.1625192484
2	1486210.162519**1551**	1486210.1625192484
4	1486210.162519**0575**	1486210.1625192484
6	1486210.162519**1069**	1486210.1625192484
8	1486210.162519**1229**	1486210.1625192484
10	1486210.162519**1348**	1486210.1625192484
12	1486210.162519**1488**	1486210.1625192484
14	1486210.162519**1637**	1486210.1625192484
16	1486210.162519**1758**	1486210.1625192484
Somme des carrées		
seq	220469855899163488.0	220469855899163488.0
2	22046985589**954208**.0	220469855899163488.0
4	22046985589**944512**.0	220469855899163488.0
6	22046985589**949408**.0	220469855899163488.0
8	22046985589**951008**.0	220469855899163488.0
10	22046985589**952192**.0	220469855899163488.0
12	22046985589**953600**.0	220469855899163488.0
14	22046985589**955072**.0	220469855899163488.0
16	22046985589**956288**.0	220469855899163488.0
Moyenne du total des carrées		
seq	6587679083848.6718750000	6587679083848.6718750000

2	6587679083848.**3945312500**	6587679083848.671875
4	6587679083848.**1044921875**	6587679083848.671875
6	6587679083848.**2509765625**	6587679083848.671875
8	6587679083848.**2988281250**	6587679083848.671875
10	6587679083848.**3339843750**	6587679083848.671875
12	6587679083848.**3759765625**	6587679083848.671875
14	6587679083848.**4199218750**	6587679083848.671875
16	6587679083848.**4560546875**	6587679083848.671875
Intervalle de confiance		
seq	20925.8917144781	20925.8917144781
2	20925.891714468	20925.8917144781
4	20925.891714454	20925.8917144781
6	20925.891714461	20925.8917144781
8	20925.891714463	20925.8917144781
10	20925.891714465	20925.8917144781
12	20925.891714467	20925.8917144781
14	20925.891714469	20925.8917144781
16	20925.891714471	20925.8917144781

Table 5.15. Les résultats obtenus par la simulation utilisant OpenMP et RSPR sur l'architecture Intel Xeon E5-2687w avec le compilateur gcc et l'option –O2.

Nous avons indiqué que les résultats obtenus de l'outil « rspr » sont toujours reproductibles sur l'architecture à multiple cœurs Intel E5-2687W Xeon avec les deux compilateurs gcc (–Ox avec x = 1, 2, 3) et icc (avec ou sans optimisation) selon les expériences précédantes.

L'outil RSPR sur deux architectures différentes de la famille Intel Xeon

Dans la partie précédente, les calculs de simulation stochastique de croissance démographique utilisant OpenMP ne sont pas toujours reproductibles, Donc pour la suite de nos investigations sur les deux architectures différentes de famille Intel Xeon Phi, nous ne gardons que notre outil RSPR.

De plus, nous n'utilisons que le compilateur icc sur les deux architectures de la famille Intel Xeon (avec un socket, il y a 60 cœurs physiques et 240 cœurs logiques) car c'est le seul compilateur pleinement opérationnel sur les Xeon Phi. Nous retenons les options d'optimisation du code et les drapeaux de compilation nécessaires au respect de la norme IEEE 754, « *-O2 –fp-model source –fp-model precise -fimf-precision=high*» sur les deux plate-formes. Sur l'Intel Xeon Phi, nous utilisons aussi l'option « *–no-fma* » (pour désactivier l'opération fused-multiply-add). Cette information est

précieuse, elle a été présentée par (Corden 2013) dans la partie III.6 du chapitre 4. Nous présentons ici les valeurs réelles avec 15 chiffres après la virgule (`printf("… %14.15f\n", value)`) dans notre code. Nos résultats de la simulation utilisant RSPR sont toujours repétables numériquement entre deux architectures différentes de la famille Intel Xeon classiques et Xeon Phi. S'il on oublie de spécifier l'option « -no-fma » et/ou l'option « -fimf-precision=high » sur l'architecture Intel Xeon Phi, on obtient encore la répétabilité numérique des résultats pour cette simulation. Si on n'utilise pas ces options, on ne peut pas garantir la répétabilité numérique entre les deux architectures (pour la simulation de 51 réplications, 1000 individus et 10 générations – cf tableau 5.15 ci-dessous avec 15 décimales). Dans ce cas, la non-reproductibilité numérique peut aussi venir des opérations arithmétiques fusionnant multiplication et addition (détaillée dans le début de ce manuscrit) et/ou la fonction mathématique sqrt() dans la bibliothèque *math.h*. Par contre, quand on utilise bien les options « -no-fma » et « -fimf-precision=high » on constate bien que les résultats de la simulation stochastique de croissance démographique utilisant RSPR sont toujours répétables sur chaque architecture identique (« run to run »), et aussi entre deux architectures Intel différentes. En d'autres termes, notre approche, notre outil et la technique MRIP garantissent la répétabilité numérique.

RSPR	**Intel E5-2687 W Xeon** *(-O2 -fp-model source -fp-model precise)*	**Intel 5110P Xeon Phi** *(-O2 -fp-model source -fp-model precise)*
Séquentiel	Nombre de replications: 51 Moyenne: 1010221.058823529398069 Ecart-type: 28590.548922721**205599** Total carree: 52089564366090.000000000000000 Moyenne carree: 1021364007178.235351562500000 Intervalle de confidence: 10312.157665937**938873**	Nombre de replications: 51 Moyenne:1010221.058823529398069 Ecart-type: 28590.548922721**074632** Total carree: 52089564366090.000000000000000 Moyenne carree: 1021364007178.235351562500000 Intervalle de confidence: 10312.157665937**893398**
Parallèle	Nombre de replications: 51 Moyenne: 1010221.058823529398069 Ecart-type: 28590.548922721**205599** Total carree: 52089564366090.000000000000000 Moyenne carree: 1021364007178.235351562500000	Nombre de replications: 51 Moyenne:1010221.058823529398069 Ecart-type: 28590.548922721**074632** Total carree: 52089564366090.000000000000000 Moyenne carree: 1021364007178.235351562500000

	Intervalle de confidence: 10312.157665937**938873**	Intervalle de confidence: 10312.157665937**893398**

Table 5.16. La non-reproductibilité numérique des valeurs d'écart type et de moyenne des carrées sur deux architectures de famille Intel Xeon avec le compilateur icc/icpc sans l'option –no-fma pour la simulation de croissance démographie utilisant RSPR.

V - Conclusion

Dans ce chapitre, nous nous sommes intéressés au problème de la reproductibilité des résultats de simulations stochastiques de type Monte Carlo lorsque les réplications sont exécutées en parallèle sur un système de calcul à haute performance (architectures multicores et manycores de la famille Intel Xeon). Nous avons montré des cas de non-reproductibilité numérique avec des générateurs modernes de nombres pseudo-aléatoires. Nous avons aussi mis en œuvre des tests statistiques parallèles de corrélation à partir des travaux de ChesterIsmay. A partir de ces études, nous pouvons recommander l'utilisation des générateurs MT19937, MLFG et MRG32k3a pour des simulations stochastiques scientifiques respectant une véritable reproductibilité numérique. Nous avons montré l'utilité de l'approche de C.Ismay pour tester la corrélation de flux parallèles issus des PRNGs.

Ensuite, nous avons montré également des problèmes de reproductibilité numérique sur un exemple de simulation stochastique parallèle utilisant Pthread ou OpenMP. L'exemple montre que si l'on utilise les options par défaut du compilateur gcc (sans optimisation) la répétabilité, même d'une exécution à l'autre est perdue! Même si, on utilise l'option d'optimisation (-O2), on trouve aussi un problème de non-reproductibilité dans une simulation parallèle utilisant OpenMP avec un grand nombre de réplications (voir notre simulation de 33467 réplications). Ce problème vient de la phase de « réduction » liée à la précision du calcul, et à la gestion des erreurs d'arrondis et peut-être à l'ordre des calculs.

Par contre, l'approche préconisée par notre outil « rspr » fonctionne très bien même dans le cas où l'on pourrait douter du compilateur. En effet, l'utilisation d'OpenMP ou de Pthread impacte la reproductibilité numérique. D'autre part, nous ne pouvons pas nous attendre à une reproductibilité bit à bit entre l'architecture multicore Intel Xeon classique et l'architecture manycore Intel Xeon Phi. Ce problème a déjà été identifié par (Corden 2013), même si l'on utilise les drapeaux et options les plus précises du compilateur icc sur ces deux architectures. Nous apportons aussi une attention

particulière à l'impact de la précision des calculs en virgule flottante dans la simulation stochastique afin de garantir la reproductibilité numérique sur plusieurs environnements d'exécution différents.

Pour un exemple simple de simulation stochastique démographique, l'accélération obtenue par notre outil « RSPR » est meilleure que celle des codes parallèles utilisant la bibliothèque OpenMP. L'interface de programmation d'OpenMP est facile à utiliser pour paralléliser un programme mais il n'est pas facile de vérifier que le programme a un comportement numérique correct. De même, il peut être délicat d'éviter les problèmes de chevauchement des données (« *data races or races* » en anglais) bien qu'il existe des outils de référence : ThreadChecker d'Intel, RaceStand de GNU, ThreadAnalyzer de Sun Inc, etc. (Ha et al. 2009). Avec ce type de bibliothèque, le problème de la reproductibilité numérique devient un défi et aussi un problème intéressant lié à la programmation parallèle utilisant l'interface OpenMP.

A partir de nos travaux, nous voulons ajouter un peu de détail pour compléter l'approche de (Hill 2015) afin de garantir la répétabilité numérique entre le code séquentiel et parallèle. Nous pensons qu'il faut mentionner le fait d'avoir une bibliothèque mathématique de qualité et on aboutit à un nouveau tuple de multi-exigence pour avoir une nouvelle simulation stochastique parallèle qui garantit la reproductibilité numérique: {*affectation de flux aux objets stochastiques définis, bon générateur de nombres pseudo-aléatoires moderne, bonne technique de distribution, bonne technique d'émulation, bonne bibliothèque mathématique et/ou bonne technique de réduction, un modèle de programmation parallèle adapté à la répétabilité numérique*}. Pour obtenir une répétabilité numérique entre deux architectures de la famille Intel (Xeon classique et Xeon Phi), il est essentille de bien maîtriser les options de compilation avec icc/icpc.

De plus, dans une nouvelle publication de cette aproche (Hill et al. 2017), nous avons mentionné l'importance de la phase de réduction. Il faut de plus utiliser une bonne bibliothèque mathématique sur des plate-formes ou environnements d'exécution différents si l'on veut faire des comparaisons équitables. Nous avons présenté plusieurs bibliothèques qui gèrent bien l'impact du calcul en virgule flottante en ce qui concerne la gestion des arrondis afin de garantir la reproductibilité numérique et/ou une meilleure précision numérique. Nous recommandons aussi les bibliothèques dérivées de la famille BLAS (cf. voir la partie II.2 du chapitre 4). S'il n'y a pas de bonne bibliothèque mathématique installée sur les environnements d'exécutions cibles, il faut au moins mettre en place une bonne technique de réduction qui contrôle l'impact du calcul en virgule flottante et garantit la repétabilité numérique.

Enfin, nos approches, nos implémentations et l'outil proposé constituent une approche cohérente pour lancer des simulations stochastiques en parallèle tout en garantissant la reproductibilité numérique.

Chapter 6 - Conclusion générale

Cette thèse traite d'une question actuelle et très importante dans de nombreux domaines. Nous avons abordé la notion de reproductibilité numérique et nous avons plus particulièrement étudié le cas des simulations stochastiques parallèles sur architecture multicores et manycores. L'évolution du matériel et du logiciel ont impacté la reproductibilité des résultats des expériences numériques. Nous avons donc fait le choix de nous concentrer dans cette thèse sur la capacité à répéter les expériences computationnelles sur des matériels de calcul à hautes performances et ce, sans avoir des objectifs de performance. En effet, le principal objectif a été d'abord pour nous de pouvoir retrouver des résultats identiques et lorsque ce n'était pas le cas, d'essayer d'expliquer les raisons de cette non-reproductibilité (répétabilité). Pour réduire les temps de calcul, nous avons identifé un cas d'étude simple qui met déjà en évidence les problèmes de répétabilité sur des simulations stochastiques parallèles.

Les méthodes numériques employées pour la parallélisation d'applications stochastiques sont souvent méconnues (Hellekalek 1998) (Hill et al. 2013). De plus, des résultats différents peuvent survenir pour la même expérience numérique du fait de l'impact des nouvelles architectures, notamment de l'usage d'accélérateurs de calcul (de type many-cœurs à base par exemple de Xeon Phi ou à base de GP-GPU), mais aussi des compilateurs, des systèmes d'exploitation ou encore de l'ordre d'exécution en parallèle des opérations en arithmétique flottantes (Dao et al 2016). Nous soulignons dans cette thèse qu'il est essentiel d'avoir conscience de ce problème de reproductibilité numérique lorsque l'on considère les résultats des simulations stochastiques parallèles. Dans ce contexte, nous avons proposé une méthode adaptée à la production de résultats reproductibles, visant à accroître la confiance dans les résultats publiés par la communauté scientifique du calcul stochastique à haute performance.

Les travaux réalisés pendant cette thèse ont permis d'établir un état de l'art poussé des causes de non reproductibilité numérique. Les principaux obstacles ne permettant pas de répéter les résultats de simulations stochastiques utilisant des systèmes de calcul à hautes performances sont identifiés. Les apports de la reproductibilité numérique pour la science expérimentale computationnelle sont également mis en avant. Ce travail de synthèse bibliographique a permis d'une part de proposer une méthode simple, capable de vérifier dans divers contexte la reproductibilité numérique et la portabilité des générateurs modernes de nombres pseudo-aléatoires. Nous avons aussi contribué à la détection de corrélations entre flux parallèles issus de générateurs de nombres pseudo-aléatoires. Nous avons aussi proposé un outil logiciel capable de répéter et de reproduire les résultats

numériques de simulations stochastiques parallèles indépendantes avec une approche *Single Program Multiple Data* (SPMD) parfaitement adaptée à l'approche *Multiple Replication In Parallel* (MRIP). D'autre part, nous avons proposé une suite de recommandations permettant d'obtenir la reproductibilité numérique de simulation stochastiques utilisant le calcul à haute performance.

1 - Contributions

Dans un premier temps, nous avons réalisé un état de l'art détaillé des problèmes de reproductibilité numérique. Tout d'abord, nous avons identifié les principales causes de non-reproductibilité d'une simulation stochastique parallèle et classé ces causes en cinq sources principales. Ces causes sont liées à la culture de publications scientifiques, au matériel utilisé et aux processeurs à mode d'exécution dans le désordre, à l'arithmétique flottante non associative, aux logiciels sous-jacents et enfin, aux techniques de distribution et de parallélisation des flux stochastiques. Ensuite, nous avons identifié et présenté les trois apports principaux de la reproductibilité numérique pour la science expérimentale computationnelle. Ces apports servent à tester les résultats de simulation d'un auteur, à juger de la pertinence des résultats publiés par un autre auteur, à promouvoir le développement scientifique. Enfin, nos travaux aident à préciser la terminologie liée à la répétabilité et à la reproductibilité scientifique et numérique.

Nous nous sommes ensuite penchés sur la source de la reproductibilité numérique pour des simulations stochastiques de type Monte Carlo. Il s'agit bien sûr du générateur de nombres pseudo-aléatoires. Ces générateurs permettent de simuler les variables aléatoires d'un modèle stochastique. En nous basant sur notre compréhension de la reproductibilité numérique abordée dans l'étape précédente, nous avons proposé une méthode en trois étapes pour vérifier la reproductibilité numérique et la portabilité des meilleurs générateurs de nombres pseudo-aléatoires utilisés dans les simulations stochastiques modernes. Nous avons pu détecter que le générateur TinyMT (issu de la famille des Mersenne Twister) n'est pas reproductible dans certains cas atypiques et ce en raison du choix du compilateur, du système d'exploitation ou encore du fait d'exécuter l'application dans certains environnements virtuels. De même, nous avons pu identifier que le générateur non linéaire MLFG_64 (conçu pour des machines 64 bits) n'est pas portable sur des compilateurs 32 bits. De ce fait, pour la simulation stochastique, nous recommandons principalement les générateurs suivants : MRG32k3a ; MT19937 ; WELL512a.

Nous avons poursuivi ensuite nos travaux autour des générateurs en étudiant le problème des corrélations entre flux stochastiques parallèles. Il ne s'agit pas à proprement parler d'une source de

non-reproductibilité, mais ce facteur peut fortement influencer les résultats obtenus. Pour tester ces corrélations, nous avons adopté une nouvelle approche multivariée proposée par Ismay (Ismay 2013). Nous avons validé cette approche avec un éventail de générateurs de nombres pseudo-aléatoires, en utilisant aussi bien des « mauvais » générateurs (certains encore présents dans des logiciels scientifiques) que des « bons » générateurs qui sont dit « *Crush resistant* » en référence à la batterie de tests statistiques TestU01 de Pierre l'Ecuyer (L'Ecuyer et Simard 2007). Nous aboutissons à des résultats qui établissent l'existence systématique de corrélations entre flux parallèles issus de générateurs de nombres pseudo-aléatoires, y compris des générateurs réputés d'excellente qualité. Bien que la méthode et l'outil proposés connaissent des limites sur le nombre de flux testés (inférieur ou égal à 128 flux parallèles), sa puissance statistique est très satisfaisante puisque cette méthode arrive à détecter des corrélations pour des générateurs *Crush resistant*, alors qu'en utilisant les méthodes monovariées précédemment proposées par TestU01, aucune corrélation n'avait jusques là été détectée entre des flux parallèles générés par ces PRNGs.

Nous avons aussi proposé quelques bonnes pratiques permettant de reproduire complètement les résultats d'une simulation stochastique parallèle utilisant un système de calcul à hautes performances, tout en comparant ces résultats avec une exécution séquentielle de référence. Ceci suppose la disponibilité des données et des codes, mais également de concevoir un code reproductible dès le début du processus de développement d'une simulation. Concevoir la simulation séquentielle de référence en pensant aux exécutions parallèles et en préconisant l'utilisation de générateurs ou de flux indépendants pour chaque objet ou processus stochastique est une clé indispensable pour l'obtention de simulations reproductibles. Assurer la reproductibilité d'une simulation dans plusieurs contextes d'exécution différents est aussi un challenge qui suppose des outils, notamment des compilateurs de qualité, mais il convient aussi d'éviter d'utiliser dans une même simulation des accélérateurs matériels différents. En effet, l'hétérogénéité des ressources matérielles au sein d'un même calcul est un facteur de non reproductibilité numérique, notamment quand les constructeurs annoncent eux-mêmes qu'il n'y aura pas de reproductibilité bit à bit entre architectures différentes (Ex : x86 et k1Om pour les Xeon Phi).

Nous avons considéré également la relation qui peut exister entre le choix du modèle de programmation parallèle et la reproductibilité numérique pour une simulation stochastique. Pour les simulations stochastiques qui ne sont pas pensées à l'origine pour travailler avec des flux indépendants de nombres pseudo-aléatoires, il est fréquent d'utiliser OpenMP ou Pthreads pour obtenir rapidement une version parallèle. L'utilisation de la bibliothèque MPI ne se justifie pas, étant donné que les processus parallèles ne communiquent pas entre eux. Cependant, si l'implémentation

avec les bibliothèques OpenMP ou Pthreads se révèle aisée, il est par contre vraiment difficile d'obtenir des résultats reproductibles et une performance satisfaisante sur des processeurs multi-cœurs qui sont en fait des systèmes parallèles symétriques de type MIMD. Le principal problème à éviter pour obtenir des résultats reproductibles est celui des accès concurrents (*race conditions*) des threads stochastiques. Pour obtenir des résultats numériquement reproductibles, on doit détecter ce problème et on peut utiliser aussi des fonctions *omp_set_lock() & omp_unset_lock()* dans le cas d'OpenMP, et *pthread_mutex_lock() & pthread_mutex_unlock()* dans le cas de la bibliothèque Pthreads.

Enfin, nous avons proposé une implémentation qui a été pensée pour la reproductibilité des résultats en assurant l'indépendance de flux de nombres pseudo-aléatoires dès la conception de l'application. Cette implémentation permet d'obtenir facilement la répétabilité et la reproductibilité numérique des résultats issus de simulations stochastiques parallèles de type Monte Carlo. Elle se compose de trois étapes principales et d'une étape supplémentaire de tests pour la reproductibilité numérique. Avec cette implémentation, nous avons créé un outil en C et en Bash shell, nommé RSPR, qui utilise la technique MRIP, le progiciel hwloc, le générateur MT19937 et la technique de parallélisation de « découpage par bloc », ou *sequence splitting*, avec un algorithme de JumpAhead. Cet outil permet également, pour une simulation de Monte Carlo, de calculer le nombre de réplications nécessaires pour obtenir une précision fixée à un niveau de confiance à 99%. Cet outil a été appliqué au cas d'une simulation démographique. Nos résultats de simulation sont reproductibles numériquement sur les architectures multi-cœur et many-cœurs de famille Intel Xeon. Nous nous sommes ensuite intéressés au problème de performance, en termes de temps de calcul.

2 - Perspectives

A l'issue de ces travaux et des résultats présentés, nous proposons des perspectives qui sont également des avancées dans le domaine du calcul à haute performance en ce qui concerne la reproductibilité numérique des simulations stochastiques parallèles. Travailler à l'obtention de résultats répétables à un coût en termes de performances. Nous n'avons pas évalué cet aspect performance, cela peut-être une des premières perspectives de ce travail.

Dans la thèse de Pierre Schweitzer (Schweitzer 2015), la perte de performance pure était de l'ordre de 20% du temps de calcul pour avoir des résultats répétables. Par contre le code « rapide » donnait

sur des simulations de taille importante (plusieurs jours de calcul) des résultats faux sur plusieurs ordres de grandeur, quel est alors l'intérêt d'une comparaison ?

Comme seconde perspective, s'il devient très difficile à l'avenir de travailler de manière fiable sur l'arithmérique flottante à l'échelle de milliards de milliards d'opérations par seconde (Exascale), on peut envisager un retour à des opérations sur des types beaucoup moins performants de type « BigDecimal » pour avoir un contrôle complet sur la gestion des arrondis.

Pour des simulations parallèles utilisant OpenMP ou Pthreads, il faut apporter un soin particulier pour s'assurer de la reproductibilité numérique de simulations stochastiques utilisant des systèmes de calcul à hautes performances. Il s'agit principalement de veiller aux résultats fournis en fonction des drapeaux de compilation, mais aussi d'éviter la non-reproductibilité en raison la réduction des opérations en virgule flottante.

On peut facilement réaliser une version de code parallèle *via* l'interface de programmation OpenMP, mais il est vraiment difficile de déterminer les raisons de la non-répétabilité ou de la non-reproductibilité numérique entre la version parallèle et la version séquentielle ou entre différentes exécutions parallèles. Comme perspective à plus long terme, on peut se proposer de concevoir des algorithmes et de développer un outil ouvert qui les implémente pour détecter automatiquement la non-répétabilité et la non-reproductibilité numérique dans certains cas (par exemple, dans le cas d'un changement du nombre de threads). Toujours à plus long terme, on peut envisager des outils qui aident à tracer les causes de non reproductibilité pour des codes parallèles utilisant l'interface de programmation OpenMP.

Une solution d'ordonnancement des tâches ou des blocs de tâches quand on utilise la technique MRIP permet d'obtenir d'une part de meilleures performances sur les processeurs modernes, et permet d'autre part d'obtenir la reproductibilité numérique escomptée, parce que cette technique, que nous avons implémentée avec l'outil RSPR, utilise des processus lourds.

En ce qui concerne les perspectives sur l'outil RSPR, redonnons tout d'abord un peu de contexte, nous utilisons le progiciel hwloc (Goglin 2014) et nous proposons un algorithme qui lance un bloc d'exécutions/processus adapté au nombre de cœurs logiques de la machine utilisée. Cet outil attend la terminaison de tous les processus créés avant de lancer un autre bloc. Cette approche avec hwloc permet de garantir une équité entre les cœurs et aussi un équilibrage de la charge de calcul à l'exécution. Cet algorithme est statique et il ne rééquilibre pas la charge en fin de calcul (les différences de charges entre cœurs sont minimes). Dans les perspectives, on pourrait penser à un

algorithme plus dynamique ou à une autre implémentation pour gagner un peu de temps d'exécution.

Les performances des cœurs logiques (ou hardware threads) ne sont pas équivalentes à celle des cœurs physiques (et de loin). Or, même un système d'exploitation de type Unix verra toutes les unités de traitement comme des cœurs de calculs, ce qui fausse donc les mesures de *speedup* habituelles si on se réfère aux attentes des lois d'Amdahl ou de Gustafson. Si nous avons mis l'accent sur la reproductibilité numérique, il ne faut pas négliger dans les perspectives de garantir le maximum de performances pour les exécutions parallèles de simulations stochastiques.

L'outil RSPR utilise un bon générateur MT19937 et la technique de "découpage par bloc" avec la méthode de JumpAhead pour distribuer un générateur en parallèle. La méthode de JumpAhead est complexe à mettre en œuvre selon (Haramoto et al. 2008), cette complexité est gérée par notre outil. Si on détermine un nombre de réplications pour une simulation selon notre algorithme détaillé précédemment au chapitre 5, il peut se produire qu'il faille consacrer un peu de temps pour générer des flux supplémentaires de nombres pseudo-aléatoire afin d'associer ces flux indépendants aux nouveaux processus. Actuellement sur nos tests nous avons pré-calculés 100 000 statuts, ce qui laisse déjà un peu de marge. Dans ce cas, on peut essayer également une autre technique de distribution des générateurs de nombres pseudo-aléatoires en parallèles avec d'autres bons générateurs, par exemple la bibliothèque SPRNG (Mascagni et Srinivasan 2000) qui est conçue pour utiliser des générateurs "paramétrables" de nombres pseudo-aléatoires pour fournir plusieurs flux au éléments de calcul parallèle. Elle contient de nombreux générateurs anciens à éviter. Nous conseillons le fleuron de cette bibliothèque qui est le générateur MLFG sur 64 bits. Une autre approche serait d'utiliser le logiciel Dynamic Creator (Matsumoto et al. 2000), il permet de créer différents « petits » générateurs de la famille MT qui peuvent être affectés chacun à un élément de calcul (PE). Le generateur MRG32k3a avec RngStream (L'Ecuyer et al. 2015a), mais les performances sont parfois jusqu'à 15 fois moins bonnes que Mersenne Twister sur le même matériel. On, peut prévoir de mettre à jour une autre version de l'outil RSPR avec ces générateurs et ces approches.

Nous ne pouvons pas nous attendre à une reproductibilité bit à bit entre l'architecture multicore Intel Xeon classique et l'architecture manycore Intel Xeon Phi du fait même de leur conception matérielle. Ce problème a déjà été signalé dans (Corden 2013) et on le retrouve même si, sur ces deux architectures, l'on utilise les drapeaux et les options de compilation qui demandent les plus de précision au compilateur C de Intel –icc. Le problème est encore plus aigu si l'on utilise des architectures hybrides à base de CPU et de GP-GPU. Pour des architectures homogènes, il existe des logiciels et des plates-formes qui permettent de faire des expériences computationnelles

reproductibles comme CDE (Guo 2011), ReproZip (Chirigati et al. 2013), Docker (Boettiger 2015), Umbrella (Meng et Thain 2015), REP (Likhomanenko et al. 2015) et Redo (Jimenez et al. 2014). Ces outils nous permettent d'envisager la résolution de certains problèmes de non-reproductibilité des simulations stochastiques parallèles sur les architectures multicores et manycores de la famille Intel Xeon. Nous espérons que ce type d'outils puisse permettre une étude de l'impact des facteurs matériels et/ou des facteurs logiciels sur la reproductibilité numérique de nos applications stochastiques. Si ce n'est pas le cas, il est envisageable de développer un logiciel ou une machine virtuelle qui serait indépendante des architectures matérielles, mais qui fonctionnerait quand même efficacement sur les accélérateurs de calcul.

On peut à l'avenir penser à une évolution vers le « *co-design* » telle qu'il se pratique pour la conception de supercalculateurs, où l'on prend en compte le matériel et le logiciel. De même, sur les nouvelles architectures du calcul à haute performance, ce type d'approche sera nécessaire pour obtenir des résultats numériquement reproductibles pour des simulations parallèles stochastiques de type Monte Carlo ou autre. Le nombre de causes entraînant la non-reproductibilité numérique des calculs ne fait que croître, que ce soit du côté logiciel ou du côté matériel. De ce fait une attention particulière devra être portée à l'avenir à cet aspect important de l'informatique. Sans reproductibilité numérique, l'explicabilité et la possibilité de déboguer les codes de calcul (et donc de vérifier les simulations et la pertinence des modèles) se perdraient.

Nous avons présenté quatre recommandations pratiques pour assurer la reproductibilité d'une simulation stochastique parallèle utilisant un système du calcul à haute performance, mais ces recommandations ne peuvent pas être considérées comme une norme complète (Dao et al. 2016). Pour aller plus loin, on peut penser à construire une norme, un processus ou un ensemble des règles communes pour les simulations stochastiques (l'utilisation de bibliothèque mathématique, l'approche de programmation, le langage de programmation utilisé et ses types de données, etc.). De même, des outils seront nécessaires pour aider à l'évaluation de la répétabilité et de la reproductibilité numérique des simulations stochastiques parallèles (avec l'ensemble de l'architecture matérielle utilisée, le compilateur et sa version, les options de compilation, etc.). Cette étape d'évaluation est nécessaire, parce qu'on ne peut pas garantir qu'une simulation stochastique (incluant les versions séquentielle et parallèle) soit toujours reproductible sur toutes les environnements d'exécutions, et particulièrement car on ne peut pas connaître à l'avance l'impact des nouvelles architectures et des nouveaux accélérateurs utilisés pour le calcul à haute performance.

Références bibliographiques

Références

Amdahl, G.M. (1967): « Validity of the single processor approach to achieving large scale computing capabilities », in AFIPS'67 proceeding, 1967, pp. 483-485.

http://www-inst.eecs.berkeley.edu/~n252/paper/Amdahl.pdf

Amundsen, L.; Landrø, M. and B. Arnsten (2015): « Supercomputers for beginners – Part I », GeoExpro, 12(3), 2015, pp. 50-52.

http://assets.geoexpro.com/uploads/5d5ac83b-3ea6-40ca-ad42-4092913100b0/GEO_ExPro_v12i5.pdf

Arthur, J.V.; Merolla, P.A.; Akopyan, F.; Alvarez, R.; Cassidy, A.; Chandra, S.; Esser, S.; Imam, N.; Risk, W.; Rubin, D.; Manohar, R. and D.S. Modha (2011): « Building block of a programmable neuromorphic substrate: A digital neurosynatic core », the 2012 international joint conference on Neural networks (IJCNN), 2012, pp.1-8. http://www.modha.org/papers/IJCNN%202012.pdf

Bailey, D.H.; Browein, J.M and V. Stodden (2015): « Facilitating reproducibility in scientific computing: Principles and practice ». John Wiley and Sons, Part III, 2015, pp. 205-232.

http://www.davidhbailey.com/dhbpapers/reprod.pdf

Banks, J. (1998): « Handbook of simulation: principles, methodology, advances, applications, and practice ». Wiley, Chapter 1, 1998, pp. 3-30.

Barash, L.Y. and L.N. Shchur (2014): « PRAND: GPU accelerated parallel random number generation library: Using most reliable algorithms and applying parallelism of modern GPUs and CPUs », Computer physics communications, 185, 2014, pp. 1343-1353.

http://arxiv.org/pdf/1307.5869v2.pdf

Bloch, J. (2008): « Effective Java™ : Programming language guide, 2nd edtion», Addison-Wesley professional, Boston, 2008.

Boettiger, C. (2015): « An introduction to Docker for reproducibility research, with examples from the R environment », ACM SIGOPS operating system review – special issue on repeatability ad sharing of experimental artifacts, 49(1), 2015, pp. 71-79. http://arxiv.org/pdf/1410.0846.pdf

Brugger, C.; Weithoffer, S.; de Schryver, C.; Wasenmuller, U. and N. Wehm (2014): « On parallel random number generation for accelerating simulation of communication systems ». Advances in Radio science, vol. 12, 2014, pp. 75-81.

http://www.adv-radio-sci.net/12/75/2014/ars-12-75-2014.pdf

Chohra, C.; Langlois, P. and D. Parello (2014): « Efficiency of reproducibility level 1 BLAS », in SCAN 2014 post-conference proceedings, lecture notes of Computer science, 2014, pp.1-10.

http://hal-lirmm.ccsd.cnrs.fr/lirmm-01101723/document

Chirigati, F.; Shasha, D. and J. Freire (2013): « ReproZip: using provenance to support computational reproducibility », TaPP'13 proceedings of the 5th USENIX, workshop on the theory and practice of provenance, article no. 1, 2013. http://vgc.poly.edu/~fchirigati/papers/chirigati-tapp2013.pdf

Craven, S. and P. Athanas (2007): « Examining the viability of FPGA supercomputing », EURASIP journal on Embedded systems, vol 1, p.13.

Coddington, P.D. (1997): « Random number generators for parallel computers ». The NHSE Review, 1997, Volume Second issue.

http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.22.9978&rep=rep1&type=pdf

Coddington, P.D. and S.H. Ko (1998): « Techniques for empirical testing of parallel random number generators », northeast Parallel Architecture Center, 1998.

http://surface.syr.edu/cgi/viewcontent.cgi?article=1091&context=npac

Coddington, P.D. and A.J. Newell (2004): « Japara - a java parallel random number generator library for high-performance computing », Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS'04), IEEE Computing Society, 2004, p. Article 156

http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=1303143

Colfax International (2013): « Parallel programming and optimization with Intel Xeon Phi coprocessors: Hand book on the development and optimization of parallel applications for Intel Xeon processors and Intel Xeon Phi coprocessors », Colfax Interanational, 2013, 500 pages.

Collange, S.; Defour, D. ; Graillat, S. and R. Iakymchuk (2015): « Numerical reproducibility for the parallel reduction on Multi- and Many-core architectures », Journal on Parallel computing, Elsevier, pp. 83-97, 2015. https://hal.archives-ouvertes.fr/hal-00949355v4/document

Coquillard, P. and D.R.C. Hill (1997): « Modélisation et simulation des Ecosystèmes », Masson, 1997, 273 pages.

Dalle, O. (2012): « On reproducibility and traceability of simulations », proceedings of the simulation conference (WCS), 2012, pp. 1-12.

http://www-sop.inria.fr/members/Olivier.Dalle/wiki/uploads/Main/Dalle_WSC12.pdf

Dao, V.T.; Maigne, L.; Mazel, C.; Breton, V.; Nguyen, H.Q. and D.R.C. Hill (2014a): « Numerical reproducibility, portability and performance of modern pseudo random number generators: Preliminary study for parallel stochastic simulations using hybrid Xeon Phi computing processors », Extended paper in Proceeding of the 28th European Simulation and Modeling, EMS'2014 Oct. 22-24 2104, pp. 80-87.

Dao, V.T; Breton, V.; Nguyen, H.Q. and D.R.C. Hill (2014b) : « Numerical reproducibility for high performance computing: The case of parallel stochastic simulations », in Proceeding of the 9th International Student Conference on Advanced Science and Technology, ICAST 2014 Dec. 11-12 2014, pp. 139-140.

Dao V.T., Breton V., Nguyen H.Q. and Hill D.R.C (2016): « La reproductibilité des simulations stochastiques parallèles et distribuées utilisant le calcul à haute performance », Journées DEVS Francophone 2016 : Théorie et applications, workshop RED, ISBN978.2.36493.539.63, pp. 109-117.

Demmel, J. and H. D. Nguyen (2013): « Fast Reproducible Floating-Point Summation », in IEEE Symposium on Computer Arithmetic (ARITH), 2013, pp. 163-172.

http://www.eecs.berkeley.edu/~hdnguyen/public/papers/ARITH21_Fast_Sum.pdf

Diaconu, M.; Puscasu, R. and A. Stancu (2003): « Monte Carlo methods in sequential and parallel computing of 2D and 3D Ising model», Journal of Optoelectronics and advanced materials, 5(4), 2003, pp. 971-976. http://joam.inoe.ro/arhiva/pdf5_4/Diaconu.pdf

Diaz, J.; Caro, C.M. and A. Nino (2012): « A survey of parallel programming models and tools in the multi and may-core era », IEEE transactions on parallel and distributed systems, 23(8), 2012, pp. 1369-1386.

Diethelm, K. (2012): « The Limits of Reproducibility in Numerical Simulation ». In Computing Science and Engineering, 14(1), 2012, pp. 64-72.

http://www.computer.org/cms/ComputingNow/homepage/2012/0312/W_CS_TheLimitsofReproducibilityinNumericalSimulation.pdf

Dongarra, J. (2004): « Trends in High performance computing », the Computer journal, 47(4), 2004, pp. 399-403.

http://www.netlib.org/utk/people/JackDongarra/PAPERS/150_2004_trends-in-high-performance-computing.pdf

Dongarra, J.; M.A. Heroux and P. Luszczek (2015): « HPCG Benchmark: a new metric for ranking high performance computing systems », International Journal on High Performance Computing Applications, published online before print, 2015.

Eichenberger, A.E.; Terboven, C.; Wong, M. and D. an Mey (2012): « The design of OpenMP thread affinity », IWOMP 2012, LNCS 7312, Springer-Verlag Berlin Heidelberge, pp. 15-28.

Einarsson, B. (2005): « What can go wrong in scientific computing? », in Accuracy and reliability in scientific computing, SIAM, 2005, pp. 3-12.

Feitelson, D.G. (2015): « From Repeatability to Reproducibility and Corroboration », ACM SIGOPS Operating Systems Review – Special Issue on Repeatability and Sharing of Experimal Artifacts, Vol 49, Issue 1, January 2015, pp. 3-11.

http://www.cs.huji.ac.il/~feit/papers/Repeat15SIGOPS.pdf

Foster, I.T. (1995): « Designing and building parallel programs: Concepts & tools for parallel software engineering », Addison-Wesley, 1995. http://www.mcs.anl.gov/~itf/dbpp/text/book.html

Glatard, T.; Lewis, L.B.; Ferreira da Silva, R.; Adalat, R.; Beck, N.; Lepage, C.; Rioux, P.; Rousseau, M.E.; Sherif, T.; Deelman, E.; Khalili-Mahani, N. and A.C. Evans (2015): « Reproducibility of neuroimaging analyses across operating systems », in Frontiers in Neuroinformatics, Vol 9, Article 12, April 2015, pp. 1-12.

http://journal.frontiersin.org/article/10.3389/fninf.2015.00012/full

Goglin, B. (2014): « Managing the topology of heterogeneous cluster nodes with hardware locality (hwloc) », International conference on high performance computing & simulation – HPCS 2014, Jul 2014, Bologna, Italy, IEEE 2014. https://hal.inria.fr/hal-00985096/document

Goldberg, D. (1991): « What every computer scientist should know about floating-point arithmetic », in ACM Computing Surveys, Vol. 23, No. 1, March 1991, pp. 5-48.

http://perso.ens-lyon.fr/jean-michel.muller/goldberg.pdf

Guo, P.J. (2011): « CDE: Run any linux application on-demand without installation », LISA'11 proceeding of the 25th international conference on large installation system administration, 2011, pp. 2-2.

http://www.pgbovine.net/publications/CDE-create-portable-Linux-packages_LISA-2011.pdf

Gustafson, J.L. (1988): « Reevaluating Amdahl's law », communications of the ACM magazine, 31(5), 1988, pp. 532-533.

http://www.qatar.cmu.edu/~msakr/15346-s13/resources/GB/Reevaluating%20Amdahl's%20law%20-gustafson.pdf

Gramma, A.; Gupta, A.; Karypis, G. and V. Kumar (2003): « Introduction to parallel computing, second edition », Pearson – Addison Wesley, 2003, 636 pages. Cité chapitre 5.

Gropp, W.D. (2005): « Issues in Accurate and reliable use of parallel computing in numerical programs », in Accuracy and reliability in scientific computing, SIAM, 2005, pp. 253-263.

Ha, O., Y. Kim, M. Kang, and Y. Jun, "Empirical Comparison of Data Race Detection Tools for OpenMP Programs," Proceedings of International Conference on Grid and Distributed Computing (GDC 2009), Jeju Island, Korea, Communications in Computer and Information Science (CCIS), 63:108-116, Springer-Verlag, December 2009.

Haramoto, H.; Matsumoto, M.; Nishimura, T.; Panneton, F. and P. L'Ecuyer (2008): « Efficient jump ahead for F_2-Linear random number generators », Informs journal on computing, 20(3), 2008, pp. 385-390. http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/ARTICLES/jumpf2-printed.pdf

He Y. and C.H.Q. Ding (2001): « Using accurate arithmetics to improve numerical reproducibility and stability in parallel applications », the Journal of supercomputing, Springer, 18(3), 2001, pp. 259-277.

Hechenleitner, B. and K. Entacher (2003): « Pitfalls when using parallel streams in OMNeT++ simulations ». Proceedings of the Inter-domain Performance and Simulation (IPS) Workshop, Austria, 2003, pp. 11-19.

http://www.ist-intermon.org/workshop/papers/01_01_hechenleitner-ips2003.pdf

Hellekalek, P. (1998): « Don't trust parallel Monte Carlo! », in PADS' 98, Proceedings of the twelfth workshop on Parallel and Distributed Simulation, pp. 82-89, 1998.

http://www.cs.odu.edu/~yaohang/cs714814/Assg/DontTrustParallelMonteCarlo.pdf

Hennessy, J.L. and D.A. Patterson (2007): « Computer architecture: A quantitative approach, Fourth edition », Morgan Kaufmann, 2007, 704 pages.

http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.115.1881&rep=rep1&type=pdf

Hey, A.J.G. (1997): « High performance computing – past, present and future », computing & control engineering journal, 8(1), 1997, pp. 33-42. https://cds.cern.ch/record/400329/files/p217.pdf

Hill, D.; Passerat-Palmbach, J.; Mazel, C. and M.K. Traore (2013): « Distribution of Random Streams for Simulation Practitioners. Concurrency and Computation: Practice and Experience », 25(10), 2013, pp. 1427-1442.

Hill, D.R.C. (2014): « Simulations stochastiques et calcul à haute performance: la "parallélisation" des générateurs de nombres pseudo-aléatoires», chapitre 23, modéliser & simuler – tome 2, 2014, pp. 725-750.

Hill, D.R.C. (2015): « Parallel Random Numbers, Simulation, Science and reproducibility ». IEEE/AIP - Computing in Science and Engineering, vol. 17, no. 4, 2015, pp. 66-71.

Hill, D.R.C.; Congo, F.Y.P and V.T. Dao (2015): « Numerical reproducibility for parallel stochastic simulation "Exascale ready" », in Numerical reproductibility at Exascale workshop NRE2015 – SC15, Austin, USA, 2015.

Hill, D., Van Toan Dao, Claude Mazel, Vincent Breton, « Répétabilité et reproductibilité numérique : Constat, conseils et bonnes pratiques pour le cas des simulations stochastiques parallèles et distribuées », Techniques et Science Informatique, en cours de relecture, 2017.

Hofner, B.; Schmid, M. and L. Edler (2015): « Reproducible research in statistics: A review and guidelines for the Biometrical Journal », Biometrical journal, 2015, pp. 1-12.

http://onlinelibrary.wiley.com/doi/10.1002/bimj.201500156/epdf

Hsu, J. (2014a): « IBM's new brain », Spectrum IEEE, 2014, pp. 17-19.

http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6905473

Iakymchuk, R.; Defour, D.; Collange, S. and S. Graillat (2015): « ExBLAS: Reproducible and accurate BLAS library », workshop on Numerical reproducibility at exascale, supercomputing 2015.

https://hal.archives-ouvertes.fr/hal-01202396/document

Ismay C. (2013): « Testing independence of parallel pseudorandom number streams incorporating the data's multivariate », Phd thesis in Arizona State university, August 2013.

http://repository.asu.edu/attachments/110682/content/Ismay_asu_0010E_13098.pdf

Jeffers, J. et J. Reinder (2013) : « Intel Xeon Phi Coprocessor High Performance Programming », MORG.KAUFF., Chapter 1, 2013, pp.1-21.

Jézéquel, F.; Lamotte, J.L. and I. Said (2015): « Estimation of numerical reproducibility on CPU and GPU ». Proceeding of Computer science and information systems, 2015, pp. 675-680.

http://www-pequan.lip6.fr/~jezequel/ARTICLES/article_CANA2015.pdf

Juristo, N. and O.S. Gómez (2012): « Replication of Software engineering experiments ». Empirical software engineering and verification book, Springer-Verlag, 2012, pp. 60-88.

Karl, A.T.; Eubank, R.; Milovanovice, J.; Reiser, M. and D. Young (2014): « Using RngStreams for parallel random number generation in C++ and R ». Comput Stat (2014), 29, pp. 1301-1320.

http://arxiv.org/pdf/1403.7645.pdf

Kim, T.S. and Y.K. Yang (2006): « On the initial seed of the random number generators ». Kangweaon-Kyungki math. Jour. 14 (1), 2006, pp. 85-93.

http://www.kkms.org/kkms/vol14_1/14110.pdf

Kovacevic, J. (2007): « How to encourage and publish reproducible research ». IEEE Internation Conference on Acoustics, Speech and Signal Processing, vol 4, 2007, pp.1273-1276. Cité page 4.

http://ivrlwww.epfl.ch/reproducible_research/ICASSP07/Kovacevic07.pdf

Krishnaiyer, R.; Kultursay, E.; Chawla, P.; Preis, S.; Zvezdin, A. and H. Saito (2013): « Compiler-based data prefetching and streaming non-temporal store generation for the Intel Xeon Phi coprocessor », Parallel and distributed processing symposium workshops & PhD forum, 2013 IEEE 27th international, pp. 1575-1586.

https://software.intel.com/sites/default/files/managed/22/a3/mtaap2013-prefetch-streaming-stores.pdf

Kroese, D.P.; Brereton, T.; Taimre, T. and Z.I. Botev (2014): « Why the Monte Carlo method is so important today », in Wiley Interdisciplinary Reviews: Computational Statistics, 6(6), 2014, pp. 386-392. http://www.maths.uq.edu.au/~kroese/ps/whyMCM_final.pdf

Kurkowski, S.; Camp, T. and M. Colagrosso (2005): « MANET simulation studies: The incredibles ». ACM SIGMOBILE Mobile computing and communications review – Special Issue on Medium access and Call admission control algorithms for Next generation wireless networks, 9(4), 2005, pp.50-61.

https://www.cs.auckland.ac.nz/courses/compsci742s2c/resources/p50-kurkowski.pdf

Langlois, P.; Nheili, R. and C. Denis (2015) : « Numerical reproducibility: Feasibility issues », IEEE Internation Conference on New Technologies, Mobility and Security (NTMS), 2015, pp.1-5. Cité page 1. https://hal.inria.fr/lirmm-01141852/document

L'Ecuyer, P. (1990): « Random number for simulations », Communications of the ACM, Oct 1990, Vol 33, No. 10, pp. 85-97. http://www.iro.umontreal.ca/~lecuyer/myftp/papers/cacm90.pdf

L'Ecuyer P. (1999a): « Good parameters and implementations for combined multiple recursive random number generators ». Operations Research Vol. 47, No. 1 (Jan. - Feb., 1999), pp. 159-164. http://www.xavierdupre.fr/biblio/monte-carlo_poly/generateur_ecuyer.pdf

L'Ecuyer, P. (1999b): « Tables of maximally equidistributed combined LFSR generators », Mathematics of Computation, 68(225), 1999, pp. 261-269.

http://www.ams.org/journals/mcom/1999-68-225/S0025-5718-99-01039-X/S0025-5718-99-01039-X.pdf

L'Ecuyer, P. and R. Touzin (2000): « Fast combined multiple recursive generators with multipliers of the form a = $\pm 2^q \pm 2^r$ », in Proceedings of the 2000 Winter simulation conference, IEEE Press, Piscataway, NJ, 2000, pp. 683-689. http://www.informs-sim.org/wsc00papers/090.PDF

L'Ecuyer, P. and R. Simard (2007): « TestU01: A C library for empirical testing of random number generators », ACM Transactions on Mathematical Software (TOMS), 33(4), article no 22, 2007.

http://www.iro.umontreal.ca/~lecuyer/myftp/papers/testu01.pdf

L'Ecuyer, P.; Munger, D.; Oreshkin, B. and R. Simard (2015a): « Random numbers for parallel computers: requirements and methods, with emphasis on GPUs ». Les cahiers du GERAD, G-2015-35, 2015. https://www.gerad.ca/en/papers/G-2015-35/view

Li-Thiao-Té, S. (2012): « Literate program execution for reproducibility research and executable papers », Procedia computer science, vol. 9, 2012, pp. 439-448.

Likhomanenko, T.; Rogozhnikov, A.; Baranov, A.; Khairullin, E. and A. Ustyuzhanin (2015): « Reproducible experiment platform », 21st International Conference on Computing in High Energy Physics - CHEP2015. http://arxiv.org/pdf/1510.00624v1.pdf

Limare, N. and J.M. Morel (2011): « The IPOL initiative: Publishing and testing algorithms on line for reproducible research in image processing », Procedia computer science, vol. 4, 2011, pp. 716-725.

Lukowicz, P.; Heizn, E.A.; Prechelt, L. and W.F. Tichy (1994): « Experimental evaluation in computer science: a quantitative study », Journal of Systems and software 28 (1994), pp. 9-18.

http://page.mi.fu-berlin.de/prechelt/Biblio/1994-17.pdf

Luszczek, P.; Dongarra, J.J; Koester, D.; Rabenseifner, R.; Lucas, B.; Kepner, J.; McCalpin, J.; Bailey, D. and D. Takahashi (2006):« Introduction to the HPC challenge benchmark suite », SC'06 proceedings of the 2006 ACM/IEEE conference on Supercomputing, article No. 213.

http://icl.cs.utk.edu/projectsfiles/hpcc/pubs/hpcc-challenge-benchmark05.pdf

Malek, M. (1993): « High-performance computing in Europe », DIANE publishing company, 1993, pp. 471-517.

Marr, D.T.; Binns, F.; Hill, D.L.; Hinton, G.; Koufaty, D.A.; Miller, J.A. and M. Upton (2002): « Hyber-Threading technology architecture and microarchitecture », Intel technology journal, 6(1), 2002, pp. 4-15.

http://www.ece.cmu.edu/~ece742/f12/lib/exe/fetch.php?media=marr_hyperthread02.pdf

Mascagni, M and A. Srinivasan (2000): « Algorithm 806: SPRNG: A scalable library for pseudorandom number generation », ACM Transactions on Mathematical Software 26 (2000), pp. 436-461. http://www.cs.fsu.edu/~mascagni/papers/RIJP2000_2.pdf

Mascagni, M. and A. Srinivasan (2004): « Parameterizing parallel multiplicative lagged-Fibonacci generators », Parallel computing, 30(7), 2004, pp. 899-916.

http://www.sciencedirect.com/science/article/pii/S0167819104000663

Matsumoto, M. and T. Nishimura (1998): « Mersenne Twister: A 623-dimensionnally equidistributed uniform pseudorandom number generator ». ACM Transactions on Modeling and Computer Simulations: Special issue on Uniform Random Number Generation, 8(1), 1998, pp. 3-30.

Matsumoto, M. and T. Nishimura (2000): « Dynamic creation of pseudorandom number generators», Monte Carlo and Quasi-Monte Carlo Methods 1998, Springer, 2000, pp. 56-99.

http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/DC/dgene.pdf

Matsumoto, M.; Saito, M.; Haramoto, H. and T. Nishimura (2006): « Pseudo number generation: Impossibility and compromise », Journal of universal computer science, 12(6), 2006, pp. 672-690.

http://www.jucs.org/jucs_12_6/pseudorandom_number_generation_impossibility/jucs_12_06_0672_0690_matsumoto.pdf

Matsumoto, M.; Wada, I.; Kuramoto, A. and H. Ashihara (2007): « Common defects in initialization of pseudorandom number generators ». ACM Transactions on modeling and computer simulation (TOMACS) journal, 17(4), 2007, article no. 15.

Meng, H. and D. Thain (2015): « Umbrella: a portable environement creator for reproducible computing on clusters, clouds, and grids », VTDC'15 proceedings of the 8th internation workshop on virtualization technologies in distributed computing, 2015, pp. 23-30.

https://daspos.crc.nd.edu/images/reports/umbrella-vtdc15.pdf

Metropolis, N. (1987): « The beginning of the Monte Carlo method », Los Alamos science (1987 Special issue dedicated to Stanislaw Ultam), pp. 125-130.

Meuer, H.S. and H. Gietl (2013): « Supercomputers – Prestige objects or crucial tools for science and industry? », PIK – Praxis de Informationsverarbeitung und Kommunikation, 36 (2), 2013, pp. 117-128. http://www.top500.org/files/Supercomputers_London_Paper_HWM_HG.pdf

Naicken, S.; Livingston, B.; Basu, A.; Rodhetbhai, S.; Wakeman, I. and D. Chalmers (2007): « The state of peer-to-peer simulators and simulations ». ACM SIGCOMM computer communication review, 37(2), 2007, pp. 95-98.

http://www.sigcomm.org/sites/default/files/ccr/papers/2007/April/1232919-1232932.pdf

Neves, S. and F. Araujo (2012): « Fast and small nonlinear pseudorandom number generators for computer simulation », Parallel processing and applied mathematics, volume 7203 of Lecture notes in computer science, springer 2012, pp. 92-101.

https://eden.dei.uc.pt/~sneves/pubs/2011-snfa2.pdf

Passerat-Palmbach, J.; Mazel, C. and D.R.C. Hill (2012): « Pseudo-random streams for distributed and parallel stochastic simulations on GP-GPU », journal of simulation, 2012, pp. 141-151.

https://hal.archives-ouvertes.fr/hal-01099194/document

Passerat-Palmbach, J.; Mazel, C. and D.R.C. Hill (2014a): « TaskLocal-Random: a statistically sound substitute to pseudorandom number generation in parallel java tasks frameworks ». Concurrency and Computation: Practice and Experience, In press, Article first published online: 18 FEB 2014, DOI: 10.1002/cpe.3214.

Passerat-Palmbach, J. and D.R.C. Hill (2014b): « OpenCL: A suitable solution to simplify and unify high performance computing development », chapter 8 in F. Magoules (Editor), Patterns for parallel programming on GPUs, Saxe-Coburg publications, Stirlingshire, Scotland, 2014, pp. 189-209.

Pawlikowski, K.; Jeong, H.-D. and J.-S. Ruth Lee (2002): « On credibility of Simulation studies of telecommunication networks ». IEEE Communications Magazine, 40 (1), 2002, pp. 132–139.

http://www.cosc.canterbury.ac.nz/krys.pawlikowski/publications/credibility01.pdf

Pawlikowski, K.; Yau, V. and McNickle (1994): « Distributed stochastic discrete-event simulation in parallel time streams », Proccedings of the 26th conference on Winter simulation, pp. 723-730. Society for Computer Simulation International.

Press, W.H.; Teukolsky, S.A.; Vetterling, W.T. and B.P. Flannery (2007): « Numerical recipes in C: The art of scientific computing », 3rd edition, Cambridge university press, 2007.

Pryor, D.V; Cuccaro, S.A; Mascagni, M. et M.L. Robinson (1994): « Implementation of a portable and reproducible parallel pseudorandom number generator », proceedings of the 1994 ACM/IEEE conference on Supercomputing, 1994m pp. 311-319.

Raychaudhuri, S. (2008): « Introduction to Monte Carlo simulation », IEEE proceedings of the 2008 winter simulation conference, 2008, pp. 91-100.

http://www.informs-sim.org/wsc08papers/012.pdf

Reinders, J. (2015): « Your path to kights landing, the next generation of Intel Xeon Phi technology », the parallel universe at Intel corporation, issue 20, 2015, pp. 4-15.

Reuillon, R.; Hill, D.R.C, El Bitar, Z. and V. Breton (2008): « Rigorous distribution of stochastic simulations using the DistMe toolkit », IEEE Transactions on nuclear science, 55(1), 2008, pp. 595-603.

Reuillon, R. (2008): « Simulations stochastiques en environnements distribués: Application aux grille de calcul », Ph.D Thesis, Blaise Pascal University, 2008.

Revol, N. and P. Théveny (2014): « Numerical reproducibility and parallel computations: Issues for interval algorithms », IEEE Transactions on Computer, 63(8), 2014, pp. 1915-1924.

http://arxiv.org/pdf/1312.3300v1.pdf

Rønnow, T.F. ; Wang, Z. ; Job, J. ; Boixo, S. ; Isakov, S.V. ; Wecker, D. ; Martinis, J.M. ; Lidar, D.A. and M. Troyer (2014): « Defining and detecting quantum speedup », Science journal, 345(6195), 2014, pp. 420-424. http://arxiv.org/pdf/1401.2910v1.pdf

Saito, M. and M. Matsumoto (2013): « Variants of Mersenne Twister suitable for graphic processors », ACM Transaction on Mathematical software, 39(2), No. 12, 2013, pp. 1-20.

http://arxiv.org/pdf/1005.4973v3.pdf

Salmon, J. K.; Moraes, M. A.; Dror, R. O. and D. E. Shaw (2011): « Parallel random numbers: As easy as 1, 2, 3 », in High Performance Computing, Networking, Storage and Analysis (SC), 2011 International conference, pp. 1-12.

http://www.thesalmons.org/john/random123/papers/random123sc11.pdf

Schwab, M.; Karrenbach, M. and J. Claerbout (2000): « Making scientific computations reproducible ». Computing in Science & Engineering, 2(6), 2000, pp. 61-67.

http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.57.7555&rep=rep1&type=pdf

Schweitzer, P. (2015): « Simulations parallèles de Monte Carlo appliquées à la Physique des hautes énergies pour plates-formes manycore et multicore : mise au point, optimisation, reproductibilité», Ph.D Thesis, Blaise Pascal University, 2015.

Srinivasan, A.; Ceperley, D.M and M. Mascagni (1998):« Random number generators for parallel applications ». Advances in chemical physics, volume 15: Monte Carlo methods in chemical physics, 1998, pp.13-37. http://people.physics.illinois.edu/Ceperley/papers/121.pdf

Srinivasan, A.; Mascagni, M. and D. Ceperley (2003): « Testing parallel random number generators ». Parallel computing journal, 29(1), 2003, pp. 69-94.

http://www.cs.fsu.edu/~mascagni/papers/RIJP2003_4.pdf

Stodden, V. (2009a): « The reproducible research standard: Reducing legal barries to scientific knowledge and innovation », IEEE Computing in Science & Engineering, 11(1), 2009, pp. 35-40.

Stodden, V. (2009b): « The legal framework for reproducible scientific research: Licensing and copyright », Computing in Science & Engineering, 11(1), 35-40, 2009s.

http://web.stanford.edu/~vcs/papers/ReproducibleResearchLegal08172008.pdf

Stodden, V.; Borwein, J.M. and D.H. Bailey (2013): « Setting the default to reproducible: Reproduciblity in Computational and Experimental mathematics », SIAM News, 46(5), June 2013, pg. 4-6. http://www.davidhbailey.com/dhbpapers/icerm-report.pdf

Taufer M.; Padron, O.; Saponaro, P. and S. Patel (2010): « Improving Numerical Reproducibility and Stability in Large-Scale Numerical Simulations on GPUs », In IEEE Int'l Symp. Parallel and Distributed Processing, 2010, pp. 1–9.

http://www.gpucomputing.net/sites/default/files/papers/1681/010IPDPSmt.pdf

TRANSTECT report (2013): « High performance computing 2013/2014: Technology Compass », 2013, 186 pages.

Traore, M.K. and D.R.C. Hill (2003) : « Réflexion sur la Théorie de la Simulation. Le statut épistémologique de la simulation, Rencontre interdisciplinaires sur les systèmes complexes naturels et artificiels », 10ème journée de Rochebrune, Ecole Nationale Supérieure des Télécom Ed., 2003, pp. 279-297.

Umlauft, M. and P. Reichl (2007): « Experience with the ns-2 network simulator - explicitly setting seeds considered harmful ». Wireless telecommunications symposium, 2007, pp.1-5.

http://publik.tuwien.ac.at/files/pub-inf_4620.pdf

Urbatsch, T.J. and T.M. Evans (1999): « Reproducibility in parallel Monte Carlo codes », LA-UR-99-1826 {Tech, Memo, XTM-99-20(U)}, Los Alamos National Lab., 1999.

Vajda, A. (2011): « Chapter 2: Multi-core an many-core processor architectures », Programming Many-core chips book, 2011, pp. 9-43.

Vandewalle, P.; Kovacevic J. and M. Vetterli (2009): « Reproducible Research in Signal Processing - What, why, and how ». IEEE Signal Processing Magazine, 26 (3), 2009, pp. 37-47.

http://infoscience.epfl.ch/record/136640/files/VandewalleKV09.pdf

Villa, O.; Chavarria-Miranda, D.; Gurumoorthi, V.; Marquez, A. and S. Krishnamoorthy (2009): « Effects on floating-point non-associativity on numerical computations on massively multithreaded systems », in Proceeding of CUG 2009, USA, 2009, pp. 1-11.

http://cass-mt.pnnl.gov/docs/pubs/pnnleffects_of_floating-pointpaper.pdf

Williams, J.; Geogre, A.D.; Richardson, J.; Gosrani, K. and S. Suresh (2008): « Computational density of fixed and reconfigurable multi-core devices for application accleration », proceedings of Reconfigurable systems Summer Institute.

Zelkowitz, M.V. and D.R. Wallance (1998): « Experimental models for validating technology », Computer 31(5), 1998, pp. 23-31. http://www.cs.umd.edu/~mvz/pub/computer97.pdf

Web-références

Allalen, M. (2015): « MIC&GPU architecture », PRACE PATC: Intel MIC and GPU programming workshop, 2015. Date d'accès 2016.

https://www.lrz.de/services/compute/courses/x_lecturenotes/MIC_GPU_Workshop/MIC-AND-GPU-2015.pdf

Ashby, S.; Beckman, P.; Chen, J.; Colella, P.; Collins, B.; Crawford, D.; Dongarra, J.; Kothe, D.; Lusk, R.; Messina, P.; Mezzacappa, T.; Moin, P.; Norma, M.; Rosner, R.; Sarkar, V.; Siegel, A.; Streitz, F.; White, A. and M. Wright (2010): « The opportunities and challenges of Exascale computing », summary report of the Advanced scientific computing advisory committee. Date d'accès 2016.

http://science.energy.gov/~/media/ascr/ascac/pdf/reports/Exascale_subcommittee_report.pdf

Barbic, J. (2007): « Multi-core architectures ». Date d'accès 2016.

http://www.cs.cmu.edu/~fp/courses/15213-s07/lectures/27-multicore.pdf

Barney B. (2015): « Introduction to Parallel computing ». Date d'accès 2016.

https://computing.llnl.gov/tutorials/parallel_comp/

Collberg, C.; Proebsting, T.; Moraila, G.; Shankaran, A.; Shi, Z. and A.M. Warren (2014): « Measuring reproducibility in computer systems research », technical report, 2014. Date d'accès 2016.

http://reproducibility.cs.arizona.edu/v1/tr.pdf

Corden, M.J. and D. Kreitzer (2010): « Consistency of floating-point results using the Intel compiler or why doesn't my application always give the same answer? », 2010. Date d'accès 2016.

https://software.intel.com/sites/default/files/article/164389/fp-consistency-102511.pdf

Corden M. (2013a): « Differences in Floating-Point Arithmetic between Intel Xeon processors and the Intel Xeon Phi coprocessor », White paper, 2013. Date d'accès 2016.

https://software.intel.com/en-us/articles/differences-in-floating-point-arithmetic-between-intel-xeon-processors-and-the-intel-xeon

Corden M. (2013b): « Run-to-run reproducibility of floating-point calculations for applications on Intel Xeon Phi coprocessors (and Intel Xeon Processors) », Intel website, 2013. Date d'accès 2016.

https://software.intel.com/en-us/articles/run-to-run-reproducibility-of-floating-point-calculations-for-applications-on-intel-xeon

Deley, D.W. (1991): « Using the C or C++ rand() function, How computers generate random numbers », 1991. Date d'accès 2016. http://www.daviddeley.com/random/crandom.htm#R1

Demmel, J. and H.D. Nguyen (2013b): « ReproBLAS: Reproducibility BLAS », talk invited in workshop on: Obtaining Bitwise Reproducible Results: Perspectives and Latest Advances, SC'2013.

http://bebop.cs.berkeley.edu/reproblas/docs/talks/SC13.pdf

D-wave (2015): « The D-wave quantum computer », technology information, 2015. Date d'accès 2016. http://www.dwavesys.com/sites/default/files/D-Wave-brochure-Sept2015B.pdf

Gray, A.; Sjöström, A. and N. Ilieva-Litova (2013): « Best practice mini-guide accelerated clusters: Using General Purpose GPUs », 2013. Date d'accès 2016.

http://www.prace-ri.eu/IMG/pdf/Best-Practice-Guide-GPGPU.pdf

Hager, G. and G. Wellein (2008): « Concepts of high performance computing », 2008. Date d'accès 2016. https://www.rrze.fau.de/ausbildung/vorlesungen/coHPC.pdf

Heinrich J. (2004): « Detecting a bad random number generator », 2004. Date d'accès 2016.

http://www-cdf.fnal.gov/physics/statistics/notes/cdf6850_badrand.pdf

Hey, T. (2016): « Quantum computing: An introduction », 1999. Date d'accès 2016.

http://www.realtechsupport.org/UB/SR/quantumcomputing/Hey_IntroQuantumComputing_1999.pdf

Hsu, J. (2014b): « When will have an Exascale supercomputer? », in IEEE spectrum, 2014. Date d'accès 2016.

http://spectrum.ieee.org/computing/hardware/when-will-we-have-an-exascale-supercomputer

Intel ICC (2016): « Intel C++ compiler user and reference guides », document number: 304968-023US. Date d'accès 2016.

https://software.intel.com/sites/default/files/b6/30/main_cls_mac.pdf

Intel MKL (2014): « Intel Math Kernel Library: Reference manual, Revision: 072, MKL 11.2 ». Date d'accès 2016. https://software.intel.com/sites/default/files/managed/9d/c8/mklman.pdf

Intel SDG (2014):« Intel Xeon Phi Coprocessor system software developers guide », SKU# 328207-003EN, 2014. Date d'accès 2016.

http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-coprocessor-system-software-developers-guide.html

Jeong, B. and G. Abram (2016): « 8 things you should know about GPGPU technology: Q&A with TACC research scientits », date d'accès 2016.

https://www.tacc.utexas.edu/documents/13601/88790/8things.pdf

Jimenez, I.; Maltzahn, C.; Moody, A. and K. Mohror (2014): « Redo: Reproducibility at Scale », technical report – UCSC-SOE-14-12, 2014. https://www.soe.ucsc.edu/research/technical-reports/UCSC-SOE-14-12

Jones D. (2010): « Good practice in (pseudo) random number generation for bioinformatics applicatiosn », 2010. Date d'accès 2016.

http://www0.cs.ucl.ac.uk/staff/d.jones/GoodPracticeRNG.pdf

Kahan, W. and J.D. Darcy (1998): « How Java's floating-point hurts everyone everywhere », in the invitation of the ACM 1998, Workshop on Java for High-Performance Network Computing, Standford University. http://www.cs.berkeley.edu/~wkahan/JAVAhurt.pdf

Katzgraber, H.G. (2010): « Random numbers in scientific computing: an introduction », lecture given at the International summer school Modern computational science, 2010, Oldenburg, Germany. Date d'accès 2016. http://arxiv.org/pdf/1005.4117.pdf

L'Ecuyer, P.; Munger, D. and N. Kemerchou (2015b): « clRNG: A random number API with multiple streams for OpenCL », 2015. http://simul.iro.umontreal.ca/clrng/clrng-api.pdf

Modha, D.S. (2016): « Introducing a Brain-inspired computer : TrueNorth's neurons to revolutionize system architecture », IBM research article. Date d'accès 2016.

http://www.research.ibm.com/articles/brain-chip.shtml

NICS (2016) : « What is HPC ? », the National Institute for Computational Sciences. Date d'accès 2016

https://www.nics.tennessee.edu/computing-resources/what-is-hpc

Nussbaum, L. and O. Richard (2014): « Realis'2014: Reproductibilité expérimentale pour l'informatique en parallélisme, architecture et système ». ComPas'2014. Date d'accès 2016.

https://hal.inria.fr/hal-01011401/document

Nussbaum, L. (2015): « Reproducible research in computer science ». Séminaire du laboratoire ICube, 2015, Strabourg, France. Date d'accès 2016.

https://hal.inria.fr/hal-01110206/file/reproducible-research-20150127.pdf

PRACE Report (2012): « The scientific case for high performance computing in Europe 2012-2020: From Petascale to Exascale », 2012. Date d'accès 2016.

http://www.prace-ri.eu/IMG/pdf/prace_-_the_scientific_case_-_full_text_-.pdf

PRACE Reprot (2013): « Supercomputers for all: The next frontier for high performance computing ». In Science Business, 2013. Date d'accès 2016.

http://www.prace-ri.eu/IMG/pdf/prace_report_october_2013.pdf

Schuller, I.K. and R. Stevens (2015): « Neuromorphic computing: from materials to systems architecture », report of a roundtable convened to consider neuromorphic computing basic research needs, 2015.

http://science.energy.gov/~/media/ascr/pdf/programdocuments/docs/Neuromorphic-Computing-Report_FNLBLP.pdf

Stodden, V. (2010): « The scientific method in practice: reproducibility in the computational sciences ». MIT Sloan research paper No. 4773-10, 2010.

https://web.stanford.edu/~vcs/papers/SMPRCS2010.pdf

Taufer, M. (2012): « Numerical reproducibility and stability of large-scale simulations on exascale architecture ». Subimission 8 in Exascale operating systems and runtime software, Washinton, DC, USA, 2012.

https://collab.mcs.anl.gov/download/attachments/3801153/exaosr12_submission_8.pdf?version=2&modificationDate=1342818264000

Taufer, M.; Becchi, M.; Johnston, T. et D. Chapp (2015): « Numerical reproducibility challenges on extreme scale multi-threading GPUs », selected invited talk in NVIDIA GPU Technology conference, San Jose, CA, USA, 2015.

http://ondemand.gputechconf.com/gtc/2015/presentation/Michela-Taufer.pdf

Rougier, N. (2015): « ReScience, refaire la science », in workshop of 4R (Retour d'expéRiences sur la Recherche Reproductible à Orléans), 2015. Date d'accès 2016.

Thomas, F. (2012): « Optimisation of weather applications on Power and x86 architectures with a focus on reproducibility », in workshop of ECMWF, 2012. Date d'accès 2016.

http://www.ecmwf.int/sites/default/files/elibrary/2012/14026-optimisation-weather-applications-power-and-x86-architectures-focus-reproducibility.pdf

Whitehead, N. and A. Fit-Florea (2011): « Precision & performance: Floating point and IEEE 754 compliance for NVIDIA GPUs ». Technical report, NVIDIA corporation, 2011. Date d'accès 2016.

http://developer.download.nvidia.com/assets/cuda/files/NVIDIA-CUDA-Floating-Point.pdf

Annexe

Nous avons validé l'approche multivariée proposée par Ismay pour tester les corrélations des générateurs de nombres pseudo-aléatoires en parallèle dans cette thèse. Donc, nous présentons complétement nos résultats obtenus pour cette validation dans cette partie. De plus, nous présentons aussi tous les codes de l'outil RSPR et de la simulation stochastique de croissance démographique.

1 - Les résultats du test statistique avec l'approche de C.Ismay pour des bons générateurs de nombres pseudos-aléatoires

D'être plus facile à voir les résultats du test statistique avec l'approche de C.Ismay, nous avons reformulé avec les codes couleurs suivantes :

- en rouge, les p-values inférieures à 1e-4 ou supérieures à 1-(1e-4) (celles-ci étant marquées comme égales à 1). Ceci signifie que le test a détecté des corrélations.

- en orange, les p-values comprises entre 1e-4 et 0,01 ou bien entre 0,99 et 1-(1e-4). Ceci signifie qu'il y a probablement des corrélations, mais que cela demanderait confirmation en réessayant le test avec d'autres valeurs (mais on a autre chose à faire pour l'instant).

- en vert, les p-values comprises entre 0,01 et 0,99. Ceci signifie que le test n'a pas détecté de corrélations.

Leapfrog								
PRNG	**Type of matrix**	**mcorr**			**mmult**	**mport**		
		Pearson	**Spearman**	**Kendall**		**Li-McLeod**	**Hosking**	**Mahdi-McLeod**
COVEU64	16384x2	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: 2.45595	Li-McLeod: 6.61343	Hosking: 6.6133	Toeplitz: 0.999393 Mahdi-McLeod :

					p-value: 0.483305	DOF: 8 p-value: 0.578855	DOF: 8 p-value: 0.57887	5.96497 DOF: 7.2 p-value: 0.566288
	8192x4	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: 18.7929 p-value: 0.042974	Li-McLeod: 64.6538 DOF: 64 p-value: 0.453623	Hosking: 64.6533 DOF: 64 p-value: 0.45364	Toeplitz: 0.979798 Mahdi-McLeod: 55.7285 DOF: 53.3333 p-value: 0.384802
	4096x8	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: 56.0841 p-value: 0.017588	Li-McLeod: 483.258 DOF: 512 p-value: 0.814508	Hosking: 483.21 DOF: 512 p-value: 0.814927	Toeplitz: 0.575928 Mahdi-McLeod: 398.834 DOF: 406.588 p-value: 0.5988
	2048x16	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: 186.486	Li-McLeod: 4127.31	Hosking: 4127.54	Toeplitz: 1.53764e-08 Mahdi-

					p-value: 0.00265867	DOF: 4096 p-value: 0.362245	DOF: 4096 p-value: 0.361334	McLeod: 3349.49 DOF: 3165.09 p-value: 0.0112312
	1024x32	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: 675.261 p-value: 1.40718e-05	Li-McLeod: 32760.3 DOF: 32768 p-value: 0.510936	Hosking: 32758.8 DOF: 32768 p-value: 0.51335	Toeplitz: 8.05219e-480 Mahdi-McLeod: 52136.8 DOF: 24954.1 p-value: 0
	512x64	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: 2464.87 p-value: 8.35054e-09	Li-McLeod: 263368 DOF: 262144 p-value: 0.045575	Hosking: 263488 DOF: 262144 p-value: 0.0319098	Toeplitz: -0 Mahdi-McLeod: inf DOF: 198132 p-value: 0
	256x128	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: 10700.4	Li-McLeod: 2.10637e+06	Hosking: 2.10896e+06	Toeplitz: 0 Mahdi-McLeod : inf

					p-value: 0	DOF: 2097152 p-value: 3.50851e-06	DOF: 2097152 p-value: 4.32603e-09	DOF: 1.57898e+06 p-value: 0
MRG32k3a	16384x2	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: 1.17073 p-value: 0.760032	Li-McLeod: 9.99667 DOF: 8 p-value: 0.26526	Hosking: 9.99671 DOF: 8 p-value: 0.265257	Toeplitz: 0.99895 Mahdi-McLeod: 10.3304 DOF: 7.2 p-value: 0.184061
	8192x4	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: 5.41426 p-value: 0.861845	Li-McLeod: 57.1084 DOF: 64 p-value: 0.716669	Hosking: 57.1061 DOF: 64 p-value: 0.716742	Toeplitz: 0.982376 Mahdi-McLeod: 48.5535 DOF: 53.3333 p-value: 0.659942
	4096x8	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: 26.2555 p-value:	Li-McLeod: 471.481	Hosking: 471.405	Toeplitz: 0.572503 Mahdi-McLeod :

					0.883304	DOF: 512 p-value: 0.899778	DOF: 512 p-value: 0.900217	403.145 DOF: 406.588 p-value: 0.538909
	2048x16	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: 128.325 p-value: 0.667779	Li-McLeod: 3999.05 DOF: 4096 p-value: 0.858217	Hosking: 3998.67 DOF: 4096 p-value: 0.85918	Toeplitz: 3.24828e-08 Mahdi-McLeod: 3210.25 DOF: 3165.09 p-value: 0.283222
	1024x32	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: 573.773 p-value: 0.0822354	Li-McLeod: 32542.4 DOF: 32768 p-value: 0.810722	Hosking: 32538.1 DOF: 32768 p-value: 0.815267	Toeplitz: 1.36647e-475 Mahdi-McLeod : 51676.5 DOF: 24954.1 p-value: 0
	512x64	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: 2244.23	Li-McLeod: 261996	Hosking: 261998	Toeplitz: 0 Mahdi-McLeod:

					p-value: 0.00634321	DOF: 262144 p-value: 0.580601	DOF: 262144 p-value: 0.579713	inf DOF: 198132 p-value: 0
	256x128	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: 10219.7 p-value: 0	Li-McLeod: 2.10235e+06 DOF: 2097152 p-value: 0.00562554	Hosking: 2.10355e+06 DOF: 2097152 p-value: 0.000906553	Toeplitz: 0 Mahdi-McLeod : inf DOF: 1.57898e+06 p-value: 0
MT19937	16384x2	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: 0.291119 p-value: 0.96169	Li-McLeod: 4.00596 DOF: 8 p-value: 0.856585	Hosking: 4.00561 DOF: 8 p-value: 0.856617	Toeplitz: 0.999644 Mahdi-McLeod : 3.50204 DOF: 7.2 p-value: 0.849856
	8192x4	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: 7.8949 p-value:	Li-McLeod: 60.6247	Hosking: 60.6248	Toeplitz: 0.982654 Mahdi-McLeod:

					0.639102	DOF: 64 p-value: 0.596586	DOF: 64 p-value: 0.596582	47.7827 DOF: 53.3333 p-value: 0.688683
	4096x8	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: 31.744 p-value: 0.671247	Li-McLeod: 556.897 DOF: 512 p-value: 0.0830726	Hosking: 556.972 DOF: 512 p-value: 0.0827358	Toeplitz: 0.552091 Mahdi-McLeod: 429.388 DOF: 406.588 p-value: 0.209447
	2048x16	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: 116.235 p-value: 0.888962	Li-McLeod: 4114.06 DOF: 4096 p-value: 0.41817	Hosking: 4113.94 DOF: 4096 p-value: 0.41869	Toeplitz: 1.46847e-08 Mahdi-McLeod: 3358.06 DOF: 3165.09 p-value: 0.0085146
	1024x32	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: 526.938	Li-McLeod: 33184.4	Hosking: 33189.1	Toeplitz: 6.51903e-479

					p-value: 0.504857	DOF: 32768 p-value: 0.0523488	DOF: 32768 p-value: 0.0504601	Mahdi-McLeod: 52038 DOF: 24954.1 p-value: 0
	512x64	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: 2247.14 p-value: 0.00561836	Li-McLeod: 262014 DOF: 262144 p-value: 0.570837	Hosking: 262011 DOF: 262144 p-value: 0.572477	Toeplitz: -0 Mahdi-McLeod : inf DOF: 198132 p-value: 0
	256x128	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: 10545.7 p-value: 0	Li-McLeod: 2.10228e+06 DOF: 2097152 p-value: 0.0061755	Hosking: 2.10377e+06 DOF: 2097152 p-value: 0.000622476	Toeplitz: -0 Mahdi-McLeod: inf DOF: 1.57898e+06 p-value: 0
	16384x2	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: 2.66794	Li-McLeod: 8.65144	Hosking: 8.65152	Toeplitz: 0.999219

RAN2					p-value: 0.445704	DOF: 8 p-value: 0.372547	DOF: 8 p-value: 0.372541	Mahdi-McLeod : 7.67766 DOF: 7.2 p-value: 0.382482
	8192x4	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: 9.06519 p-value: 0.525929	Li-McLeod: 68.865 DOF: 64 p-value: 0.316263	Hosking: 68.8684 DOF: 64 p-value: 0.31616	Toeplitz: 0.981119 Mahdi-McLeod: 52.0516 DOF: 53.3333 p-value: 0.524073
	4096x8	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: 25.7515 p-value : 0.897165	Li-McLeod: 488.996 DOF: 512 p-value: 0.760845	Hosking: 488.958 DOF: 512 p-value: 0.761221	Toeplitz: 0.576732 Mahdi-McLeod : 397.825 DOF: 406.588 p-value: 0.612602
	2048x16	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: 133.736	Li-McLeod: 4012.55	Hosking: 4012.3	Toeplitz: 3.24761e-08

					p-value: 0.538849	DOF: 4096 p-value: 0.821464	DOF: 4096 p-value: 0.822206	Mahdi-McLeod: 3210.29 DOF: 3165.09 p-value: 0.283061
	1024x32	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: 577.36 p-value: 0.0675366	Li-McLeod: 32881.6 DOF: 32768 p-value: 0.327875	Hosking: 32885.9 DOF: 32768 p-value: 0.321869	Toeplitz: 3.59187e-479 Mahdi-McLeod : 52066.1 DOF: 24954.1 p-value: 0
	512x64	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: 2145.6 p-value: 0.154593	Li-McLeod: 262045 DOF: 262144 p-value: 0.553866	Hosking: 262056 DOF: 262144 p-value: 0.547775	Toeplitz: -0 Mahdi-McLeod: inf DOF: 198132 p-value: 0
	256x128	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: 9955.65	Li-McLeod: 2.10143e+06	Hosking: 2.10318e+06	Toeplitz: -0 Mahdi-

					p-value: 0	DOF: 2097152 p-value: 0.0184104	DOF: 2097152 p-value: 0.00163542	McLeod: inf DOF: 1.57898e+06 p-value: 0
MLFG	16384x2	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: 4.66191 p-value: 0.198295	Li-McLeod: 7.38164 DOF: 8 p-value: 0.496071	Hosking: 7.38172 DOF: 8 p-value: 0.496063	Toeplitz: 0.999458 Mahdi-McLeod: 5.33214 DOF: 7.2 p-value: 0.641325
	8192x4	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: 9.61292 p-value: 0.475084	Li-McLeod: 59.9345 DOF: 64 p-value: 0.620942	Hosking: 59.9315 DOF: 64 p-value: 0.621048	Toeplitz: 0.980151 Mahdi-McLeod: 54.746 DOF: 53.3333 p-value: 0.420718
	4096x8	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: 33.8034	Li-McLeod: 479.324	Hosking: 479.258	Toeplitz: 0.569674 Mahdi-

					p-value: 0.573487	DOF: 512 p-value: 0.846785	DOF: 512 p-value: 0.847298	McLeod: 406.727 DOF: 406.588 p-value: 0.488739
	2048x16	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: 151.793 p-value: 0.167801	Li-McLeod: 3954.52 DOF: 4096 p-value: 0.942268	Hosking: 3953.95 DOF: 4096 p-value: 0.943019	Toeplitz: 3.16157e-08 Mahdi-McLeod : 3215.29 DOF: 3165.09 p-value: 0.262393
	1024x32	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: 508.741 p-value: 0.71885	Li-McLeod: 32382.7 DOF: 32768 p-value: 0.934294	Hosking: 32375.9 DOF: 32768 p-value: 0.937654	Toeplitz: 1.57719e-476 Mahdi-McLeod: 51778.6 DOF: 24954.1 p-value: 0
	512x64	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: 2128 p-value:	Li-McLeod: 262284	Hosking: 262296	Toeplitz: 0 Mahdi-

					0.22698	DOF: 262144 p-value: 0.423205	DOF: 262144 p-value: 0.41653	McLeod: inf DOF: 198132 p-value: 0
	256x128				LR: 9825.92 p-value: 0	Li-McLeod: 2.10275e+06 DOF: 2097152 p-value: 0.00314791	Hosking: 2.10423e+06 DOF: 2097152 p-value: 0.000278016	Toeplitz: 0 Mahdi-McLeod: inf DOF: 1.57898e+06 p-value: 0

Table A. 1. Les résultats du test statistique des générateurs de nombres pseudo-aléatoires COVEYOU64, MRG32k3a, MT19937, MLFG, ran2 avec deux techniques de parallélisation « coupé par bloc » (sequence splitting en anglais) et « tourniquet » (roundrobin en anglais).

Avec les termes :

- LR = Likelihood Ratio test
- Li-McLeod = Portmanteau test for white noise
- DOF = degrees of freedom
- Toeplitz = Determinant of Toeplitz Block matrix

2 - Les résultats du test statistique avec l'approche de C.Ismay pour des mauvaises générateurs de nombres pseudos-aléatoires

Leapfrog								
PRNG	**Type of matrix**	**mcorr**			**mmult**	**mport**		
		Pearson	**Spearman**	**Kendall**		**Li-McLeod**	**Hosking**	**Mahdi-McLeod**
BSD	16384x2	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: -nan p-value: 0	Li-McLeod: 11.1364 DOF : 8 p-value: 0.194088	Hosking: 11.1367 DOF : 8 p-value: 0.194074	Toeplitz: 0.998933 Mahdi-McLeod : 10.4913 DOF: 7.2 p-value: 0.1754
	8192x4	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: -nan p-value: 0	Li-McLeod : 64.9742 DOF: 64 p-value : 0.442525	Hosking: 64.9724 DOF: 64 p-value : 0.442587	Toeplitz Block: 0.978056 Mahdi-McLeod: 60.5893 DOF: 53.3333 p-value: 0.230587
	4096x8	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: -nan p-value: 0	Li-McLeod: 520.024	Hosking: 520.026 DOF: 512	Toeplitz Block: 0.559942 Mahdi-McLeod : 419.181 DOF: 406.588

						DOF: 512 p-value: 0.393482	p-value: 0.393454	p-value: 0.322594
	2048x16	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: -nan p-value: 0	Li-McLeod: 4019.46 DOF: 4096 p-value: 0.800543	Hosking: 4019.1 DOF: 4096 p-value: 0.801687	Toeplitz Block: 2.37846e-08 Mahdi-McLeod: 3268.28 DOF: 3165.09 p-value: 0.098271
	1024x32	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: -nan p-value: 0	Li-McLeod: 33130.7 DOF: 32768 p-value: 0.0786252	Hosking: 33138.1 DOF: 32768 p-value: 0.0745342	Toeplitz Block: 7.17253e-478 Mahdi-McLeod: 51924.6 DOF: 24954.1 p-value: 0
	512x64	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: -nan p-value: 0	Li-McLeod: 262736 DOF: 262144 p-value: 0.206859	Hosking: 262787 DOF: 262144 p-value: 0.187262	Toeplitz Block: -0 Mahdi-McLeod: inf DOF: 198132 p-value: 0
	256x128	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: -nan p-value: 0	Li-McLeod: 2.10637e+06 DOF:	Hosking: 2.10948e+06 DOF: 2097152	Toeplitz Block: 0 Mahdi-McLeod: inf

						2097152 p-value: 3.45605e-06	p-value: 9.32103e-10	DOF: 1.57898e+06 p-value: 0
JAVA	16384x2	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: 16.6679 p-value: 0.000827037	Li-McLeod: 7.10233 DOF: 8 p-value: 0.525633	Hosking: 7.10234 DOF: 8 p-value: 0.525632	Toeplitz Block: 0.999443 Mahdi-McLeod: 5.47548 DOF: 7.2 p-value: 0.624184
	8192x4	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: 20.8441 p-value: 0.0222071	Li-McLeod: 78.3885 DOF: 64 p-value: 0.106523	Hosking: 78.3929 DOF: 64 p-value: 0.10646	Toeplitz Block: 0.976275 Mahdi-McLeod: 65.5659 DOF: 53.3333 p-value: 0.121447
	4096x8	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: 34.1701 p-value: 0.555834	Li-McLeod: 508.683 DOF: 512 p-value:	Hosking: 508.673 DOF: 512 p-value:	Toeplitz Block: 0.565536 Mahdi-McLeod:

						0.533086	0.533207	411.996 DOF: 406.588 p-value: 0.415987
	2048x16	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: 119.665 p-value: 0.839478	Li-McLeod: 4070.71 DOF: 4096 p-value: 0.607422	Hosking: 4070.57 DOF: 4096 p-value: 0.608031	Toeplitz Block: 1.92925e-08 Mahdi-McLeod : 3307.25 DOF: 3165.09 p-value: 0.0384437
	1024x32	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: 509.132 p-value: 0.714681	Li-McLeod: 32827.4 DOF: 32768 p-value: 0.407343	Hosking: 32827.1 DOF: 32768 p-value: 0.407828	Toeplitz Block: 2.93893e-478 Mahdi-McLeod: 51966.8 DOF: 24954.1 p-value: 0
	512x64	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: 2258.02 p-value: 0.00351652	Li-McLeod: 260919 DOF: 262144	Hosking: 260800 DOF: 262144	Toeplitz Block: 0 Mahdi-McLeod :

						p-value: 0.954765	p-value: 0.968473	inf DOF: 198132 p-value: 0
	256x128	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: 10428.8 p-value: 0	Li-McLeod: 2.09589e+06 DOF: 2097152 p-value: 0.731102	Hosking: 2.09518e+06 DOF: 2097152 p-value: 0.832272	Toeplitz Block: -0 Mahdi-McLeod: inf DOF: 1.57898e+06 p-value: 0
RANDU	16384x2	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: 2.33376 p-value: 0.506085	Li-McLeod: 7.02205 DOF: 8 p-value: 0.534256	Hosking: 7.02191 DOF: 8 p-value: 0.53427	Toeplitz Block: 0.999315 Mahdi-McLeod: 6.73833 DOF: 7.2 p-value: 0.478764
	8192x4	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: 11.1927 p-value: 0.342705	Li-McLeod: 63.4825 DOF: 64 p-value:	Hosking: 63.4829 DOF: 64 p-value:	Toeplitz Block: 0.981346 Mahdi-McLeod:

						0.494755	0.494743	51.4196 DOF: 53.3333 p-value: 0.548852
	4096x8	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: 42.6685 p-value: 0.206239	Li-McLeod: 569.256 DOF: 512 p-value: 0.0402988	Hosking: 569.31 DOF: 512 p-value: 0.0401605	Toeplitz Block : 0.529408 Mahdi-McLeod: 459.713 DOF: 406.588 p-value: 0.0350992
	2048x16	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: 140.407 p-value: 0.380302	Li-McLeod: 4194.12 DOF: 4096 p-value: 0.139423	Hosking : 4194.35 DOF: 4096 p-value: 0.138862	Toeplitz: 9.96488e-09 Mahdi-McLeod: 3430.25 DOF: 3165.09 p-value: 0.000572694
	1024x32	Reject none of the null	Reject none of the null hypotheses	Reject none of the null	LR: 523.059	Li-McLeod:	Hosking:	Toeplitz Block:

		hypotheses		hypotheses	p-value: 0.552507	33239.8 DOF: 32768 p-value: 0.0331122	33248.3 DOF: 32768 p-value: 0.0307679	5.39726e-484 Mahdi-McLeod: 52591 DOF: 24954.1 p-value: 0
	512x64	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: 2148.89 p-value: 0.143018	Li-McLeod: 262748 DOF: 262144 p-value: 0.20193	Hosking: 262746 DOF: 262144 p-value: 0.202647	Toeplitz Block: -0 Mahdi-McLeod: inf DOF: 198132 p-value: 0
	256x128	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: 9705.03 p-value: 0	Li-McLeod: 2.10074e+06 DOF: 2097152 p-value: 0.0398192	Hosking: 2.10164e+06 DOF: 2097152 p-value: 0.0141749	Toeplitz Block: -0 Mahdi-McLeod: inf DOF: 1.57898e+06 p-value: 0
	16384x2	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: -nan	Li-McLeod: 3.42314	Hosking: 3.42277	Toeplitz Block:

MARSAGLIA					p-value: 0	DOF: 8 p-value: 0.905072	DOF: 8 p-value: 0.9051	0.999738 Mahdi-McLeod: 2.57802 DOF: 7.2 p-value: 0.930161
	8192x4	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: -nan p-value: 0	Li-McLeod: 39.7918 DOF: 64 p-value: 0.992453	Hosking: 39.7843 DOF: 64 p-value: 0.992472	Toeplitz Block: 0.98782 Mahdi-McLeod: 33.4635 DOF: 53.3333 p-value: 0.984873
	4096x8	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: -nan p-value: 0	Li-McLeod: 526.629 DOF: 512 p-value: 0.317868	Hosking: 526.661 DOF: 512 p-value: 0.317525	Toeplitz Block: 0.567469 Mahdi-McLeod: 409.529 DOF: 406.588 p-value: 0.449767

	2048x16	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: -nan p-value: 0	Li-McLeod: 4128.68 DOF: 4096 p-value: 0.356636	Hosking: 4128.63 DOF: 4096 p-value: 0.356835	Toeplitz Block: 1.50209e-08 Mahdi-McLeod: 3353.85 DOF: 3165.09 p-value: 0.00976813
	1024x32	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: -nan p-value: 0	Li-McLeod: 32962.4 DOF: 32768 p-value: 0.22343	Hosking: 32966.8 DOF: 32768 p-value: 0.218393	Toeplitz Block: 2.56029e-483 Mahdi-McLeod: 52517.4 DOF: 24954.1 p-value: 0
	512x64	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: -nan p-value: 0	Li-McLeod: 263464 DOF: 262144 p-value: 0.0342606	Hosking: 263547 DOF: 262144 p-value: 0.0264462	Toeplitz Block: 0 Mahdi-McLeod: inf DOF:

								198132 p-value: 0
	256x128	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: -nan p-value: 0	Li-McLeod: 2.10115e+06 DOF: 2097152 p-value: 0.0256236	Hosking: 2.10247e+06 DOF: 2097152 p-value: 0.00473909	Toeplitz Block: -0 Mahdi-McLeod: inf DOF: 1.57898e+06 p-value: 0
FISHMAN	16384x2	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: -nan p-value: 0	Li-McLeod: 4.32135 DOF: 8 p-value: 0.827029	Hosking: 4.32099 DOF: 8 p-value: 0.827064	Toeplitz: 0.99957 Mahdi-McLeod: 4.2305 DOF: 7.2 p-value: 0.771417
	8192x4	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: -nan p-value: 0	Li-McLeod : 57.1223 DOF: 64 p-value: 0.71622	Hosking : 57.1212 DOF: 64 p-value: 0.716255	Toeplitz: 0.983785 Mahdi-McLeod: 44.6419 DOF: 53.3333 p-value: 0.795779
	4096x8	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: -nan p-value: 0	Li-McLeod: 566.504	Hosking: 566.574 DOF: 512	Toeplitz: 0.542037 Mahdi-McLeod: 442.673 DOF: 406.588

					DOF: 512 p-value: 0.0477599	p-value: 0.0475574	p-value: 0.105101
	2048x16	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: -nan p-value: 0	Li-McLeod: 4244.5 DOF: 4096 p-value: 0.0516963	Hosking: 4244.68 DOF: 4096 p-value: 0.0514851	Toeplitz: 7.57884e-09 Mahdi-McLeod: 3481.21 DOF: 3165.09 p-value: 5.75803e-05
	1024x32	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: -nan p-value: 0	Li-McLeod: 32762 DOF: 32768 p-value: 0.508306	Hosking: 32761.1 DOF: 32768 p-value: 0.509666	Toeplitz: 4.95328e-480 Mahdi-McLeod: 52159.8 DOF: 24954.1 p-value: 0
	512x64	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: -nan p-value: 0	Li-McLeod: 261594 DOF: 262144 p-value: 0.776086	Hosking: 261516 DOF: 262144 p-value: 0.806993	Toeplitz: -0 Mahdi-McLeod: inf DOF: 198132 p-value: 0
	256x128	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: -nan p-value: 0	Li-McLeod: 2.09814e+06 DOF: 2097152 p-value: 0.315165	Hosking: 2.09802e+06 DOF: 2097152 p-value: 0.335713	Toeplitz: -0 Mahdi-McLeod: inf DOF: 1.57898e+06 p-value: 0

MICROSOFT VISUAL BASIC 6.0	16384x2	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: 2.09473 p-value: 0.552981	Li-McLeod: 11.1419 DOF: 8 p-value: 0.193789	Hosking: 11.1424 DOF: 8 p-value: 0.193759	Toeplitz: 0.999249 Mahdi-McLeod: 7.38792 DOF: 7.2 p-value: 0.410824
	8192x4	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: 3.63861 p-value: 0.962181	Li-McLeod: 62.9611 DOF: 64 p-value: 0.513275	Hosking: 62.959 DOF: 64 p-value: 0.513349	Toeplitz: 0.979244 Mahdi-McLeod: 57.274 DOF: 53.3333 p-value: 0.331142
	4096x8	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: 28.8964 p-value: 0.793765	Li-McLeod: 481.451 DOF: 512 p-value: 0.829807	Hosking: 481.417 DOF: 512 p-value: 0.830083	Toeplitz: 0.587441 Mahdi-McLeod: 384.527 DOF: 406.588 p-value: 0.77774
	2048x16	Reject none of the	Reject none of the null	Reject none of the	LR: 132.23	Li-McLeod:	Hosking:	Toeplitz:

		null hypotheses	hypotheses	null hypotheses	p-value: 0.575415	3890.21 DOF: 4096 p-value: 0.98943	3889.37 DOF: 4096 p-value: 0.989698	3.67188e-08 Mahdi-McLeod: 3187.43 DOF: 3165.09 p-value: 0.386492
	1024x32	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: 531.156 p-value: 0.453264	Li-McLeod: 32333.5 DOF: 32768 p-value: 0.955634	Hosking: 32328.1 DOF: 32768 p-value: 0.957584	GV Test Statistic is infinite. Program exiting.
	512x64	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: 2180.62 p-value: 0.0610681	Li-McLeod: 261168 DOF: 262144 p-value: 0.911266	Hosking: 261129 DOF: 262144 p-value: 0.919626	Toeplitz: 0 Mahdi-McLeod: inf DOF: 198132 p-value: 0
	256x128	Reject none of the null hypotheses	Reject none of the null hypotheses	P-value_(1): Vector 46 and Vector	LR: 10058.7 p-value: 0	Li-McLeod: 2.10155e+06 DOF:	Hosking: 2.1034e+06 DOF:	Toeplitz: -0 Mahdi-McLeod:

				110 P-value_(2): Vector 33 and Vector 57		2097152 p-value: 0.0158563	2097152 p-value\: 0.00114391	inf DOF: 1.57898e+06 p-value: 0
COVE	16384x2	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: -nan p-value: 0	Li-McLeod: 6.06581 DOF: 8 p-value: 0.639861	Hosking: 6.06569 DOF: 8 p-value: 0.639874	Toeplitz: 0.999508 Mahdi-McLeod: 4.8402 DOF: 7.2 p-value: 0.700174
	8192x4	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: -nan p-value: 0	Li-McLeod: 60.4347 DOF: 64 p-value: 0.603315	Hosking: 60.4347 DOF: 64 p-value: 0.603315	Toeplitz: 0.982897 Mahdi-McLeod: 47.1067 DOF: 53.3333 p-value: 0.713227
	4096x8	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: -nan p-value: 0	Li-McLeod: 460.871 DOF: 512 p-value:	Hosking: 460.828 DOF: 512 p-value:	Toeplitz: 0.611976 Mahdi-McLeod: 354.952

						0.948796	0.948946	DOF: 406.588 p-value: 0.96926
	2048x16	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: -nan p-value: 0	Li-McLeod: 4270.25 DOF: 4096 p-value: 0.0283297	Hosking: 4271.41 DOF: 4096 p-value: 0.0275347	Toeplitz: 1.50467e-08 Mahdi-McLeod: 3353.53 DOF: 3165.09 p-value: 0.00986936
	1024x32	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: -nan p-value: 0	Li-McLeod: 32837.8 DOF: 32768 p-value: 0.391677	Hosking: 32835.1 DOF: 32768 p-value: 0.395724	Toeplitz: 4.79342e-478 Mahdi-McLeod: 51943.7 DOF: 24954.1 p-value: 0
	512x64	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: -nan p-value: 0	Li-McLeod: 262407 DOF: 262144 p-value: 0.358027	Hosking: 262404 DOF: 262144 p-value: 0.359689	Toeplitz: 0 Mahdi-McLeod: inf DOF:

								198132 p-value: 0
	256x128	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: -nan p-value: 0	Li-McLeod: 2.10014e+06 DOF: 2097152 p-value: 0.0720105	Hosking: 2.10101e+06 DOF: 2097152 p-value: 0.0299756	Toeplitz: 0 Mahdi-McLeod: inf DOF: 1.57898e+06 p-value: 0
Sequence splitting								
BSD	16384x2	P-value_(1): Vector 1 and Vector 2	P-value_(1): Vector 1 and Vector 2	P-value_(1): Vector 1 and Vector 2	LR: -nan p-value: 0			
	8192x4	P-value_(1): Vector 2 and Vector 4	P-value_(1): Vector 2 and Vector 4	P-value_(1): Vector 2 and Vector 4	LR: -nan p-value: 0	Li-McLeod : -4.32812 DOF: 64 p-value: 1	Hosking: -4.34926 DOF: 64 p-value: 1	GV Test Statistic is infinite
	4096x8	3 couples of correlation vectors	3 couples of correlation vectors	4 couples of correlation vectors	LR: nan p-value: 0	Li-McLeod: 4.33526e+22 DOF: 512 p-value: 0	Hosking: 4.3412e+22 DOF: 512 p-value: 0	GV Test Statistic is infinite

	2048x16	3 couples of correlation vectors	3 couples of correlation vectors	8 couples of correlation vectors	LR: -nan p-value: 0	Li-McLeod: 4.89544e+18 DOF: 4096 p-value: 0	Hosking: 4.93074e+18 DOF: 4096 p-value: 0	GV Test Statistic is infinite
	1024x32	6 couples of correlation vectors	6 couples of correlation vectors	16 couples of correlation vectors	LR: -nan p-value: 0	Li-McLeod: -1.75295e+20 DOF: 32768 p-value: 1	Hosking: -2.0186e+20 DOF: 32768 p-value: 1	GV Test Statistic is infinite
	512x64	17 couples of correlation vectors	17 couples of correlation vectors	32 couples of correlation vectors	LR: -nan p-value: 0	Li-McLeod: 8.61188e+19 DOF: 262144 p-value: 0	Hosking: 9.13577e+19 DOF: 262144 p-value: 0	GV Test Statistic is infinite
	256x128	33 couples of correlation vectors	33 couples of correlation vectors	64 couples of correlation vectors	LR: -nan p-value: 0	Li-McLeod: -4.37385e+18 DOF: 2097152 p-value: 1	Hosking: 3.9887e+19 DOF: 2097152 p-value: 0	GV Test Statistic is infinite
	16384x2	Reject none of	Reject none of	Reject none of	LR: 1795.24	Li-McLeod: 5.62258	Hosking: 5.62238	Toeplitz: 0.9995

JAVA		the null hypothes es	the null hypothese s	the null hypothese s	p-value: 0	DOF: 8 p-value: 0.689424	DOF: 8 p-value: 0.689447	Mahdi-McLeod: 4.91383 DOF: 7.2 p-value: 0.691398
	8192x4	Reject none of the null hypothes es	Reject none of the null hypothese s	4 couples of correlation vectors	LR : 3001 p-value: 0	Li-McLeod: 62.8019 DOF: 64 p-value: 0.518946	Hosking: 62.8009 DOF: 64 p-value: 0.518979	Toeplitz: 0.9805 Mahdi-McLeod: 53.774 DOF: 53.3333 p-value: 0.45735
	4096x8	28 couples of correlatio n vectors	28 couples of correlation vectors	24 couples of correlation vectors	LR: 6014.3 p-value: 0	Li-McLeod: 481.1 DOF: 512 p-value: 0.832684	Hosking: 481.069 DOF: 512 p-value: 0.832935	Toeplitz: 0.588886 Mahdi-McLeod: 382.751 DOF: 406.588 p-value: 0.79654
	2048x16	106 couples of correlatio	106 couples of correlation	77 couples of correlation	LR: 8025.37 p-value:	Li-McLeod: 3968.59 DOF: 4096	Hosking: 3967.82 DOF: 4096	Toeplitz: 2.61719e-08

		n vectors	vectors	vectors	0	p-value: 0.92149	p-value: 0.92276	Mahdi-McLeod: 3250.47 DOF: 3165.09 p-value: 0.141865
	1024x32	143 couples of correlation vectors	133 couples of correlation vectors	136 couples of correlation vectors	LR : 9415.25 p-value: 0	Li-McLeod: 33246.2 DOF: 32768 p-value: 0.0313263	Hosking: 33252.9 DOF: 32768 p-value: 0.0295599	GV Test Statistic is infinite
	512x64	178 couples of correlation vectors	173 couples of correlation vectors	236 couples of correlation vectors	LR: 11767.7 p-value: 0	Li-McLeod: 262796 DOF: 262144 p-value: 0.183803	Hosking: 262893 DOF: 262144 p-value: 0.150585	Toeplitz: 0 Mahdi-McLeod: inf DOF: 198132 p-value: 0
	256x128	303 couples of correlation vectors	300 couples of correlation vectors	301 couples of correlation vectors	LR: 20368.8 p-value: 0	Li-McLeod: 2.10051e+06 DOF: 2097152 p-value:	Hosking: 2.10166e+06 DOF: 2097152 p-value:	Toeplitz: 0 Mahdi-McLeod: inf DOF:

						0.0506944	0.0138208	1.57898e+06 p-value: 0
RANDU	16384x2	Reject none of the null hypothes es	Reject none of the null hypothese s	Reject none of the null hypothese s	LR: 1838.22 p-value: 0	Li-McLeod: 7.44275 DOF: 8 p-value: 0.489701	Hosking: 7.44258 DOF: 8 p-value: 0.489719	Toeplitz: 0.999198 Mahdi-McLeod: 7.88925 DOF: 7.2 p-value: 0.362599
	8192x4	Reject none of the null hypothes es	Reject none of the null hypothese s	Reject none of the null hypothese s	LR: 2946.86 p-value: 0	Li-McLeod: 60.396 DOF: 64 p-value: 0.604682	Hosking: 60.3954 DOF: 64 p-value: 0.604704	Toeplitz: 0.982232 Mahdi-McLeod: 48.9555 DOF: 53.3333 p-value: 0.644685
	4096x8	13 couples of correlatio n vectors	12 couples of correlation vectors	16 couples of correlation vectors	LR: 3673.86 p-value: 0	Li-McLeod: 494.942 DOF: 512 p-value: 0.697842	Hosking: 494.941 DOF: 512 p-value: 0.697859	Toeplitz: 0.588769 Mahdi-McLeod: 382.895 DOF: 406.588

								p-value: 0.795058
	2048x16	24 couples of correlation vectors	24 couples of correlation vectors	36 couples of correlation vectors	LR: 4120.17 p-value: 0	Li-McLeod: 4016.8 DOF: 4096 p-value: 0.80876	Hosking : 4016.45 DOF: 4096 p-value: 0.809825	Toeplitz: 3.13829e-08 Mahdi-McLeod: 3216.66 DOF: 3165.09 p-value: 0.256848
	1024x32	37 couples of correlation vectors	35 couples of correlation vectors	63 couples of correlation vectors	LR: 4601.74 p-value: 0	Li-McLeod: 32857.8 DOF: 32768 p-value: 0.361982	Hosking: 32862.1 DOF: 32768 p-value: 0.355803	GV Test Statistic is infinite
	512x64	37 couples of correlation vectors	35 couples of correlation vectors	60 couples of correlation vectors	LR: 6690.11 p-value: 0	Li-McLeod: 263744 DOF: 262144 p-value: 0.0136818	Hosking: 263820 DOF: 262144 p-value: 0.0104262	Toeplitz: 0 Mahdi-McLeod: inf DOF: 198132 p-value: 0
	256x128	65	64 couples	18 couples	LR:	Li-McLeod: 2.09339e+06	Hosking: 2.09035e+0	Toeplitz: -0

		couples of correlation vectors	of correlation vectors	of correlation vectors	14329.7 p-value: 0	DOF: 2097152 p-value: 0.966992	6 DOF: 2097152 p-value: 0.999558	Mahdi-McLeod: inf DOF: 1.57898e+06 p-value: 0
MAR	16384x2	Reject none of the null hypothes es	Reject none of the null hypothese s	Reject none of the null hypothese s	LR: -nan p-value: 0	Li-McLeod: 6.84446 DOF: 8 p-value: 0.553504	Hosking: 6.8442 DOF: 8 p-value: 0.553532	Toeplitz: 0.999221 Mahdi-McLeod: 7.66103 DOF: 7.2 p-value: 0.384074
	8192x4	Reject none of the null hypothes es	Reject none of the null hypothese s	4 couples of correlation vectors	LR: -nan p-value: 0	Li-McLeod: 58.8439 DOF: 64 p-value: 0.65881	Hosking: 58.8413 DOF: 64 p-value: 0.658899	Toeplitz: 0.981148 Mahdi-McLeod: 51.9695 DOF: 53.3333 p-value: 0.527287
	4096x8	15 couples of correlation vectors	13 couples of correlation vectors	24 couples of correlation vectors	LR: -nan p-value: 0	Li-McLeod: 541.388 DOF: 512 p-value: 0.178235	Hosking: 541.39 DOF: 512 p-value: 0.178224	Toeplitz: 0.531842 Mahdi-McLeod: 456.397 DOF: 406.588

								p-value: 0.0442649
	2048x16	25 couples of correlation vectors	25 couples of correlation vectors	40 couples of correlation vectors	LR: -nan p-value: 0	Li-McLeod: 4156.62 DOF: 4096 p-value: 0.250203	Hosking: 4156.55 DOF: 4096 p-value: 0.250448	Toeplitz: 1.21032e-08 Mahdi-McLeod: 3394.06 DOF: 3165.09 p-value: 0.0024053
	1024x32	35 couples of correlation vectors	34 couples of correlation vectors	48 couples of correlation vectors	LR: -nan p-value: 0	Li-McLeod: 32618.1 DOF: 32768 p-value: 0.720316	Hosking: 32620.5 DOF: 32768 p-value: 0.717121	GV Test Statistic is infinite
	512x64	38 couples of correlation vectors	36 couples of correlation vectors	66 couples of correlation vectors	LR: -nan p-value: 0	Li-McLeod: 261538 DOF: 262144 p-value: 0.798517	Hosking: 261430 DOF: 262144 p-value: 0.837913	Toeplitz: -0 Mahdi-McLeod: inf DOF: 198132 p-value: 0

	256x128	65 couples of correlation vectors	64 couples of correlation vectors	118 couples of correlation vectors	LR: -nan p-value: 0	Li-McLeod: 2.09939e+06 DOF: 2097152 p-value: 0.137416	Hosking: 2.09984e+06 DOF: 2097152 p-value: 0.0944971	Toeplitz: 0 Mahdi-McLeod: inf DOF: 1.57898e+06 p-value: 0
FISH	16384x2	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: -nan p-value: 0	Li-McLeod: 17.772 DOF: 8 p-value: 0.023002	Hosking: 17.7731 DOF: 8 p-value: 0.0229934	Toeplitz: 0.998543 Mahdi-McLeod: 14.3335 DOF: 7.2 p-value: 0.0504195
	8192x4	Reject none of the null hypotheses	Reject none of the null hypotheses	4 couples of correlation vectors	LR: -nan p-value: 0	Li-McLeod: 59.334 DOF: 64 p-value: 0.641905	Hosking: 59.3331 DOF: 64 p-value: 0.641938	Toeplitz: 0.982438 Mahdi-McLeod: 48.3821 DOF: 53.3333 p-value: 0.666393
	4096x8	14 couples of correlation vectors	13 couples of correlation vectors	24 couples of correlation vectors	LR: -nan p-value: 0	Li-McLeod: 500.106 DOF: 512 p-value: 0.638225	Hosking: 500.072 DOF: 512 p-value: 0.638634	Toeplitz: 0.564151 Mahdi-McLeod: 413.769 DOF:

								406.588 p-value: 0.392165
	2048x16	24 couples of correlation vectors	23 couples of correlation vectors	40 couples of correlation vectors	LR: -nan p-value: 0	Li-McLeod: 4076.74 DOF: 4096 p-value: 0.581531	Hosking: 4076.61 DOF: 4096 p-value: 0.582062	Toeplitz: 1.8035e-08 Mahdi-McLeod: 3319.8 DOF: 3165.09 p-value: 0.0273018
	1024x32	36 couples of correlation vectors	33 couples of correlation vectors	43 couples of correlation vectors	LR: -nan p-value: 0	Li-McLeod: 32910.9 DOF: 32768 p-value: 0.287686	Hosking: 32917 DOF: 32768 p-value: 0.279634	GV Test Statistic is infinite
	512x64	34 couples of correlation vectors	33 couples of correlation vectors	66 couples of correlation vectors	LR: -nan p-value: 0	Li-McLeod: 261448 DOF: 262144 p-value: 0.83193	Hosking: 261329 DOF: 262144 p-value: 0.869945	Toeplitz: 0 Mahdi-McLeod: inf DOF: 198132 p-value: 0

	256x128	65 couples of correlation vectors	65 couples of correlation vectors	116 couples of correlation vectors	LR: -nan p-value: 0	Li-McLeod: 2.10946e+06 DOF: 2097152 p-value: 9.85614e-10	Hosking: 2.11357e+06 DOF: 2097152 p-value: 6.66134e-16	Toeplitz: -0 Mahdi-McLeod: inf DOF: 1.57898e+06 p-value: 0
MVB	16384x2	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: 1.6078 p-value: 0.657623	Li-McLeod: 7.46406 DOF: 8 p-value: 0.487488	Hosking: 7.46406 DOF: 8 p-value o: 0.487488	Toeplitz: 0.999361 Mahdi-McLeod: 6.28405 DOF: 7.2 p-value: 0.529446
	8192x4	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: 8.02202 p-value: 0.626686	Li-McLeod: 54.7654 DOF: 64 p-value: 0.787988	Hosking: 54.7621 DOF : 64 p-value: 0.78808	Toeplitz: 0.982975 Mahdi-McLeod: 46.8906 DOF: 53.3333 p-value: 0.720924
	4096x8	Reject none of the null hypotheses	Reject none of the null hypotheses	Reject none of the null hypotheses	LR: 26.3347 p-value: 0.881028	Li-McLeod: 478.562 DOF: 512 p-value: 0.852601	Hosking: 478.501 DOF : 512 p-value: 0.853061	Toeplitz : 0.576222 Mahdi-McLeod: 398.465 DOF:

								406.588 p-value: 0.603858
	2048x16	4 couples of correlation vectors	4 couples of correlation vectors	4 couples of correlation vectors	LR: nan p-value: 0	Li-McLeod: -6.92575e+53 DOF: 4096 p-value: 1	Hosking: -6.96188e+53 DOF: 4096 p-value: 1	GV Test Statistic is infinite
	1024x32	12 couples of correlation vectors	12 couples of correlation vectors	13 couples of correlation vectors	LR : 234346 p-value: 0			
	512x64	7 couples of correlation vectors	7 couples of correlation vectors	91 couples of correlation vectors	LR: 513138 p-value: 0			
	256x128	10 couples of correlation vectors	10 couples of correlation vectors	335 couples of correlation vectors	LR: 439668 p-value: 0			
COVEU	16384x2	Reject none of the null hypothes es	Reject none of the null hypothese s	Reject none of the null hypothese s	LR: -nan p-value: 0	Li-McLeod: 8.755 DOF: 8 p-value : 0.363384	Hosking: 8.75518 DOF: 8 p-value: 0.363368	Toeplitz: 0.999316 Mahdi-McLeod: 6.72335

								DOF: 7.2 p-value: 0.480396
	8192x4	Reject none of the null hypotheses	Reject none of the null hypotheses	4 couples of correlation vectors	LR: -nan p-value: 0	Li-McLeod: 74.1424 DOF: 64 p-value : 0.181071	Hosking: 74.1464 DOF: 64 p-value: 0.180988	Toeplitz: 0.97841 Mahdi-McLeod: 59.5996 DOF: 53.3333 p-value: 0.25834
	4096x8	12 couples of correlation vectors	12 couples of correlation vectors	24 couples of correlation vectors	LR: -nan p-value: 0	Li-McLeod: 554.122 DOF: 512 p-value: 0.0963633	Hosking: 554.172 DOF: 512 p-value: 0.0961125	Toeplitz: 0.539411 Mahdi-McLeod: 446.182 DOF: 406.588 p-value: 0.0855009
	2048x16	88 couples of correlation vectors	88 couples of correlation vectors	91 couples of correlation vectors	LR: -nan p-value: 0	Li-McLeod: 4261.91 DOF: 4096	Hosking: 4262.56 DOF: 4096	Toeplitz: 6.98258e-09 Mahdi-

						p-value: 0.034672	p-value: 0.0341384	McLeod: 3496.47 DOF: 3165.09 p-value: 2.71962e-05
	1024x32	349 couples of correlation vectors	345 couples of correlation vectors	310 couples of correlation vectors	LR: -nan p-value: 0	Li-McLeod: 32193.7 DOF: 32768 p-value: 0.987904	Hosking: 32180.9 DOF: 32768 p-value: 0.989403	GV Test Statistic is infinite
	512x64	1114 couples of correlation vectors	1075 couples of correlation vectors	982 couples of correlation vectors	LR: -nan p-value: 0	Li-McLeod: 262818 DOF: 262144 p-value: 0.175812	Hosking: 262916 DOF: 262144 p-value: 0.143221	Toeplitz: -0 Mahdi-McLeod: inf DOF: 198132 p-value: 0
	256x128	1464 couples of correlation vectors	1349 couples of correlation vectors	1613 couples of correlation vectors	LR: -nan p-value: 0	Li-McLeod: 2.09166e+06 DOF:	Hosking: 2.08978e+06 DOF:	Toeplitz: -0 Mahdi-McLeod:

						2097152 p-value: 0.996347	2097152 p-value: 0.999842	inf DOF : 1.57898e+06 p-value: 0

Table A. 2. Les résultats du test statistique des générateurs de nombres pseudo-aléatoires LCG 2^{64} Microsoft Visual Basic 6.0, LCG 2^{31} RANDU d'IBM, LCG 2^{32} Marsaglia, LCG 2^{32} Fishman, LCG 2^{48} Java, LCG 2^{15} ANSI C BSD avec deux techniques de parallélisation « découpé par bloc » (sequence splitting en anglais) et « tourniquet » (roundrobin en anglais).

3 - Les codes de l'outil RSPR et l'implémentation de la simulation stochastique de croissance démographique avec la méthode Monte Carlo

Comme nous l'avons présenté dans le chapitre « Proposition », notre outil RSPR utilise le progiciel hwloc pour l'affinité entre les processus et les cœurs physiques et logiques. Il faut donc installer ce progiciel avant de lancer cet outil.

Tout d'abord, nous présentons notre spécification pour le programme « make » pour faire automatiquement la compilation de l'outil RSPR et la simulation stochastique de croissance démographique. Ce fichier « Makefile » contient des fichiers du code source pour exécuter un ensemble d'actions, par exemple : rspr : rspr.c, genMRIP.c, data.c et TGenMT.c.

```
CC=gcc           ### icc
CXX=g++          ### icpc
LDFLAGS=-lm -O2  ### -mmic  for Intel Xeon Phi

all: rspr calnum simuo simup simur simseq genMT

rspr: rspr.c genMRIP.c data.c TGenMT.c
        $(CC) $^ -o $@ $(LDFLAGS)

calnum: calnum.c genMRIP.c data.c runSim.c TGenMT.c
        $(CC) $^ -o $@ $(LDFLAGS)

genMT: mainMt.c TGenMT.c
        $(CC) $^ -o $@

simuo: simuo.cpp TGenMT.cpp simuPo.cpp
        $(CXX) $^ -o $@ $(LDFLAGS) -fopenmp

simseq: simuseq.cpp TGenMT.cpp simuPo.cpp
        $(CXX) $^ -o $@ $(LDFLAGS)

simup: simup.cpp TGenMT.cpp simuPo.cpp
        $(CXX) $^ -o $@ $(LDFLAGS) -lpthread

simur: simur.cpp TGenMT.cpp simuPo.cpp
        $(CXX) $^ -o $@ $(LDFLAGS)

clean:
        rm -f *.o simuo simup simur simseq rspr calnum genMT
```

Code A. 1. Le fichier « Makefile » de l'outil RSPR.

```
/* ------------------------------------------------------------ */
/*   rspr.c: A tool runs a stochastic simulation in parallel    */
/*           and guarantee numerical reproducibility            */
/*   Input : stochastic simulation, inputs, parameters          */
/*   Output: Average or total calculated                        */
/*   Created by DAO Van Toan - v1.0  2016                       */
/* ------------------------------------------------------------ */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <time.h>
```

```
#include <sys/time.h>
#include <math.h>
#include "data.h"
#include "genMRIP.h"

#define SIZE_SIMU 35000

int main (int argc, char **argv)
{
   double  total = 0., average, average_in, total_in = 0.;
   double  cof_eff, scart, imme;
   double  res_imm[SIZE_SIMU] = { 0. };
   int     nbCore, nbRep, jj, ii, nBlocks, nbRes, nbFile = 0;
   int     nbCPU_in,typeCal, nbPU, nbSock, chFork, nbLimit = 15000;
   char    prog[50], indata[50], command[200], buffer[80];
   char  * path_work="./working";
   char  * path_data="./data";
   char  * path_out="./out";
   FILE  * fp;
   struct  timeval start_t, finish_t;

   if(argc < 6 || argc > 6){
      printf("rspr.c - e000: args invalid: ./rspr"
                    "program nbCPU nbRep inputfile type\n");
      printf("rspr.c: type: 0 - total, "
             "1 - average,deviation standard, confidence interval\n");
      return -1;
   }

   strcpy(prog,argv[1]);
   nbCPU_in = atoi(argv[2]);
   nbRep = atoi(argv[3]);
   strcpy(indata,argv[4]);
   typeCal = atoi(argv[5]);

   if(typeCal != 0 && typeCal != 1)
   {
      printf("rspr.c - e001: args invalid: 0 - total,"
                     "1 - average,deviation std,interval\n");
      return -1;
   }

   /*Get number of processor per core*/
   fp = popen ("hwloc-calc --number-of pu socket:0.core:0", "r");
   if (fp!=NULL)
   {
      fscanf(fp, "%d", &nbPU);
      pclose(fp);
   }else {
      printf("rspr.c - e00p: Can't open file!\n");
      exit(1);
   }

   /*Get number of sockets in machine*/
   fp = popen ("hwloc-calc --number-of socket all", "r");
   if (fp!=NULL)
   {
      fscanf(fp, "%d", &nbSock);
      pclose(fp);
   } else {
      printf("rspr.c - e002: Can't open file!\n");
      exit(1);
   }

   if(check_folder(path_data) == 0){
      create_folder(path_data);
      genMT_new(nbRep);
   }else if(check_folder(path_data) == 1
```

```
          && nbRep > count_file(path_data)
          && nbRep < 100000)
   {
      genMT_res(count_file(path_data), nbRep);
   }

   if(check_folder(path_work) == 0){
      create_folder(path_work);
   }

   /*Number of blocks with nbProc*/
   nbCore    = nbCPU_in/nbPU;
   int call  = nbCPU_in%nbPU;
   if(nbCore  == 0 || call != 0)
   {
      printf("rspr.c - e003: Number of CPU should be even or Nx%d \n",nbPU);
      return -1;
   }

   nBlocks = nbRep / nbCPU_in;
   nbRes = nbRep % nbCPU_in;

   /*Delete old datas*/
   if(count_file(path_work)!= 0)
      remove_file(path_work);

   /*Create shell scripts for MRIP technic*/
   genSimu(nbCore, prog, indata);
   if(nBlocks == 0 || nbRes != 0)
   {
      genRes(nbCore,nBlocks,nbRes,prog,indata);
   }

   /*begin of time calculating*/
   if(gettimeofday(&start_t,NULL) != 0){
      printf("rspr.c - e004: gettime error for starting\n");
      return -1;
   }

   /*Runs blocks of processus in parallel */
   jj = nBlocks - 1;
   do {
      sprintf(command, "./simuMRIP %d", jj);
      system(command);
      jj--;
   }while (jj >= 0);

   if(nBlocks == 0 || nbRes != 0){
      sprintf(command, "./simuRes %d %d", nBlocks, nbRes);
      system(command);
   }

   /*End of time calculating*/
   if(gettimeofday(&finish_t,NULL) != 0){
      printf("rspr.c - e005: gettime error for stoping\n");
      return -1;
   }

   /*Save intermediate results*/
   sprintf(command, "out/res_%d.im", nbCPU_in);

   fp  = fopen(command, "w");
   if(fp == NULL){
      printf("rspr.c - e006: Can't open file to write output\n");
      return -1;
   }

   for(jj = 0; jj < nbRep; jj++)
```

```
   {
      fprintf(fp, "%d : %14.10f\n", jj, res_imm[jj]);
   }
   fclose(fp);

   /*Verify intermediate results between seq vs parallel*/
   int idenCom, fileStt;
   fileStt  = file_exist("out/res_seq.im");
   if(fileStt == 1)
      idenCom = compare_intermediate("out/res_seq.im",command);

   /*Reduction of results*/
   ii = 0;
   while(ii < nbRep)
   {
      sprintf(command, "working/resRep_%06d", ii);
      fp = fopen(command,"r");
      if(fp == NULL){
         printf("rsprc.c - e007: Can't open file to read output\n");
         return -1;
      }
      fread(&buffer, sizeof(double), 1, fp);
      res_imm[ii] = atof(buffer);
      fclose(fp);

      total    += res_imm[ii];
      total_in += res_imm[ii] * res_imm[ii];

      ii++;
   }

   average    = total/(double) nbRep;
   average_in = total_in/(double) nbRep;
   imme       = average_in - (average * average);
   scart      = sqrt(imme);
   cof_eff    = (2.5758 * scart)/sqrt(nbRep);

   /*Write final results to a file*/
   sprintf(command, "out/resFinal_%d", nbCPU_in);

   fp  = fopen(command, "w");
   if(fp == NULL){
      printf("rspr.c - e008: Can't open file to write output\n");
      return -1;
   }

   fprintf(fp, "Execution parallèle avec %d cores:\n", nbCPU_in);
   for(jj = 0; jj < nbRep; jj++)
   {
      fprintf(fp, "%d : %14.10f\n", jj, res_imm[jj]);
   }

   fprintf(fp,"Numbre of replications: %d\n", nbRep);
   if(typeCal == 0){
   fprintf(fp,"Total calcule: %14.10f\n", total);
   }else{
      fprintf(fp, "Moyenne: %14.10f\n", average);
      fprintf(fp, "Ecart-type: %14.10f\n", scart);
      fprintf(fp, "Total carree: %14.10f\n", total_in);
      fprintf(fp, "Moyenne carree: %14.10f\n",average_in);
      fprintf(fp, "Interval de confidence: %14.10f\n", cof_eff);
   }
   fprintf(fp, "Temps execution: %lf\n",
           ((finish_t.tv_sec + finish_t.tv_usec * 1e-6)
            -
           (start_t.tv_sec + start_t.tv_usec * 1e-6)));
   if(fileStt != 1) fprintf(fp,"Intermediate results of sequence"
                                              " does not exist\n");
```

```
   else{
      if(idenCom == -1) fprintf(fp,"Cannot open files to compare"
                                        " sequence vs parallel\n");
      else if(idenCom == 0) fprintf(fp, "Intermediate results are different\n");
      else fprintf(fp,"Intermediate results are identical\n");
   }
   fclose(fp);

   /*Delete temporary file*/
   if(count_file(path_work) != 0)
     remove_file(path_work);

   return 0;
}
```

Code A. 2. Le code rspr.c de l'outil RSPR.

```
void genSimu(int nbCore, char * nameProg, char * inFile);
void genRes(int nbProc, int nBlocks, int nbRes, char * nameProg, char * inFile);
void genRes_sup(int nbCores, int nbRep_out, char * nameProg, char * inFile);
```

Code A. 3. Le code genMRIP.h de l'outil RSPR

```
/* ------------------------------------------------------------ */
/*   genMRIP.c: Generate shell scripts for simu program         */
/*              to allow the programm to be run in parallel     */
/*   Input : None                                               */
/*   Output: Shell scripts for simu program                     */
/*   Created by DAO Van Toan - August 2016                      */
/* ------------------------------------------------------------ */

#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

/* ------------------------------------------------------------ */
/*   genSimu: Create a shell script that has nbProc executions  */
/*              in parallel                                     */
/*   Input : number of cores/processors                         */
/*   Output: Shell scripts for simu program                     */
/* ------------------------------------------------------------ */
void genSimu(int nbCore, char * nameProg, char * inFile)
{
  int    ii = 0, jj, nbSock, nbCPSoc, nbPU, count=0, kk, tt;
  char   command[50];
  FILE * fp;

  /*Get number of sockets in machine*/
  fp = popen ("hwloc-calc --number-of socket all", "r");
  if (fp!=NULL)
  {
     fscanf(fp, "%d", &nbSock);
     pclose(fp);
  }else {
     printf("genMRIP.c - e000: Can't open file!\n");
     exit(1);
  }

  /*Get number of cores per socket*/
  fp = popen ("hwloc-calc --number-of core socket:0", "r");
  if (fp!=NULL)
  {
     fscanf(fp, "%d", &nbCPSoc);
```

```
    pclose(fp);
}else {
    printf("genMRIP.c - e001: Can't open file!\n");
    exit(1);
}


/*Get number of processor per core*/
fp = popen ("hwloc-calc --number-of pu socket:0.core:0", "r");
if (fp!=NULL)
{
    fscanf(fp, "%d", &nbPU);
    pclose(fp);
}else {
    printf("genMRIP.c - e002: Can't open file!\n");
    exit(1);
}


/* Creat a shell script */
FILE *file = fopen("simuMRIP", "w");

if (file==NULL)
{
    printf("genMRIP.c - e003: Can't open file!\n");
    exit(1);
}

const char * text = "#!/bin/bash";
fprintf(file,"%s \n",text);
fprintf(file,"\n");
fprintf(file,"#number of physical cores from input:%d\n", nbCore);
fprintf(file,"#number of sockets in machine:%d\n", nbSock);
fprintf(file,"#number of cores per socket:%d\n", nbCPSoc);
fprintf(file,"#number of logical processors per core:%d\n", nbPU);
fprintf(file,"\n");
fprintf(file,"listPids=\"\"\n");

fprintf(file,"t=$(expr $1 \\* %d \\* %d)\n", nbPU, nbCore);
fprintf(file,"\n");

if(nbSock == 1) {
  for(tt = 0; tt < nbPU; tt++){
     for(kk = 0; kk < nbCore; kk++)
     {
        fprintf(file,"idRep%d=$(expr $t + %d)", ii, ii);
        fprintf(file,"\n");
        fprintf(file,"hwloc-bind socket:0.core:%d.pu:%d"
           "./%s %s $idRep%d & \n",kk,tt, nameProg, inFile, ii);
        fprintf(file,"listPids=\"$listPids $!\"\n");
        fprintf(file,"\n");
        ii++;
     }
  }
} else if(nbCore == 1){
   int nbRep = nbPU;

   for(tt = 0; tt < nbPU; tt++){
      for(jj = 0; jj < nbSock; jj++)
      {
         if(ii < nbRep){
            fprintf(file,"idRep%d=$(expr $t + %d)", ii, ii);
            fprintf(file,"\n");
            fprintf(file,"hwloc-bind socket:%d.core:1.pu:%d"
               "./%s %s $idRep%d & \n", jj,tt, nameProg, inFile, ii);
            fprintf(file,"listPids=\"$listPids $!\"\n");
            fprintf(file,"\n");
            ii++;
```

```
          }
        }
      }
  }else {

    int nbRep = nbPU * nbCore;
    nbCore = nbCore/nbSock;

    for(tt = 0; tt < nbPU; tt++){
       for(kk = 0; kk < nbCore; kk++){
          for(jj = 0; jj < nbSock; jj++)
          {
             if(ii < nbRep){
                fprintf(file,"idRep%d=$(expr $t + %d)", ii, ii);
                fprintf(file,"\n");
                fprintf(file,"hwloc-bind socket:%d.core:%d.pu:%d"
                   " ./%s %s $idRep%d & \n",jj,kk, tt, nameProg, inFile, ii);
                fprintf(file,"listPids=\"$listPids $!\"\n");
                fprintf(file,"\n");
                ii++;
             }
          }
        }
     }
  }

  fprintf(file,"wait $listPids\n");
  fprintf(file,"\n");

  fclose(file);

  sprintf(command, "chmod +x ./simuMRIP");
  system(command);

}

/* ------------------------------------------------------------ */
/*   genRes: Create a additional shell script for a case of     */
/*           redundant in the division between nbRep and nbProc  */
/*   Input : number of cores, blocs, redundancy                 */
/*   Output: Additional shell script for simu program           */
/* ------------------------------------------------------------ */
void genRes(int nbCore, int nBlocks, int nbRes, char * nameProg, char * inFile)
{
  int    ii=0, jj, tt,kk, nbSock, nbPU, count=0;
  char   command[50];
  FILE * fp;

  /*Get number of sockets in machine*/
  fp = popen ("hwloc-calc --number-of socket all", "r");
  if (fp!=NULL)
  {
     fscanf(fp, "%d", &nbSock);
     pclose(fp);
  }else{
     printf("genMRIP.c - e004: Can't open file!\n");
     exit(1);
  }

  /*Get number of processor per core*/
  fp = popen ("hwloc-calc --number-of pu socket:0.core:0", "r");
  if (fp!=NULL)
  {
     fscanf(fp, "%d", &nbPU);
     pclose(fp);
  }else{
     printf("genMRIP.c - e005: Can't open file!\n");
     exit(1);
```

```
  }

  /* Creat a shell script */
  FILE *file = fopen("simuRes", "w");

  if (file==NULL)
  {
     printf("genMRIP.c - e006: Can't open file!\n");
     exit(1);
  }

  int aBloc = nbCore * nbPU;
  const char * text = "#!/bin/bash";
  fprintf(file,"%s \n",text);
  fprintf(file,"\n");
  fprintf(file,"listPids=\"\"\n");
  fprintf(file,"t=$(expr %d \\* %d)\n", aBloc, nBlocks);
  fprintf(file,"\n");

  if(nbRes == 1){
     fprintf(file,"idRep%d=$(expr $t + %d)", ii, ii);
     fprintf(file,"\n");
     fprintf(file,"hwloc-bind socket:0.core:0.pu:0"
        " ./%s %s $idRep%d & \n", nameProg, inFile, ii);
     fprintf(file,"listPids=\"$listPids $!\"\n");
  }

  if(nbSock == 1) {
     for(tt = 0; tt < nbPU; tt++)
        for(kk = 0; kk < nbCore; kk++)
        {
           if(ii < nbRes){
              fprintf(file,"idRep%d=$(expr $t + %d)", ii, ii);
              fprintf(file,"\n");
              fprintf(file,"hwloc-bind socket:0.core:%d.pu:%d"
                  " ./%s %s $idRep%d & \n", kk,tt, nameProg, inFile, ii);
              fprintf(file,"listPids=\"$listPids $!\"\n");
              fprintf(file,"\n");
              ii++;
           }
         }
  }else {
     nbCore = nbCore / nbSock;

     for(tt = 0; tt < nbPU; tt++)
        for(kk = 0; kk < nbCore; kk++)
           for(jj = 0; jj < nbSock; jj++)
           {
              if(ii < nbRes){
                 fprintf(file,"idRep%d=$(expr $t + %d)", ii, ii);
                 fprintf(file,"\n");
                 fprintf(file,"hwloc-bind socket:%d.core:%d.pu:%d"
                    " ./%s %s $idRep%d & \n", jj, kk, tt, nameProg, inFile, ii);
                 fprintf(file,"listPids=\"$listPids $!\"\n");
                 fprintf(file,"\n");
                 ii++;
              }
           }
     }

  fprintf(file,"wait $listPids\n");
  fprintf(file,"\n");
  fclose(file);

  sprintf(command, "chmod +x ./simuRes");
  system(command);
}
```

```
/* ------------------------------------------------------------ */
/* genRes_sup: Generate a supplemaintaire script                */
/* ------------------------------------------------------------ */

void genRes_sup(int nbCore, int nbRep_out, char * nameProg, char * inFile)
{
  int ii=0, jj, kk, tt, nbSock, count=0,  nbRes, nbSup, nbPU;
  char command[50];
  FILE *fp;

  /*Get number of sockets in machine*/
  fp = popen ("hwloc-calc --number-of socket all", "r");
  if (fp!=NULL)
  {
     fscanf(fp, "%d", &nbSock);
     pclose(fp);
  }else {
     printf("genMRIP.c - e007: Can't open file!\n");
     exit(1);
  }

  /*Get number of processor per core*/
  fp = popen ("hwloc-calc --number-of pu socket:0.core:0", "r");
  if (fp!=NULL)
  {
     fscanf(fp, "%d", &nbPU);
     pclose(fp);
  }else {
     printf("genMRIP.c - e008: Can't open file!\n");
     exit(1);
  }

  int aBloc = nbCore * nbPU;
  nbRes     = nbRep_out % aBloc;
  nbSup     = aBloc - nbRes;

  /* Creat a shell script */
  FILE *file = fopen("simuRes_sup", "w");

  if (file==NULL)
  {
     printf("genMRIP.c - e009: Can't open file!\n");
     exit(1);
  }

  const char * text = "#!/bin/bash";
  fprintf(file,"%s \n",text);
  fprintf(file,"\n");
  fprintf(file,"listPids=\"\"\n");
  fprintf(file,"t=%d\n", nbRep_out);
  fprintf(file,"\n");

  if(nbSup == 1){
     fprintf(file,"idRep%d=$(expr $t + %d)", ii, ii);
     fprintf(file,"\n");
     fprintf(file,"hwloc-bind socket:0.core:0.pu:0"
                     " ./%s %s $idRep%d & \n", nameProg, inFile, ii);
     fprintf(file,"listPids=\"$listPids $!\"\n");
  }

  if(nbSock == 1) {
     for(tt = 0; tt < nbPU; tt++)
        for(kk = 0; kk < nbCore; kk++)
        {
           if(ii < nbSup){
              fprintf(file,"idRep%d=$(expr $t + %d)", ii, ii);
              fprintf(file,"\n");
              fprintf(file,"hwloc-bind socket:0.core:%d.pu:%d"
```

```
                        " ./%s %s $idRep%d & \n", kk,tt, nameProg, inFile, ii);
               fprintf(file,"listPids=\"$listPids $!\"\n");
               fprintf(file,"\n");
               ii++;
            }
         }
      } else {

      nbCore = nbCore / nbSock;

      for(tt = 0; tt < nbPU; tt++)
         for(kk = 0; kk < nbCore; kk++)
            for(jj = 0; jj < nbSock; jj++)
            {
               if(ii < nbSup){
                  fprintf(file,"idRep%d=$(expr $t + %d)", ii, ii);
                  fprintf(file,"\n");
                  fprintf(file,"hwloc-bind socket:%d.core:%d.pu:%d"
                     " ./%s %s $idRep%d & \n", jj, kk, tt, nameProg, inFile, ii);
                  fprintf(file,"listPids=\"$listPids $!\"\n");
                  fprintf(file,"\n");
                  ii++;
               }
            }
      }

   fprintf(file,"wait $listPids\n");

   fprintf(file,"\n");
   fclose(file);

   sprintf(command, "chmod +x ./simuRes_sup");
   system(command);
}
```

Code A. 4. Le code genMRIP.c de l'outil RSPR.

```
int  count_file(char * path);
int  check_folder(char * folder);
void creat_folder(char * pathFolder);
void remove_file(char * path);
void genMT_new(int nbRep);
void genMT_res(int nbReal, int nbRep);
int  file_exist(const char * nameFile);
int  compare_intermediate(char * nameFile1, char * nameFile2);
```

Code A. 5. Le code data.h de l'outil RSPR.

```
/* ------------------------------------------------------------ */
/*   data.c: To manager datas, files and folders                */
/*   Input : None                                               */
/* ------------------------------------------------------------ */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <dirent.h>
#include <errno.h>
#include <sys/stat.h>

#include "TGenMT.h"

/* ------------------------------------------------------------ */
```

```
/*   check_folder: Check existence status of a folder           */
/*   Input : path or name of folder                             */
/*   Output: an integer is representing for a status            */
/* ------------------------------------------------------------ */
int check_folder(char * folder)
{
   int             check;
   DIR           * dir;
   struct dirent * readir;

   dir = opendir(folder);

   if(dir)
   {
      /*Folder exists*/
      check = 1;
      closedir(dir);
   }else if(errno == ENOTDIR){
      /*Need for creating of  a new folder*/
      check = 0;
   }else{
      perror ("data.c - e000: Can't open folder with opendir()\n");
   }

   return check;
}

/* ------------------------------------------------------------ */
/*   create_folder: Create a new folder                         */
/*   Input : path or name of folder                             */
/*   Output: a folder created with a pricise path               */
/* ------------------------------------------------------------ */
void create_folder(char * pathFolder)
{
   char command[256];

   sprintf(command,"mkdir -p %s", pathFolder);
   if(system(command) == -1)
      perror("data.c - e001: Can't creat a new folder");
}

/* ------------------------------------------------------------ */
/*   count_file: Count number of files in a folder              */
/*   Input : path or name of folder                             */
/*   Output: number of files                                    */
/* ------------------------------------------------------------ */
int count_file( char * path)
{
  int             count = 0;
  DIR           * dir;
  struct dirent * readir;

  dir = opendir(path);

  if(dir)
  {
    while((readir = readdir (dir)) != NULL)
    {
       /*Don't count parent and child's folder*/
       if((strcmp(readir->d_name, ".")) != 0
            &&
          strcmp(readir->d_name, "..") != 0)
         count++;
    }

    closedir (dir);
  }
  else
```

```
    perror ("data.c - e002: Can't open this folder");

  return count;
}

/* ------------------------------------------------------------ */
/*   remove_file: Delete all of files in a folder               */
/*   Input : path or name of folder                             */
/*   Output: None                                               */
/* ------------------------------------------------------------ */
void remove_file( char * path)
{
  DIR           * dir;
  char            buf[256];
  struct dirent * readir;

  dir = opendir(path);

  if(dir)
  {
    while((readir = readdir (dir)) != NULL)
    {
      if(strcmp(readir->d_name, ".") != 0
            &&
         strcmp(readir->d_name, "..") != 0)
      {
         sprintf(buf, "%s/%s", path, readir->d_name);
         remove(buf);
      }
    }
    closedir (dir);
  }
  else
    perror ("data.c - e003: Can't open folder to delete files");
}

/* ------------------------------------------------------------ */
/*   genMT_new: Generate a set of initial status of PRNG        */
/*   Input : Number of replications                             */
/*   Output: None                                               */
/* ------------------------------------------------------------ */
void genMT_new(int nbRep){
  int i,j;
  unsigned long init[4]={0x123, 0x234, 0x345, 0x456}, length=4;

  init_by_array(init, length);

  for(j = 0; j < nbRep; j++)
  {
     char nameFile[50];
     sprintf(nameFile,"data/inStatus_%06d",j);

     saveStatus(nameFile);

     for(i = 0; i < 1000000000; i++)
     {
        mtRand();
     }
  }
}

/* ------------------------------------------------------------ */
/*   genMT_res: Add the missing initial status                  */
/*   Input : Current number of initial status and               */
/*           number of replication                              */
/*   Output: None                                               */
/* ------------------------------------------------------------ */
void genMT_res(int nbReal, int nbRep){
```

```
  int i, t, j, kk = nbReal - 1;
  unsigned long init[4]={0x123, 0x234, 0x345, 0x456}, length=4;

  init_by_array(init, length);

  char nameFile[50];
  sprintf(nameFile, "data/inStatus_%06d", kk);
  restoreStatus(nameFile);

  for(t = 0; t < 1000000000; t++)
  {
        mtRand();
  }

  for(j = nbReal; j < nbRep; j++)
  {
     char nameFile[50];
     sprintf(nameFile,"data/inStatus_%06d",j);

     saveStatus(nameFile);

     for(i = 0; i < 1000000000; i++)
     {
        mtRand();
     }
  }
}

/* ------------------------------------------------------------ */
/*   search_file: Find a file in a folder                       */
/*   Input : Name of file                                       */
/*   Output: Status of file                                     */
/* ------------------------------------------------------------ */
int file_exist(const char *nameFile)
{
    struct stat buf;
    int checkValue = 0;

    if (stat(nameFile, &buf) == 0)
      checkValue = 1;

   return checkValue;
}

/* ------------------------------------------------------------ */
/*   compare_immediate: Compare two files (immediate results)   */
/*   Input : two files                                          */
/*   Output: Status of comparaison                              */
/* ------------------------------------------------------------ */
int compare_intermediate(char * nameFile1, char * nameFile2)
{
   FILE * fp1, * fp2;
   int    char1, char2, status = 0;

   fp1 = fopen(nameFile1,"r");
   fp2 = fopen(nameFile2,"r");

   /*Cannot open file to compare*/
   if(fp1 == NULL || fp2 == NULL) {
      status = -1;
   }

   char1 = getc(fp1);
   char2 = getc(fp2);

   while((char1 != EOF) && (char2 != EOF) && (char1 == char2))
   {
      char1 = getc(fp1);
```

```
      char2 = getc(fp2);
   }

   if(char1 == char2) status = 1;

   fclose(fp1);
   fclose(fp2);

   return status;

}
```

Code A. 6. Le code data.c de l'outil RSPR.

```
/* ---------------------------------------------------------------*/
/* C code for Matsumoto & Nishimura - Mersene Twister             */
/* This file is updated by DAO Van Toan - May 2016                */
/* from the updates developped version adapted by David. Hill     */
/* ---------------------------------------------------------------*/
void init_genrand(unsigned long s);
void init_by_array(unsigned long init_key[], int key_length);
void saveStatus(char * inFileName);
void restoreStatus(char * inFileName);
double mtRand(void);
void init_genrand(unsigned long s);
void init_by_array(unsigned long init_key[], int key_length);
unsigned long genrand_int32(void);
double genrand_real1(void);
double genrand_real2(void);
```

Code A. 7. Le code TGenMT.h de l'outil RSPR.

```
/* ---------------------------------------------------------------*/
/* C code for Matsumoto & Nishimura - Mersene Twister             */
/* This version is compact with a limited API & 3 new functions */
/* 2 functions to save and restore statuses                       */
/* 1 function mtRand optimized for speed (MT is already one if  */
/*   the fastest high quality generator, if not the fastest with*/
/*   MLFG from Michael Mascagni et al                             */
/* Updates by D. Hill - March 2016                                */
/* ---------------------------------------------------------------*/
#include <stdio.h>
#include <stdlib.h>
#include "TGenMT.h"

/* Period parameters */
#define N 624
#define M 397
#define MATRIX_A 0x9908b0dfUL   /* constant vector a */
#define UPPER_MASK 0x80000000UL /* most significant w-r bits */
#define LOWER_MASK 0x7fffffffUL /* least significant r bits */

static unsigned long mt[N]; /* the array for the state vector  */
static int mti=N+1; /* mti==N+1 means mt[N] is not initialized */

/* -------------- Prototypes ------------------------------- */
void init_genrand(unsigned long s);
void init_by_array(unsigned long init_key[], int key_length);

/* ---------------------------------------------------------- */
/* saveStatus         Saves the MT status in a text File      */
/*                                                            */
/* Input: inFileName  The file name                  D. Hill */
/* ---------------------------------------------------------- */
```

```
void saveStatus(char * inFileName)
{
    FILE * fp  = fopen(inFileName, "w");
    int    i   = 0;

    if(fp == NULL){
      printf("Error: Can't open file to save\n");
      exit(1);
    }

    fprintf(fp, "%d\n", mti);

    for(i = 0 ; i < N ; i++)
    {
       fprintf(fp, "%ld ", mt[i]);
    }
    fclose(fp);
}

/* ------------------------------------------------------------ */
/* restoreStatus     Restores the MT status from a text File   */
/*                                                              */
/* Input: inFileName  The file name                   D. Hill */
/* ------------------------------------------------------------ */
void restoreStatus(char * inFileName)
{
    FILE * fp  = fopen(inFileName, "r");
    int    i   = 0;

    if(fp == NULL){
      printf("Error: Can't open file to read\n");
      exit(1);
    }

    fscanf(fp, "%d", &mti);

    for(i = 0 ; i < N ; i ++)
    {
       fscanf(fp, "%ld", &mt[i]);
    }

    fclose(fp);
}

/* ------------------------------------------------------------*/
/* mtRand   Fast generation of a double precision pseudo random */
/*            number with the Mersenne Twister algorithm        */
/*            Uniform on [0,1)-real-interval                    */
/*                                                              */
/* This version is including the code of genrand_int32 for speed*/
/* and it also avoids the local allocation of y - interesting   */
/* for large simulations with many calls to generator           */
/* D. Hill - March 2016                                         */
/* ------------------------------------------------------------*/
double mtRand(void)
{
    static unsigned long y;
    /* y is set static for speed - avoids reallocation at each call */
    static unsigned long mag01[2]={0x0UL, MATRIX_A};
    /* mag01[x] = x * MATRIX_A  for x=0,1 */

    if (mti >= N) { /* generate N words at one time */
        int kk;

        if (mti == N+1)   /* if init_genrand() has not been called, */
            init_genrand(5489UL); /* a default initial seed is used */

        for (kk = 0; kk < N-M; kk++) {
```

```
            y = (mt[kk]&UPPER_MASK)|(mt[kk+1]&LOWER_MASK);
            mt[kk] = mt[kk+M] ^ (y >> 1) ^ mag01[y & 0x1UL];
        }
        for (; kk < N-1; kk++) {
            y = (mt[kk]&UPPER_MASK)|(mt[kk+1]&LOWER_MASK);
            mt[kk] = mt[kk+(M-N)] ^ (y >> 1) ^ mag01[y & 0x1UL];
        }
        y = (mt[N-1]&UPPER_MASK)|(mt[0]&LOWER_MASK);
        mt[N-1] = mt[M-1] ^ (y >> 1) ^ mag01[y & 0x1UL];

        mti = 0;
    }

    y = mt[mti++];

    /* Tempering */
    y ^= (y >> 11);
    y ^= (y << 7) & 0x9d2c5680UL;
    y ^= (y << 15) & 0xefc60000UL;
    y ^= (y >> 18);

    return y *(1.0/4294967296.0);
    /* divided by 2^32 */
}

/* ---------------------------------------------------- */
/* init_genrand                                         */
/*                                                      */
/* Input: initializes mt[N] with a simple int seed - s */
/*                                                      */
/* This can be confusing since the MT status is huge   */
/* ---------------------------------------------------- */
void init_genrand(unsigned long s)
{
    mt[0]= s & 0xffffffffUL;
    for (mti=1; mti<N; mti++) {
        mt[mti] =
            (1812433253UL * (mt[mti-1] ^ (mt[mti-1] >> 30)) + mti);
        /* See Knuth TAOCP Vol2. 3rd Ed. P.106 for multiplier. */
        /* In the previous versions, MSBs of the seed affect   */
        /* only MSBs of the array mt[].                        */
        /* 2002/01/09 modified by Makoto Matsumoto             */
        mt[mti] &= 0xffffffffUL;
        /* for >32 bit machines */
    }
}

/* ---------------------------------------------------- */
/* init_by_array     initializes MT by an array        */
/*                                                      */
/* Input:  init_key is the array for initializing keys */
/*         key_length is its length                     */
/*         slight changes for C++, 2004/2/26           */
/* ---------------------------------------------------- */
void init_by_array(unsigned long init_key[], int key_length)
{
    int i, j, k;
    init_genrand(19650218UL);
    i=1; j=0;
    k = (N>key_length ? N : key_length);
    for (; k; k--) {
        mt[i] = (mt[i] ^ ((mt[i-1] ^ (mt[i-1] >> 30)) * 1664525UL))
          + init_key[j] + j; /* non linear */
        mt[i] &= 0xffffffffUL; /* for WORDSIZE > 32 machines */
        i++; j++;
        if (i>=N) { mt[0] = mt[N-1]; i=1; }
        if (j>=key_length) j=0;
    }
```

```
    for (k=N-1; k; k--) {
        mt[i] = (mt[i] ^ ((mt[i-1] ^ (mt[i-1] >> 30)) * 1566083941UL))
          - i; /* non linear */
        mt[i] &= 0xffffffffUL; /* for WORDSIZE > 32 machines */
        i++;
        if (i>=N) { mt[0] = mt[N-1]; i=1; }
    }

    mt[0] = 0x80000000UL; /* MSB is 1; assuring non-zero initial array */
}

/* ---------------------------------------------------------------*/
/* genrand_int32        generates an integer random number      */
/*                                                               */
/* Output: a random number on the [0,0xffffffff]-interval       */
/* ---------------------------------------------------------------*/
unsigned long genrand_int32(void)
{
    unsigned long y;
    static unsigned long mag01[2]={0x0UL, MATRIX_A};
    /* mag01[x] = x * MATRIX_A  for x=0,1 */

    if (mti >= N) { /* generate N words at one time */
        int kk;

        if (mti == N+1)   /* if init_genrand() has not been called, */
            init_genrand(5489UL); /* a default initial seed is used */

        for (kk=0;kk<N-M;kk++) {
            y = (mt[kk]&UPPER_MASK)|(mt[kk+1]&LOWER_MASK);
            mt[kk] = mt[kk+M] ^ (y >> 1) ^ mag01[y & 0x1UL];
        }
        for (;kk<N-1;kk++) {
            y = (mt[kk]&UPPER_MASK)|(mt[kk+1]&LOWER_MASK);
            mt[kk] = mt[kk+(M-N)] ^ (y >> 1) ^ mag01[y & 0x1UL];
        }
        y = (mt[N-1]&UPPER_MASK)|(mt[0]&LOWER_MASK);

        if (mti == N+1)   /* if init_genrand() has not been called, */
            init_genrand(5489UL); /* a default initial seed is used */

        for (kk=0;kk<N-M;kk++) {
            y = (mt[kk]&UPPER_MASK)|(mt[kk+1]&LOWER_MASK);
            mt[kk] = mt[kk+M] ^ (y >> 1) ^ mag01[y & 0x1UL];
        }
        for (;kk<N-1;kk++) {
            y = (mt[kk]&UPPER_MASK)|(mt[kk+1]&LOWER_MASK);
            mt[kk] = mt[kk+(M-N)] ^ (y >> 1) ^ mag01[y & 0x1UL];
        }
        y = (mt[N-1]&UPPER_MASK)|(mt[0]&LOWER_MASK);
        mt[N-1] = mt[M-1] ^ (y >> 1) ^ mag01[y & 0x1UL];

        mti = 0;
    }

    y = mt[mti++];

    /* Tempering */
    y ^= (y >> 11);
    y ^= (y << 7) & 0x9d2c5680UL;
    y ^= (y << 15) & 0xefc60000UL;
    y ^= (y >> 18);

    return y;
}

/* ---------------------------------------------------------------*/
/* genrand_real1  generates an random number in double precision*/
```

```
/*                                                                      */
/* Output: a random number on the [0,1]-real-interval                   */
/* -------------------------------------------------------------------*/
double genrand_real1(void)
{
    return genrand_int32()*(1.0/4294967295.0);
    /* divided by 2^32-1 */
}
/* -------------------------------------------------------------------*/
/* genrand_real2 generates an random number in double precision */
/*                                                                      */
/* Output: a random number on the [0,1)-real-interval                   */
/* -------------------------------------------------------------------*/
double genrand_real2(void)
{
    return genrand_int32()*(1.0/4294967296.0);
    /* divided by 2^32 */
}
/* -------------------------------------------------------------------*/
```

Code A. 8. Le code TGenMT.c de l'outil RSPR.

```
#include "TGenMT.hpp"
using namespace std;

class simuPo {
   public:
            simuPo();
            simuPo(double lam, int id);
            void   setSimu(double lamda, int id);
            double getLamd();
            int    getId();
            unsigned long long int poisson();
            unsigned long long int growPo();
   private:
            double    lambda;
            int       idSimu;
            TGenMT  * mtGen;
};
```

Code A. 9. Le code simuPo.hpp de la simulation stochastique de croissance démographique.

```
/* ------------------------------------------------------------ */
/*  simuPo.cpp: Create a stochastic simulation of demographic   */
/*              using Poison law with Monte Carlo method        */
/*  Input : Lambda and identification of simulation             */
/*  Output: None                                                */
/* ------------------------------------------------------------ */

#include <iostream>
#include <cstdio>
#include <cstdlib>
#include <sstream>
#include <cstring>
#include <cmath>

#include "TGenMT.hpp"
#include "simuPo.hpp"

#define NBGENERATION 20
#define NBINDIV 2

using namespace std;
```

```
/* ------------------------------------------------------------ */
/*   Define funtions and construct a new object                 */
/* ------------------------------------------------------------ */
simuPo::simuPo()
{
   lambda = 0.;
   idSimu = 0;
}

simuPo::simuPo(double lam, int id )
{
   setSimu(lam,id);
}

void simuPo::setSimu(double lam, int id)
{
   lambda = lam;
   idSimu = id;
}

double simuPo::getLamd()
{
   return lambda;
}

int simuPo::getId()
{
   return idSimu;
}

/* ------------------------------------------------------------ */
/* Ouput: A value of the poisson law with MT generator          */
/* ------------------------------------------------------------ */
unsigned long long int simuPo::poisson(){
   double        s = getLamd();
   long long int r = -1;

   while(s >= 0)
   {
       r = r + 1;
       s = s + log(mtGen->mtRand());
   }

   return r;
}

/* ------------------------------------------------------------*/
/* Ouput: A value of the population growth for a replication    */
/* ------------------------------------------------------------*/
unsigned long long int simuPo::growPo() {
   unsigned long long int  individu = NBINDIV, nbDes;
   int                     iter = 0, ind;
   int                     replicationId = getId();
   char                   *randFileStatus = new char[80];

   sprintf(randFileStatus,"data/inStatus_%06d", replicationId);

   mtGen = new TGenMT(randFileStatus);

   while(individu > 0 && iter < NBGENERATION)
   {
      nbDes = 0;

      for(ind = 1; ind <= individu; ind++)
      {
         nbDes = nbDes + poisson();
      }
```

```
      individu = nbDes;
      iter++;
   }

   delete []randFileStatus;

   delete mtGen;

   return individu;
}
```

Code A. 10. Le code simuPo.cpp de la simulation stochastique de croissance démographique.

```
/* ------------------------------------------------------------ */
/*   simur.cpp: Run only a stochastic simulation of demographic */
/*   Input  : Parameters of simulation                          */
/*   Output : Number of population for this simulation          */
/* ------------------------------------------------------------ */

#include <iostream>
#include <cstdio>
#include <sstream>
#include <string>
#include <cstdlib>
#include <fstream>
#include <cstring>

#include "TGenMT.hpp"
#include "simuPo.hpp"

using namespace std;

int main(int argc, char **argv)
{
   double    res, lambda;
   int       idRep;
   FILE     *fp;
   simuPo   *sim = NULL;
   char      outFile[100], inFile[50];
   ifstream  readfile;

   if(argc < 2 || argc > 3)
   {
      cout << "simur.cpp - e000: invalid args:”;
      cout << ”./simu data_file idRep" << endl;
      return -1;
   }

   strcpy(inFile,argv[1]);
   idRep  = atoi(argv[2]);

   /*Get inputs from file*/
   readfile.open(inFile);
   readfile >> lambda;

   sim = new simuPo(lambda, idRep);
   res = sim->growPo();

   delete sim;

   sprintf(outFile, "working/resRep_%06d", idRep);

   fp = fopen(outFile, "w");

   if(fp == NULL){
      cout << "simur.cpp - e001: Can't open file" << endl;
```

```
      return -1;
   }

   fprintf(fp, "%lf\n", res);

   fclose(fp);

   return 0;
}
/* ----------------------------------------------------------------*/
```

Code A. 11. Le code simur.cpp de la simulation stochastique de croissance démographique

```
#include <iostream>
#include <cstdio>
#include <cstdlib>
#include <sstream>
#include <string>
#include <limits>
#include <cstdlib>
#include <cmath>
#include <ctime>
#include <sys/time.h>

#include "TGenMT.hpp"
#include "simuPo.hpp"

using namespace std;

#define SIZE_SIMU 35000

/* ----------------------------------------------------------------*/
int main(int argc, char **argv)
{
   double         seq[SIZE_SIMU] = {0.};
   double         total = 0., total_in = 0., averages, average_in;
   double         lamda, scart, cof_eff;
   int            ind, nbRep;
   char           outFile[100];
   simuPo         *instanceSM[SIZE_SIMU] = { NULL };
   struct timeval start_t, finish_t;
   FILE           *fp;

   if(argc < 2 || argc > 3)
   {
      cout << "simuseq.cpp - e001: invalid: ./simseq lambda nbRep" << endl;
      return -1;
   }

   lamda = atof(argv[1]);
   nbRep = atoi(argv[2]);

   if(gettimeofday(&start_t,NULL) != 0){
      cout << "simuseq.cpp - e002: gettime error for starting " << endl;
      return -1;
   }

   for(ind = 0; ind < nbRep; ind++)
   {
      instanceSM[ind] = new simuPo(lamda,ind);
      seq[ind]        = instanceSM[ind]->growPo();

      total    += seq[ind];
      total_in += seq[ind] * seq[ind];
   }
```

```
   if(gettimeofday(&finish_t,NULL) != 0){
      cout << "simuseq.cpp - e003: gettime error for starting " << endl;
      return -1;
   }

   for(ind = 0; ind < nbRep; ind++)
   {
      delete instanceSM[ind];
   }


   /*Save intermediate results*/
   fp  = fopen("out/res_seq.im", "w");
   if(fp == NULL){
      printf("rspr.c - e004: Can't open file to write output\n");
      return -1;
   }

   for(ind = 0; ind < nbRep; ind++)
   {
      fprintf(fp, "%d : %14.10f\n", ind, seq[ind]);
   }
   fclose(fp);

   /*Calculating*/
   averages    = total/(double) nbRep;
   average_in  = total_in/(double)nbRep;
   double imme = average_in - (averages * averages);
   scart       = sqrt(imme);
   cof_eff     = (2.5758 * scart)/sqrt(nbRep);

   /*Write all of results to file*/
   fp  = fopen("out/resRep_seq", "w");
   if(fp == NULL){
      cout << "simuseq.cpp - e005: Can't open file\n" << endl;
      return -1;
   }

   fprintf(fp, "Execution sequentielle:\n");

   for(ind = 0; ind < nbRep; ind++)
   {
      fprintf(fp, "%d : %14.10f\n", ind, seq[ind]);
   }

   fprintf(fp, "Nombre de replications: %d\n", nbRep);
   fprintf(fp, "Moyenne: %14.10f\n", averages);
   fprintf(fp, "Ecart-type: %14.10f\n", scart);
   fprintf(fp, "Total caree: %14.10f\n", total_in);
   fprintf(fp, "Moyenne carree: %14.10f\n",average_in);
   fprintf(fp, "Intervalle de confidence: %14.10f\n", cof_eff);
   fprintf(fp, "Temps d'executions: %lf\n",
               ((finish_t.tv_sec + finish_t.tv_usec * 1e-6)
               - (start_t.tv_sec + start_t.tv_usec * 1e-6)));
   fclose(fp);

   return 0;
}
/* ------------------------------------------------------------*/
```

Code A. 12. Le code séquentiel simuseq.cpp de la simulation stochastique de croissance démographique

```
#include <iostream>
#include <cstdio>
#include <sstream>
#include <string>
```

```
#include <limits>
#include <cstdlib>
#include <omp.h>
#include <cmath>
#include <ctime>
#include <sys/time.h>

#include "TGenMT.hpp"
#include "simuPo.hpp"

using namespace std;

#define SIZE_SIMU 35000

int main(int argc, char **argv)
{
   double         par[SIZE_SIMU] = { 0. };
   double         total = 0., averagep, total_in = 0., average_in;
   double         lamda, , scart, cof_eff;
   int            ind, nbRep, nthreads, nbthread = 0;
   char           outFile[100];
   simuPo         *instanceSM[SIZE_SIMU] = { NULL };
   FILE           *fp;
   struct timeval start_t, finish_t;

   if(argc < 2 || argc > 3)
   {
      cout << "Number of args invalid: ./simuo lambda nbRep" << endl;
      return -1;
   }

   lamda = atof(argv[1]);
   nbRep = atoi(argv[2]);

   if(gettimeofday(&start_t,NULL) != 0){
      cout << "simuo.cpp: gettime error for starting " << endl;
      return -1;
   }

   #pragma omp parallel for reduction(+:total,total_in) private(ind)
   for( ind= 0; ind < nbRep; ind++)
   {
      if( omp_get_thread_num() == 0)
         nbthread = omp_get_num_threads();

      instanceSM[ind] = new simuPo(lamda,ind);
      par[ind]        = instanceSM[ind]->growPo();

      total    += par[ind];
      total_in += par[ind] * par[ind];
   }


   if(gettimeofday(&finish_t,NULL) != 0){
      cout << "simuo.cpp: gettime error for stoping " << endl;
      return -1;
   }

   for(ind = 0; ind < nbRep; ind++)
   {
      delete instanceSM[ind];
   }

   /*Calculating/
   averagep    = total/(double) nbRep ;
   average_in  = total_in/(double) nbRep;
   double imme = average_in - (averagep * averagep);
   scart       = sqrt(imme);
```

```
   cof_eff     = (2.5758 * scart)/sqrt(nbRep);

   sprintf(outFile, "out/resRep_om_%d", nbthread);

   fp  = fopen(outFile, "w");
   if(fp == NULL){
      cout << "simuo.cpp - e000: Can't open file\n" << endl;
      return -1;
   }

   fprintf(fp, "Parallel execution of %d threads:\n", nbthread);
   for(ind = 0; ind < nbRep; ind++)
   {
      fprintf(fp, "%d : %14.10f\n", ind, par[ind]);
   }

   fprintf(fp, "Nombre de replications: %d\n", nbRep);
   fprintf(fp, "Moyenne: %14.10f\n", averagep);
   fprintf(fp, "Ecart-type: %14.10f\n", scart);
   fprintf(fp, "Total carree: %14.10f\n", total_in);
   fprintf(fp, "Moyenne carree: %14.10f\n", average_in);
   fprintf(fp, "Intervalle de confidence: %14.10f\n", cof_eff);
   fprintf(fp, "Temps d'exectuions: %lf\n",
               ((finish_t.tv_sec + finish_t.tv_usec * 1e-6)
               - (start_t.tv_sec + start_t.tv_usec * 1e-6)));

   fclose(fp);

   return 0;
}
```

Code A. 13. Le code parallèle simuo.cpp de la simulation stochastique de croissance démographique utilisant l'OpenMP

```
#include <iostream>
#include <cstdio>
#include <cstdlib>
#include <sstream>
#include <cstring>
#include <cmath>
#include <limits>
#include <pthread.h>
#include <unistd.h>

#include "TGenMT.hpp"
#include "simuPo.hpp"

using namespace std;

#define NBTHREADS 4
#define SIZE_SIMU 35000

/* ----------------------------------------------------------------*/
/* A structure for a thread of replications                        */
/* ----------------------------------------------------------------*/
typedef struct {
   int idRep;
   int nbRep;
   int nbRes;
   int lamda;
}s_data;

double result_imme[SIZE_SIMU] = { 0. }; /*Save intermediates*/

/* ----------------------------------------------------------------*/
/* Run a simulation with N replications                            */
/* ----------------------------------------------------------------*/
```

```
void *runSimu(void * data)
{
   int      tid, ind, nbExp, ires, lambda, numRes;
   int      res_begin = 0, res_end = 0;
   FILE   * fp[SIZE_SIMU];
   s_data * data2 = (s_data *)data;

   tid    = data2->idRep;
   nbExp  = data2->nbRep;
   lambda = data2->lamda;
   numRes = data2->nbRes;

   simuPo * instanceSM[SIZE_SIMU] = { NULL };
   int nbegin = tid * nbExp, nbEnd = nbExp * (tid + 1);

   for(ind = nbegin; ind < nbEnd; ind++)
   {
      instanceSM[ind] = new simuPo(lambda,ind);
      result_imme[ind] = instanceSM[ind]->growPo();
   }

   if(numRes != 0 && tid == 0)
   {
      res_begin = nbExp * NBTHREADS;
      res_end   = res_begin + numRes;

      if(res_end <= SIZE_SIMU){
         for(ires = res_begin; ires < res_end; ires++)
         {
            instanceSM[ires] = new simuPo(lambda,ires);
            result_imme[ires] = instanceSM[ires]->growPo();}
      }
      else cout << "Size of a simulation is not enough" << endl;
   }

   for(ind = nbegin; ind < nbEnd + res_end; ind++)
   {
      delete instanceSM[ind];
   }
}

/* ----------------------Main function-------------------------*/

int main(int argc, char **argv)
{
   double         total = 0., total_in = 0., average, average_in;
   double         scart, cof_eff;
   int            nbRepT, ind, nbRep, lam, rc, nbres;
   pthread_t      threadID[NBTHREADS];
   s_data         data[NBTHREADS];
   FILE         * fp;
   char           command[100], outFile[100];
   struct timeval start_t, finish_t;

   /*Affinity thread*/
   pthread_attr_t attr;
   cpu_set_t cpus;
   pthread_attr_init(&attr);

   if(argc < 2 || argc > 3)
   {
      cout << "Number of args invalid: ./simup lambda nbRep" << endl;
      return -1;
   }

   lam    = atoi(argv[1]);
   nbRep  = atoi(argv[2]);
   nbRepT = nbRep / NBTHREADS;
```

```
   nbres  = nbRep % NBTHREADS;

   if(gettimeofday(&start_t,NULL) != 0){
      cout << "simuo.cpp: gettime error for starting " << endl;
      return -1;
   }

   for(ind = 0; ind < NBTHREADS; ind++)
   {
      data[ind].idRep = ind;
      data[ind].nbRep = nbRepT;
      data[ind].lamda = lam;
      data[ind].nbRes = nbres;

      CPU_ZERO(&cpus);
      CPU_SET(ind,&cpus);
      pthread_attr_setaffinity_np(&attr, sizeof(cpu_set_t), &cpus);
      rc = pthread_create(&threadID[ind], NULL, runSimu, &data[ind]);

      if(rc){
        cout << "simup.cpp - e001 : Unable to create thread,"
             << rc << endl;
        return -1;
      }
   }

   /*waiting thread is terminated*/
   for(ind = 0; ind < NBTHREADS; ind++){
      pthread_join(threadID[ind], NULL);
   }

   if(gettimeofday(&finish_t,NULL) != 0){
      cout << "simuo.cpp: gettime error for stoping " << endl;
      return -1;
   }

   /*Reduction*/
   for(ind = 0; ind < nbRep; ind++)
   {
      total    += result_imme[ind];
      total_in += result_imme[ind] * result_imme[ind];
   }

   average     = total/(double) nbRep;
   average_in  = total_in/(double) nbRep;
   double imme = average_in - (average * average);
   scart       = sqrt(imme);
   cof_eff     = (2.5758 * scart)/sqrt(nbRep);

   sprintf(outFile, "out/resRep_pth_%d", NBTHREADS);
   fp  = fopen(outFile, "w");
   if(fp == NULL){
      cout << "simup.cpp - e002 : Can't open file\n" << endl;
      return -1;
   }

   fprintf(fp, "Parallel execution of %d threads\n", NBTHREADS);
   for(ind = 0; ind < nbRep; ind++)
   {
      fprintf(fp, "%d : %14.10f\n", ind, result_imme[ind]);
   }

   fprintf(fp, "Nombre de replications: %d\n", nbRep);
   fprintf(fp, "Moyenne: %14.10f\n", average);
   fprintf(fp, "Total carree: %14.10f\n", total_in);
   fprintf(fp, "Moyenne carree: %14.10f\n", average_in);
   fprintf(fp, "Ecart-type: %14.10f\n", scart);
   fprintf(fp, "Intervalle de confidence: %14.10f\n", cof_eff);
```

```
   fprintf(fp, "Temps d'exectuions: %lf\n",
               ((finish_t.tv_sec + finish_t.tv_usec * 1e-6)
               - (start_t.tv_sec + start_t.tv_usec * 1e-6)));
   fclose(fp);

   return 0;
}
```

Code A. 14. Le code parallèle simup.cpp de la simulation stochastique de croissance démographique utilisant le Pthread

```
void runSimu(int nbCore, int nbRep, char * prog, char * inFile);
void runSimu_sup(int nbCore, int nbRep_out, int nbRep_new,
                                    char * prog, char * inFile);
```

Code A. 15. Le code runSim.h pour déterminer le nombre de réplications

```
/* ------------------------------------------------------------ */
/*   runSim.c: Runs a simulation in parallel,a supporting for   */
/*             determine number of replication of simulation    */
/* ------------------------------------------------------------ */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

#include "genMRIP.h"

/* ------------------------------------------------------------ */
/*   runSim: Runs a simulation in parallel                      */
/*   Input: number of cores, number of replications, inputfile  */
/*          name of program                                     */
/*   Output: None                                               */
/* ------------------------------------------------------------ */
void runSimu(int    nbCore, int    nbPU, int nbSock, int nbRep,
             char * prog,   char * inFile)
{
   int jj, chFork, aBloc, nBlocks, nbRes, nbFile = 0;
   char command[200];
   FILE *fp;

   /*Number of blocks with nbProc*/
   aBloc = nbCore * nbPU;
   nBlocks = (int)(nbRep / aBloc);
   nbRes = nbRep % aBloc;

   /*Create shell scripts for MRIP technic*/
   if(nbRes == 0) genSimu(nbCore,prog,inFile);
   else {
      genSimu(nbCore,prog,inFile);
      genRes(nbCore, nBlocks, nbRes, prog, inFile);

      sprintf(command, "./simuRes %d %d", nBlocks, nbRes);
      system(command);
   }

   /*Runs blocks of programs, each block runs programs in parallel*/
   while(jj < nBlocks)
   {
      sprintf(command, "./simuMRIP %d", jj);
      system(command);
      jj++;
   }
}
```

```
/* ------------------------------------------------------------ */
/*   runSim: Run new adding simulation in parallel              */
/*   Input: number of cores, number of replications, inputfile  */
/*          name of program                                     */
/*   Output: None                                               */
/* ------------------------------------------------------------ */
void runSimu_sup(int nbCore,    int    nbPU, int    nbSock, int nbRep_out,
                 int nbRep_new, char * prog, char * inFile)
{
   int jj, chFork, aBloc, nBlocks, nBloc_old;
   int nbRes, nbRep, nbRes_old, nbSup, nbFile;
   char command[200];
   int tmp;
   FILE *fp;

   /*Calculation existed*/
   aBloc = nbCore * nbPU;
   nBloc_old = (int)(nbRep_out/aBloc);
   nbRes_old = nbRep_out % aBloc;

   if(nbRes_old !=0)
   {
      sprintf(command, "find . -name 'simuRes_sup' -print0 | xargs -0 rm");
      system(command);

      genRes_sup(nbCore, nbRep_out, prog, inFile);

      sprintf(command, "./simuRes_sup");
      system(command);
      tmp = nBloc_old + 1;
   }
   else tmp = nBloc_old;

   /*Number of blocks with nbProc*/
   nBlocks =(int)(nbRep_new / aBloc);
   nbRes = nbRep_new % aBloc;

   /*Create shell scripts for MRIP technic*/
   if(nbRes != 0 || nBlocks == 0)
   {
      sprintf(command, "find . -name 'simuRes' -print0 | xargs -0 rm");
      system(command);

      genRes(nbCore, nBlocks, nbRes, prog, inFile);

      sprintf(command, "./simuRes");
      system(command);
   }

   /*Runs blocks of programs, each block runs programs in parallel*/
   while(jj < nBlocks)
   {
      sprintf(command, "./simuMRIP %d", jj);
      system(command);
      jj++;
   }
}
```

Code A. 16. Le code runSim.c pour déterminer le nombre de réplications

```
/* ------------------------------------------------------------ */
/*   calnum.c: Determine the number of replications for a simu  */
/*   Input : None                                               */
/*   Output: Average of individus in parallel                   */
/*   Created by DAO Van Toan - August 2016                      */
```

```
/* ------------------------------------------------------------ */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <math.h>
#include <time.h>

#include "data.h"
#include "genMRIP.h"
#include "runSim.h"

/*-----------------------------------------------------*/
int main (int argc, char **argv)
{
   double  total=0., total_in=0., val, average, average_in;
   double  scart, cof,cof_eff,imme;
   int     nbRep, nBlocks, nbRes, nbNew, nbIm, nbTemp, nbPU, nbSock, nbCPU;
   int     nbFile = 0, count = 0, check = 0, ii, jj, nbCore, nbRep_int;
   char    prog[50], inFile[50], command[100];
   char  * path_work="./working";
   char  * path_out="./out";
   char  * path_data="./data";
   FILE  * fp;
   struct  timeval start_t, finish_t;

   if (argc != 5){
      printf("calnum.c - e001: invalid of args: ./calnum"
             " program (nbCPU) nbRep inputfile\n");
      return -1;
   }

   strcpy(prog,argv[1]);
   nbCPU  = atoi(argv[2]);
   nbRep  = atoi(argv[3]);
   strcpy(inFile,argv[4]);


   /*Get number of processor per core*/
   fp = popen ("hwloc-calc --number-of pu socket:0.core:0", "r");
   if (fp!=NULL)
   {
      fscanf(fp, "%d", &nbPU);
      pclose(fp);
   }else {
      printf("rspr.c - e002: Can't open file!\n");
      exit(1);
   }

   /*Get number of sockets in machine*/
   fp = popen ("hwloc-calc --number-of socket all", "r");
   if (fp!=NULL)
   {
       fscanf(fp, "%d", &nbSock);
       pclose(fp);
   } else {
      printf("rspr.c - e003: Can't open file!\n");
      exit(1);
   }

   nbCore = nbCPU / nbPU;
   int call = nbCPU % nbPU;
   if(nbCore == 0 || call != 0){
      printf("calnum -e004: invalid of args: Nb of CPU should be Nx%d\n",nbPU);
      return -1;
   }

   if(check_folder(path_data) == 0){
```

```
      create_folder(path_data);
      genMT_new(nbRep);
   }else if(check_folder(path_data) == 1
      && nbRep > count_file(path_data)
      && nbRep < 100000){
      genMT_res(count_file(path_data), nbRep);
   }

   if(check_folder(path_work) == 0){
      create_folder(path_work);
   }

   if(check_folder(path_out) == 0){
      create_folder(path_out);
   }

   /*Delete old datas*/
   remove_file(path_work);

   /*begin of time calculating*/
   if(gettimeofday(&start_t,NULL) != 0){
      printf("calnum -e005: gettime error for starting\n");
      return -1;
   }

   nbRep_int = nbRep;

   /*Runs blocks of programs */
   runSimu(nbCore, nbPU, nbSock, nbRep, prog,inFile);

   /*Reduction of results*/
   for(ii = 0; ii < nbRep; ii++)
   {
     sprintf(command, "working/resRep_%06d", ii);
     val = 0.;

     fp  = fopen(command,"r");
     if(fp == NULL){
         printf("calnum.c - e006: Can't open file to read output\n");
         return -1;
      }

     fscanf(fp,"%lf",&val);

     total    += val;
     total_in += val * val;

     fclose(fp);
   }

   average    = total/(double) nbRep;
   average_in = total_in/(double) nbRep;
   imme       = average_in - (average * average);
   scart      = sqrt(imme);

   nbTemp     = (int)(((2.764 * 100 * scart)/average)
                *
                  ((2.764 * 100 * scart)/average));

   nbRep      = nbTemp - nbRep;

   printf("Temporaire value : %d new Rep value: %d \n", nbTemp, nbRep);

   while(check == 0)
   {
      total = 0.; total_in = 0.; nbFile = 0;

      if(check_folder(path_data) == 1 \
```

```
         && nbRep > count_file(path_data) && nbRep < 100000){
         genMT_res(count_file(path_data), nbRep);
      }

      if(nbRep > count_file(path_work)){
         runSimu_sup(nbCore, nbPU, nbSock, \
            count_file(path_work), nbRep, prog, inFile);
      }

      /*Reduction of results*/
      for(ii = 0; ii < nbRep; ii++)
      {
         sprintf(command, "working/resRep_%06d", ii);
         val = 0.;

         fp = fopen(command,"r");
            if(fp == NULL){
            printf("calnum.c - e007: Can't open file to read output\n");
            return -1;
         }

         fscanf(fp,"%lf",&val);

         total    += val;
         total_in += val * val;

         fclose(fp);
      }

      average    = total/(double) nbRep;
      average_in = total_in/(double) nbRep;
      imme       = average_in - (average * average);
      scart      = sqrt(imme);
      cof_eff    = (2.5758 * scart)/sqrt(nbTemp);
      cof        = average/100.;

      if(cof_eff > cof){
         nbTemp = (int)(((2.5758 * scart)/cof) * ((2.5758 * scart)/cof));
         nbRep  = nbTemp - nbRep;
         count++;
      }
      else{
         check = 1;
         if(count == 0)
         {
            nbTemp = (int)(((2.5758 * scart)/cof) * ((2.5758 * scart)/cof));
            nbRep  = nbRep - nbTemp;
         }
      }
   }

   /*Final calculation*/
   if(check_folder(path_data) == 1 && \
      nbTemp > count_file(path_data) && nbTemp < 100000){
      genMT_res(count_file(path_data), nbTemp);
   }

   if(nbTemp > count_file(path_work)){
      runSimu_sup(nbCore, nbPU, nbSock, \
             count_file(path_work), nbTemp, prog, inFile);
   }

   total = 0.; total_in = 0.;

   for(ii = 0; ii < nbTemp; ii++)
   {
      sprintf(command, "working/resRep_%06d", ii);
      val = 0.;
```

```
      fp = fopen(command,"r");
      if(fp == NULL){
         printf("calnum.c - e008: Can't open file to read output\n");
         return -1;
      }

      fscanf(fp,"%lf",&val);

      total    += val;
      total_in += val * val;

      fclose(fp);
   }

   average    = total/(double) nbTemp;
   average_in = total_in/(double) nbTemp;
   imme       = average_in - (average * average);
   scart      = sqrt(imme);
   cof_eff    = (2.5758 * scart)/sqrt(nbTemp);
   cof        = average / 100.;
   int nbRec  = (int)(((2.5758 * scart)/cof)*((2.5758 * scart)/cof));

   time(&stop);

   sprintf(command, "out/resFinal_%d", nbTemp);
   fp  = fopen(command, "w");
   if(fp == NULL){
      printf("calnum.c - e009: Can't open file to write final result\n");
      return -1;
   }

  fprintf(fp, "Nombre de replication detecte: %d\n", nbTemp);
  fprintf(fp, "Moyenne: %14.10f\n", average);
  fprintf(fp, "Ecart type: %14.10f\n", scart);
  fprintf(fp, "Interval de confidence calcule: %14.10f\n", cof_eff);
  fprintf(fp, "Interval de confidence est 1/100 de moyenne: %14.10f\n",cof);
  fprintf(fp, "Nombre de replication recalcule: %d\n", nbRec);
  fprintf(fp, "Nombre de coeurs utilise: %d\n", nbCore);
  fprintf(fp, "Nombre de replication intial: %d\n", nbRep_int);
  fprintf(fp, "Temps execution: %.0f\n", difftime(stop,start));
  fclose(fp);

  /*Delete old datas*/
  remove_file(path_work);

  return 0;

}
```

Code A. 17. Le code calnum.c pour déterminer le nombre de réplications

```
#pragma once

#include <string>

#define N 624
#define M 397

#define MATRIX_A 0x9908b0dfUL   /* constant vector a */
#define UPPER_MASK 0x80000000UL /* most significant w-r bits */
#define LOWER_MASK 0x7fffffffUL /* least significant r bits */

class TGenMT {
        public:
                TGenMT();
```

```
                TGenMT(char * filename);
                TGenMT(unsigned long s);
                TGenMT(unsigned long init_key[], int key_length);


                double mtRand();
                unsigned long genrand_int32();
                double genrand_real1();
                double genrand_real2();
                void restoreStatus(char * inFileName);
                void saveStatus(char * inFileName);


        private:
                void init_genrand(unsigned long s);
                void init_by_array(unsigned long init_key[], int key_length);

                unsigned long mt[N]; /* the array for the state vector  */
                int mti; /* mti==N+1 means mt[N] is not initialized */
};
```

Code A. 18. Le code TGenMT.hpp

```
#define N 624
#define M 397
#define MATRIX_A 0x9908b0dfUL   /* constant vector a */
#define UPPER_MASK 0x80000000UL /* most significant w-r bits */
#define LOWER_MASK 0x7fffffffUL /* least significant r bits */

#include <cstdio>
#include <cfloat>
#include <climits>
#include <iostream>

#include "TGenMT.hpp"


TGenMT::TGenMT() {
        mti=N+1;
        init_genrand(5489UL);
}

TGenMT::TGenMT(unsigned long s) {
        mti=N+1;
        init_genrand(s);
}

TGenMT::TGenMT(unsigned long init_key[], int key_length) {
        mti=N+1;
        init_by_array(init_key, key_length);
}

TGenMT::TGenMT(char * fileName)
{
   // printf("%s\n",fileName);
   restoreStatus( fileName );
}

/* ---------------------------------------------------------- */
/* saveStatus         Saves the MT status in a text File      */
/*                                                            */
/* Input: inFileName  The file name                  D. Hill */
/* ---------------------------------------------------------- */

void TGenMT::saveStatus(char * inFileName)
{
    FILE * fp  = fopen(inFileName, "w");
    int    i   = 0;
```

```
    if(fp == NULL){
      printf("Error: Can't open file to save\n");
      //exit(1);
    }

    fprintf(fp, "%d\n", mti);

    for(i = 0 ; i < N ; i++)
    {
       fprintf(fp, "%ld ", mt[i]);
    }

    fclose(fp);
}
/* ------------------------------------------------------------ */
/* restoreStatus     Restores the MT status from a text File   */
/*                                                              */
/* Input: inFileName  The file name                   D. Hill */
/* ------------------------------------------------------------ */

void TGenMT::restoreStatus(char * inFileName) {

    FILE * fp;
    int    i   = 0;

    fp  = fopen(inFileName, "r");

    if(fp == NULL){
      printf("Error: Can't open file to read\n");
      //exit(1);
    }

    fscanf(fp, "%d", &mti);

    for(i = 0 ; i < N ; i ++)
    {
       fscanf(fp, "%ld", &mt[i]);
    }

    fclose(fp);
}

/* ------------------------------------------------------------- */
/* mtRand     Fast generation of a double precision pseudo random */
/*            number with the Mersenne Twister algorithm         */
/*            Uniform on [0,1)-real-interval                     */
/*                                                               */
/* This version is embedding the code of genrand_int32 for speed  */
/* and it also avoids the local allocation of y - interesting     */
/* for large simulations when the generator is called very often. */
/* D. Hill - March 2016                                          */
/* ------------------------------------------------------------- */
double TGenMT::mtRand() {
    static unsigned long y;
    /* y is set static for speed - avoids reallocation at each call */
    static unsigned long mag01[2]={0x0UL, MATRIX_A};
    /* mag01[x] = x * MATRIX_A  for x=0,1 */

    if (mti >= N) { /* generate N words at one time */
        int kk;

        if (mti == N+1)   /* if init_genrand() has not been called, */
            init_genrand(5489UL); /* a default initial seed is used */

        for (kk = 0; kk < N-M; kk++) {
            y = (mt[kk]&UPPER_MASK)|(mt[kk+1]&LOWER_MASK);
            mt[kk] = mt[kk+M] ^ (y >> 1) ^ mag01[y & 0x1UL];
        }
```

```
        for (; kk < N-1; kk++) {
            y = (mt[kk]&UPPER_MASK)|(mt[kk+1]&LOWER_MASK);
            mt[kk] = mt[kk+(M-N)] ^ (y >> 1) ^ mag01[y & 0x1UL];
        }
  y = (mt[N-1]&UPPER_MASK)|(mt[0]&LOWER_MASK);
        mt[N-1] = mt[M-1] ^ (y >> 1) ^ mag01[y & 0x1UL];

        mti = 0;
    }

    y = mt[mti++];

    /* Tempering */
    y ^= (y >> 11);
    y ^= (y << 7) & 0x9d2c5680UL;
    y ^= (y << 15) & 0xefc60000UL;
    y ^= (y >> 18);

    return y *(1.0/4294967296.0);
    /* divided by 2^32 */
}

/* ---------------------------------------------------- */
/* init_genrand                                         */
/*                                                      */
/* Input: initializes mt[N] with a simple int seed - s */
/*                                                      */
/* This can be confusing since the MT status is huge   */
/* ---------------------------------------------------- */
void TGenMT::init_genrand(unsigned long s) {
    mt[0]= s & 0xffffffffUL;
    for (mti=1; mti<N; mti++) {
        mt[mti] =
            (1812433253UL * (mt[mti-1] ^ (mt[mti-1] >> 30)) + mti);
        /* See Knuth TAOCP Vol2. 3rd Ed. P.106 for multiplier. */
        /* In the previous versions, MSBs of the seed affect   */
        /* only MSBs of the array mt[].                        */
        /* 2002/01/09 modified by Makoto Matsumoto             */
        mt[mti] &= 0xffffffffUL;
        /* for >32 bit machines */
    }
}

/* ---------------------------------------------------- */
/* init_by_array     initializes MT by an array        */
/*                                                      */
/* Input:  init_key is the array for initializing keys */
/*         key_length is its length                     */
/*         slight changes for C++, 2004/2/26           */
/* ---------------------------------------------------- */
void TGenMT::init_by_array(unsigned long init_key[], int key_length) {
    int i, j, k;
    init_genrand(19650218UL);
    i=1; j=0;
    k = (N>key_length ? N : key_length);
    for (; k; k--) {
        mt[i] = (mt[i] ^ ((mt[i-1] ^ (mt[i-1] >> 30)) * 1664525UL))
          + init_key[j] + j; /* non linear */
        mt[i] &= 0xffffffffUL; /* for WORDSIZE > 32 machines */
        i++; j++;
        if (i>=N) { mt[0] = mt[N-1]; i=1; }
        if (j>=key_length) j=0;
    }
    for (k=N-1; k; k--) {
mt[i] = (mt[i] ^ ((mt[i-1] ^ (mt[i-1] >> 30)) * 1566083941UL))
          - i; /* non linear */
        mt[i] &= 0xffffffffUL; /* for WORDSIZE > 32 machines */
        i++;
```

```
        if (i>=N) { mt[0] = mt[N-1]; i=1; }
    }

    mt[0] = 0x80000000UL; /* MSB is 1; assuring non-zero initial array */
}

/* ---------------------------------------------------------- */
/* genrand_int32        generates an integer random number    */
/*                                                            */
/* Output: a random number on the [0,0xffffffff]-interval     */
/* ---------------------------------------------------------- */
unsigned long TGenMT::genrand_int32() {
    unsigned long y;
    static unsigned long mag01[2]={0x0UL, MATRIX_A};
    /* mag01[x] = x * MATRIX_A  for x=0,1 */

    if (mti >= N) { /* generate N words at one time */
        int kk;

        if (mti == N+1)   /* if init_genrand() has not been called, */
            init_genrand(5489UL); /* a default initial seed is used */

        for (kk=0;kk<N-M;kk++) {
            y = (mt[kk]&UPPER_MASK)|(mt[kk+1]&LOWER_MASK);
            mt[kk] = mt[kk+M] ^ (y >> 1) ^ mag01[y & 0x1UL];
        }
        for (;kk<N-1;kk++) {
            y = (mt[kk]&UPPER_MASK)|(mt[kk+1]&LOWER_MASK);
            mt[kk] = mt[kk+(M-N)] ^ (y >> 1) ^ mag01[y & 0x1UL];
        }
        y = (mt[N-1]&UPPER_MASK)|(mt[0]&LOWER_MASK);
        mt[N-1] = mt[M-1] ^ (y >> 1) ^ mag01[y & 0x1UL];

        mti = 0;
    }

    y = mt[mti++];

    /* Tempering */
    y ^= (y >> 11);
    y ^= (y << 7) & 0x9d2c5680UL;
    y ^= (y << 15) & 0xefc60000UL;
    y ^= (y >> 18);

    return y;
}

/* ---------------------------------------------------------- */
/* genrand_real1   generates an random number in double precision */
/*                                                            */
/* Output: a random number on the [0,1]-real-interval         */
/* ---------------------------------------------------------- */
double TGenMT::genrand_real1() {
    return genrand_int32()*(1.0/4294967295.0);
    /* divided by 2^32-1 */
}

/* ---------------------------------------------------------- */
/* genrand_real2   generates an random number in double precision */
/*                                                            */
/* Output: a random number on the [0,1)-real-interval         */
/* ---------------------------------------------------------- */
double TGenMT::genrand_real2(void) {
    return genrand_int32()*(1.0/4294967296.0);
    /* divided by 2^32 */
}
```

Code A. 19. Le code TGenMT.cpp

```
#!/bin/bash

#number of physical cores from input:16
#number of sockets in machine:2
#number of cores per socket:8
#number of logical processors per core:2

listPids=""
t=$(expr $1 \* 2 \* 16)

idRep0=$(expr $t + 0)
hwloc-bind socket:0.core:0.pu:0 ./simur inFile.dat $idRep0 &
listPids="$listPids $!"

idRep1=$(expr $t + 1)
hwloc-bind socket:1.core:0.pu:0 ./simur inFile.dat $idRep1 &
listPids="$listPids $!"

idRep2=$(expr $t + 2)
hwloc-bind socket:0.core:1.pu:0 ./simur inFile.dat $idRep2 &
listPids="$listPids $!"

idRep3=$(expr $t + 3)
hwloc-bind socket:1.core:1.pu:0 ./simur inFile.dat $idRep3 &
listPids="$listPids $!"

idRep4=$(expr $t + 4)
hwloc-bind socket:0.core:2.pu:0 ./simur inFile.dat $idRep4 &
listPids="$listPids $!"

idRep5=$(expr $t + 5)
hwloc-bind socket:1.core:2.pu:0 ./simur inFile.dat $idRep5 &
listPids="$listPids $!"

idRep6=$(expr $t + 6)
hwloc-bind socket:0.core:3.pu:0 ./simur inFile.dat $idRep6 &
listPids="$listPids $!"

idRep7=$(expr $t + 7)
hwloc-bind socket:1.core:3.pu:0 ./simur inFile.dat $idRep7 &
listPids="$listPids $!"

idRep8=$(expr $t + 8)
hwloc-bind socket:0.core:4.pu:0 ./simur inFile.dat $idRep8 &
listPids="$listPids $!"

idRep9=$(expr $t + 9)
hwloc-bind socket:1.core:4.pu:0 ./simur inFile.dat $idRep9 &
listPids="$listPids $!"

idRep10=$(expr $t + 10)
hwloc-bind socket:0.core:5.pu:0 ./simur inFile.dat $idRep10 &
listPids="$listPids $!"

idRep11=$(expr $t + 11)
hwloc-bind socket:1.core:5.pu:0 ./simur inFile.dat $idRep11 &
listPids="$listPids $!"

idRep12=$(expr $t + 12)
hwloc-bind socket:0.core:6.pu:0 ./simur inFile.dat $idRep12 &
listPids="$listPids $!"
idRep13=$(expr $t + 13)
hwloc-bind socket:1.core:6.pu:0 ./simur inFile.dat $idRep13 &
listPids="$listPids $!"
```

```
idRep14=$(expr $t + 14)
hwloc-bind socket:0.core:7.pu:0 ./simur inFile.dat $idRep14 &
listPids="$listPids $!"

idRep15=$(expr $t + 15)
hwloc-bind socket:1.core:7.pu:0 ./simur inFile.dat $idRep15 &
listPids="$listPids $!"

idRep16=$(expr $t + 16)
hwloc-bind socket:0.core:0.pu:1 ./simur inFile.dat $idRep16 &
listPids="$listPids $!"

idRep17=$(expr $t + 17)
hwloc-bind socket:1.core:0.pu:1 ./simur inFile.dat $idRep17 &
listPids="$listPids $!"

idRep18=$(expr $t + 18)
hwloc-bind socket:0.core:1.pu:1 ./simur inFile.dat $idRep18 &
listPids="$listPids $!"

idRep19=$(expr $t + 19)
hwloc-bind socket:1.core:1.pu:1 ./simur inFile.dat $idRep19 &
listPids="$listPids $!"

idRep20=$(expr $t + 20)
hwloc-bind socket:0.core:2.pu:1 ./simur inFile.dat $idRep20 &
listPids="$listPids $!"

idRep21=$(expr $t + 21)
hwloc-bind socket:1.core:2.pu:1 ./simur inFile.dat $idRep21 &
listPids="$listPids $!"

idRep22=$(expr $t + 22)
hwloc-bind socket:0.core:3.pu:1 ./simur inFile.dat $idRep22 &
listPids="$listPids $!"

idRep23=$(expr $t + 23)
hwloc-bind socket:1.core:3.pu:1 ./simur inFile.dat $idRep23 &
listPids="$listPids $!"

idRep24=$(expr $t + 24)
hwloc-bind socket:0.core:4.pu:1 ./simur inFile.dat $idRep24 &
listPids="$listPids $!"

idRep25=$(expr $t + 25)
hwloc-bind socket:1.core:4.pu:1 ./simur inFile.dat $idRep25 &
listPids="$listPids $!"

idRep26=$(expr $t + 26)
hwloc-bind socket:0.core:5.pu:1 ./simur inFile.dat $idRep26 &
listPids="$listPids $!"

idRep27=$(expr $t + 27)
hwloc-bind socket:1.core:5.pu:1 ./simur inFile.dat $idRep27 &
listPids="$listPids $!"

idRep28=$(expr $t + 28)
hwloc-bind socket:0.core:6.pu:1 ./simur inFile.dat $idRep28 &
listPids="$listPids $!"

idRep29=$(expr $t + 29)
hwloc-bind socket:1.core:6.pu:1 ./simur inFile.dat $idRep29 &
listPids="$listPids $!"

idRep30=$(expr $t + 30)
hwloc-bind socket:0.core:7.pu:1 ./simur inFile.dat $idRep30 &
listPids="$listPids $!"
```

```
idRep31=$(expr $t + 31)
hwloc-bind socket:1.core:7.pu:1 ./simur inFile.dat $idRep31 &
listPids="$listPids $!"

wait $listPids
```

Script A. 1. Le script simuMRIP pour lancer plusieurs réplications en parallèle avec gestion des affinités (placement des processus sur tous les cœurs logiques d'une machine 32 cœurs)