Calcul à hautes performances : Reproductibilité et répétabilité des résultats numériques et des mesures de performances

Benjamin Antunes

UNIVERSITE CLERMONT AUVERGNE

THESE

présentée pour obtenir le grade de
Docteur en informatique
délivré par l'Université Clermont Auvergne

par

Benjamin Antunes

sous la direction de David R.C. Hill

Titre :

Calcul à hautes performances : Reproductibilité et répétabilité des résultats numériques et des mesures de performances

Date de la soutenance : 14 novembre 2024

JURY

Fatiha Bendali – PR - Président

Mamadou Traoré – PR - Rapporteur

Christophe Pouzat – CR HDR- Rapporteur

David R.C Hill – PR – Directeur de thèse

Eric Suraud – PR - Examinateur

Claude Mazel – MCF – Invité

Abstract

Reproducibility is widely recognized as a fundamental principle of scientific research. Currently, the scientific community faces numerous challenges related to reproducibility, often referred to as the "reproducibility crisis." This crisis has affected many scientific disciplines. In this thesis, we examined the factors in scientific practices that could contribute to this lack of reproducibility. Particular attention is given to the pervasive integration of computing in research, which sometimes functions as a black box. This thesis primarily focuses on high-performance computing (HPC), which presents unique reproducibility challenges. We provide a comprehensive state-of-the-art review of these concerns and potential solutions. Additionally, we discuss the crucial role of reproducible research in advancing science and identifying persistent issues in the field of HPC. We discuss the multiple reasons that can lead to a loss of reproducibility when using computing tools present in many areas of research. These include the importance of open science, rigorous documentation, correct application of statistics, scientific culture, software environments, workflows, and software engineering. We also delve into the issues of high-performance scientific computing, where new factors can impact reproducibility. These include problems related to parallel computing, Monte Carlo simulations using pseudo-random number generators, optimization processes, hardware heterogeneity, so-called "silent" errors, and the challenges posed by new paradigms such as quantum computing. We proposed various case studies on reproducibility in high-performance computing and provided insights and recommendations for researchers. We developed an epidemiological model in C++ to easily parallelize large scale individual based simulations of virus spreading, testing parameters were set for Covid 19 and can be changed to tackle other outbreaks like mpox, or other deases that may arise in the future. We studied the statistical quality of stochastic streams based on the initialization of Mersenne Twister, thus providing information on its parallel initialization. We also investigated various reproducibility issues related to the use of pseudo-random number generators in machine learning technologies. During this thesis, we also examined a hardware aspect by questioning the relevance of the systematic activation of simultaneous multi-threading on computing clusters for performance depending on the application profile. Finally, we studied the reliability of quantum machines, a computing paradigm becoming more reliable with the potential to revolutionize high-performance computing in specific areas.

Résumé

La reproductibilité est largement reconnue comme un principe fondamental de la recherche scientifique. Actuellement, la communauté scientifique se heurte à de nombreux défis liés à la reproductibilité, souvent désignés comme la « crise de la reproductibilité ». Cette crise a touché de nombreuses disciplines scientifiques. Dans cette thèse, nous avons examiné les facteurs dans les pratiques scientifiques qui pourraient contribuer à ce manque de reproductibilité. Une attention particulière est portée sur l'intégration omniprésente de l'informatique dans la recherche, qui fonctionne parfois comme une boîte noire. Cette thèse se concentre principalement sur le calcul haute performance (HPC), qui présente des défis uniques en matière de reproductibilité. Nous fournissons un état de l'art complet de ces préoccupations et des solutions potentielles. De plus, nous discutons du rôle crucial de la recherche reproductible dans l'avancement de la science et de l'identification des problèmes persistants dans le domaine du HPC. Nous discutions des multiples raisons pouvant conduire à une perte de reproductibilité lors de l'utilisation d'outils informatique présents dans bien des domaines de la recherche. Nous pouvons citer l'importance de la science ouverte, d'une documentation rigoureuse, de l'application correcte des statistiques, de la culture scientifique, des environnements logiciels, des workflows, ainsi que du génie logiciel. Nous voyons également en détail les problématiques du calcul scientifique à hautes performances, où de nouveaux facteurs peuvent impacter la reproductibilité. Nous pouvons citer les problèmes liés au calcul parallèle, aux simulations de Monte-Carlo utilisant des générateurs de nombres pseudo-aléatoires, aux processus d'optimisation, à l'hétérogénéité du matériel, aux erreurs dites « silencieuses », ainsi qu'aux défis posés par de nouveaux paradigmes tels que l'informatique quantique. Nous avons proposé différents cas concret d'étude de reproductibilité en calcul haute performance, et avons apporté des connaissances et recommandations pour les chercheurs. Nous avons développé un modèle épidémiologique en C++, afin de pouvoir facilement parallèliser des simulations de propagation de individus cenrtrés à large échelle. Les tests ont étés réalisés avec des paramètres pour le Covid 19 mais peuvent être adaptés à d'autres types d'épidémie (M-pox par exemple). Nous avons étudié la qualité statistique des flux stochastiques selon l'initialisation de Mersenne Twister, et ainsi fournit des informations sur son initialisation parallèle. Nous avons étudié les différents problèmes de reproductiblité lié à l'utilisation des générateurs de nombres pseudo-aléatoires dans les technologies du machine learning. Lors de cette thèse nous avons également étudié l'aspect matériel, en questionnant la pertinence de l'activation systématique du simultaneous multi-threading sur les clusters de calcul pour les performances en fonction du

type de calcul. Enfin, nous avons étudié la fiabilité des machines quantiques, un paradigme de calcul qui devient plus fiable et qui a le potentiel de révolutionner le calcul haute performance sur certaines thématiques spécifiques.

Remerciements :

Table des matières

Table des figures

Tables des tables

Introduction générale

Cette thèse s'inscrit dans la démarche grandissante de la recherche reproductible. En effet, de nombreuses disciplines, durant la dernière décennie, se sont inquiétées de plus en plus de la faible capacité à reproduire les résultats publiés. Dans notre travail, nous nous concentrons plus spécifiquement sur le domaine de l'informatique et du calcul intensif. L'informatique étant désormais omniprésente dans de nombreux autres domaines de recherche, si elle a apporté de nombreuses solutions et de nombreux bénéfices, elle a aussi amené son lot de problèmes, dont notamment des soucis de reproducbilité scientifique. Durant cette thèse, nous proposons un état de l'art complet des problémes de reproductibilité lié a l'informatique et au calcul intensif. Ensuite, nous étudions de manière précise certains détails, afin de proposer des solutions aux chercheurs. Tout cela s'inscrit dans le contexte de la crise de la reproductiblité en Science et nous espérons que cette thèse apportera une contribution utile à la communauté.

La crise de la reproductibilité en Science est désormais un phénomène mondial et largement transdisciplinaire qui contribue à la méfiance de la société envers le monde de la recherche. En 2016, Baker (2016) a publié dans la revue Nature une enquête auprès de 1576 scientifiques pour connaître leur opinion sur le fait que nous sommes actuellement confrontés à une crise de reproductibilité. Au total, 90 % des répondants pensent qu'il existe une crise de reproductibilité significative ou légère. Seuls 3 % sont convaincus qu'il n'y a pas de crise du tout. Cette étude souligne le consensus au sein de la communauté scientifique qu'une crise de reproductibilité est bien présente, couvrant un large spectre de disciplines.

La littérature existante offre une myriade d'exemples, tant théoriques qu'empiriques, qui mettent en lumière la crise de la reproductibilité. En médecine, nous avons le célèbre titre provocateur de Ioannidis, un éminent épidémiologiste. L'un de ses articles majeurs dans ce domaine était intitulé : « Pourquoi la plupart des résultats de recherche publiés sont faux » (Ioannidis 2005). Dans cet article très cité, il discute des failles statistiques qui pourraient affecter les résultats publiés. Errington *et al.* (2021) présentent les résultats de reproductibilité d'études sur le cancer, montrant un taux de succès de seulement 46 % sur 112 tentatives. Begley et Ellis (2012) ont eux aussi lancé une alerte sur la reproductibilité dans la recherche sur le cancer. Ce taux de 46% peut sembler relativement optimiste si l'on considère (Ioannidis 2015), qui affirme que 85 % des financements de la recherche sont gaspillés. Eklund *et al.* (2016) ont trouvé un problème significatif avec les études concernant l'imagerie par résonance magnétique (IRM), indiquant que de nombreux articles pourraient avoir rapporté de faux résultats. L'Open Science Collaboration (2015) a révélé un problème similaire dans les articles de psychologie,

avec des taux de reproductibilité allant de 30 % à 50 %. Dans le domaine des neurosciences, Topalidou *et al.* (2015) n'ont pas pu reproduire un modèle et ont dû passer trois mois à le ré-implémenter.

Nous avons vu que le domaine médical est fortement impacté par cette crise de la reproductibilité, y compris des domaines tels que le domaine pharmaceutique, les dispositifs médicaux (y compris l'IRM), la psychologie, la recherche sur le cancer et les neurosciences. Cependant, comme mentionné précédemment, aucun domaine n'est exempt de ce problème. Grâce à ses machines déterministes, nous aurions pu considérer l'informatique comme une science exacte et ne faisant pas face à ce problème. Néanmoins, l'informatique n'est non seulement pas à l'abri de la crise de reproductibilité; mais en réalité, elle fait partie des causes significatives de la crise. Une étude approfondie de Collberg et Proebsting (2016) sur la reproductibilité des articles en informatique a donné des résultats plutôt mauvais, avec seulement environ 30 % des articles de recherche reproductibles sur 601 très bons articles examinés (publiés par l'Association for Computing Machinery (ACM)). Manninen *et al.* (2017) ont tenté de reproduire quatre modèles de « L'excitabilité calcique dans les astrocytes », et 3 modèles sur 4 manquaient d'informations essentielles, et 2 modèles sur 4 avaient des équations incorrectes. Même après avoir corrigé les différents modèles, ils n'ont pas produit des résultats cohérents entre eux. Mesnard et Barba (2017) ont travaillé sur la mécanique des fluides, et il leur a fallu trois ans pour reproduire les résultats obtenus à partir de différentes versions leurs propres codes et outils. L'intelligence artificielle n'est également pas exempte de ce problème, comme l'ont démontré Gundersen et Kjensmo (2018). Dans le domaine des réseaux, Kurkowski *et al.* (2005) ont trouvé que moins de 15 % des articles sur les simulations de réseaux MANET étaient reproductibles. En traitement d'image, Kovacevic (2007) a étudié 15 articles publiés dans son domaine, et a constaté qu'aucun algorithme présenté n'était soutenu par un code, et que seulement 33 % des données étaient disponibles. Vandewalle *et al.* (2009) ont examiné 134 articles également dans le domaine du traitement d'image, constatant que 9 % des articles avaient un code disponible, et 33 % pour les données.

Cela souligne que l'informatique et ses différents sous-domaines ne sont pas à l'abri des défis de la reproductibilité. Ces points montrent l'importance d'établir une recherche reproductible en informatique comme dans tous les dommaines. Mettre l'accent sur la nécessité d'une recherche reproductible peut prévenir les fraudes et les scandales potentiels. Par exemple, Reinhart & Rogoff, spécialistes mondiaux de l'économie, ont affirmé en 2010 qu'augmenter la dette d'un pays au-delà de 90 % du Produit Intérieur Brut (PIB) d'un pays arrêterait sa croissance

économique. Cette affirmation a conduit de nombreux pays occidentaux à adopter des politiques d'austérité. Cependant, Herndon *et al.* (2014) ont ensuite prouvé que l'étude était erronée. Reinhart et Rogoff avaient exclu des données qui contredisaient leurs conclusions et commis des erreurs de calcul. Puisqu'ils ont partagé le code et les données (un fichier Excel), il a été possible de vérifier leur travail. Cela plaide fortement en faveur d'une science ouverte (partage de toutes les données liées à un article) afin de donner une chance à la recherche reproductible. La seule façon de démontrer qu'un article est faux est d'avoir accès à ses artefacts. Un autre exemple décrit dans (Miller 2006) concerne Geoffrey Chang, qui a eu un impact significatif sur le domaine de la structure protéique des bactéries résistantes aux antibiotiques. Cependant, plusieurs années plus tard, ses résultats se sont avérés faux en raison de la découverte d'une erreur de programmation dans les outils internes de Chang.

Récemment, la crise du Covid19 a fortement augmenté la sensibilisation des citoyens non scientifiques à l'importance de la recherche reproductible, sans que ces derniers ne le réalisent vraiment. Nous avons été témoins de scandales dans le contexte du Covid-19, tels que la rétractation de deux articles du Lancet et du New England Journal of Medicine (Piller, Servick 2020). Les deux études ont influencé la politique internationale sur l'utilisation, ou non, de certains médicaments dans la cadre de la pandémie et ont dû être rapidement rétractés. Un autre cas était le modèle Covid de Neil Ferguson (Ferguson *et al.* 2020), pour lequel Pouzat (2022) a écrit un article humoristique, soulignant le scandale causé par la non-publication du code initial de Ferguson. Ce modèle avait déjà influencé les politiques internationales sur les mesures de confinement, en particulier en Angleterre. Sous la pression internationale, principalement des États-Unis, Neil Ferguson a publié une version fortement révisée de son code et ce dernier a ensuite été rapporté comme gravement défectueux (Zolfagharifard et Boland 2020) menant à une pétition sur GitHub (Holmes 2020) pour retirer le code afin d'éviter son utilisation comme base pour d'autres modèles épidémiologiques. La pression financière, en particulier en médecine à l'échelle mondiale (Abbasi 2020), peut fortement diminuer la qualité scientifique, car sans reproductibilité, les chercheurs d'entreprises ou de laboratoires publics peuvent produire de la pseudoscience selon les critères énoncés par Karl Popper. Le financement et les conflits d'intérêts gangrènent ces situations, dans un cadre où nous avions besoin d'une collaboration publique internationale rapide. Iqbal *et al.* (2016) ont déclaré que « Les articles publiés dans des revues dans la catégorie de la médecine clinique, par rapport à d'autres domaines, étaient presque deux fois plus susceptibles de ne pas inclure d'informations sur le financement et d'avoir en fait un financement privé ». De plus, il est reconnu que les articles

avec un financement industriel ou des auteurs industriels sont moins susceptibles de partager des codes et des données (Collberg *et al.* 2015). Ceci conduit au fait que nous ne pouvons pas accorder toute notre confiance aux articles en provenance de l'industrie si nous n'avons pas accès aux artefacts des publications proposées. Une initiative Française du Ministère des Solidarités et de la Santé est à remarquer, la mise en ligne des informations concernant les conventions, les rémunérations et les avantages liant des entreprises et des acteurs du secteur de la santé. Ce dispositif a le mérite d'engager les personnels de santé à plus de clarté.

Dans ce contexte, les objectifs de cette thèse sont multiples. Tout d'abord, nous voulons étudier la recherche reproductible dans son ensemble, puis, nous intéresser en particulier au monde de l'informatique et du calcul à haute performance. Il existe un très large panel de raison pour lesquelles la reproductibilité peut être perdue. De nombreux articles de recherche ont été publiés durant la dernière décénnies sur ce sujet. Dans cette thèse nous cherchons à rassembler l'ensemble des connaissances relatives à la recherche reproductible, à lister et détailler les problèmes qui se présentent aux chercheurs, mais aussi à présenter les solutions possibles, cela ayant pour but de réellement contribuer à la prise en main de ce sujet par les chercheurs voulant s'y interesser. Etant donné les directions prises par les journaux et les conférences les plus sérieuses pour alers vers une démarche de recherche reproductible, ce travail devrait pouvoir bénéficier à un grand nombre de chercheurs en informatique. Ensuite, un des objectifs de la thèse est de contribuer à l'élargissement des connaissances sur certains points précis du calcul intensif, en lien avec la reproductibilité, tel que la bonne utilisation des flux stochastiques via les générateurs de nombres pseudo-aléatoires. Nous testerons quelques applications, et notamment le développement d'un modèle épidémiologique. La lecture de cette thèse devrait aider à une vraie prise en main de la recherche reproductible en informatique, à en mesurer les enjeux… Nous essaierons également d'apporter des connaissances plus techniques, spécifiques au calcul intensif, via certaines expériences que nous avons ménées.

Dans un 1er chapitre, nous proposons un état de l'art complet sur la reproductibilité en informatique et son impact sur le calcul scientifique. Nous détaillons le contexte général ainsi que l'évolution des définitions en recherche reproductible. Puis nous listons les problèmes liés à l'utilisation de l'informatique en science, tel que l'environnement logiciel, les workflows, la documentation, le partage du code et des données (science ouverte), et enfin nous présentons les problèmes spécifiquement liés au calcul intensif, comme l'execution de programme parallèle, l'hétérogénéité du matériel, l'informatique quantique ou encore les erreurs silencieuses.

Dans le second chapitre, nous décrivons en détail les solutions existantes pour les problèmes listés précédemment (comme par exemple les outils permettant de gérer les workflows scientifiques, les environnemens logiciels, etc.).

Dans le troisième chapitre, nous exposons nos différentes propositions afin de contribuer aux connaissances concernant la reproductibilité, numérique et temps de calcul, dans le domaine du calcul intensif.

Le dernier chapitre présente les expériences numériques que nous avons mis en place afin d'illustrer les propositions citées précédemment. Nous décrivons l'importance de la bonne utilisation des PRNGs dans les frameworks du machine learning, puis nous parlons des différentes méthodes de parallélisations de nombres pseudo-aléatoires. Nous proposons ensuite un outil pour faire des simulations épidémiologiques stochastiques répétables et reproductibles. Puis, nous discutons du matériel et des bibliothèques scientifiques pour la reproductibilité des performances dans le cadre de l'utilisation de multiples threads, mais aussi dans un cadre de simulation pour la Physique (étude de la dynamique des systèmes quantiques à plusieurs corps loin de l'équilibre). Le code final de cette dernière application tourne sur le supercalculateur Olympe (CALMIP – Toulouse). Enfin, nous évoquons l'arrivée du nouveau paradigme de calcul qu'est l'informatique quantique et nous testons les limites de cette technologie sur un algorithme de référence (Grover).

I. La crise de la reproductibilité et les différents problèmes liés à l'informatique et au calcul à haute performance

I.1. Introduction

La reproductibilité est reconnue comme l'un des piliers de la Science. En 1660, la Royal Society adopta la devise « Nullius in verba », qui se traduit par « ne croire personne sur parole » (Royal Society 1660). Les philosophes des sciences s'accordent à dire que la reproductibilité est l'un des critères permettant de différencier la science de la pseudoscience. Depuis les années 2010, on constate une augmentation significative de l'intérêt international des scientifiques pour la recherche reproductible. Fanelli (2018) a montré une augmentation exponentielle du nombre de publications faisant référence au thème de la crise de la reproductibilité. Nous observons un nombre croissant de revues et de conférences préoccupées par la reproductibilité de leurs articles publiés (Drummond 2018; Bajpai, Bonaventure, *et al.* 2019). Nous pouvons également remarquer la création de journaux dédiés à la reproduction d'articles (Rougier *et al.* 2017). Le domaine de la recherche reproductible est très actif et en rapide évolution. Nous avons trouvé une étude fournissant un état de l'art de la reproductibilité dans le calcul scientifique (Ivie et Thain 2018), et plusieurs livres tentant de le faire (Desquilbet *et al.* 2019; National Academies of Sciences 2019; Randall et Welser 2018). Sans minimiser la qualité des travaux précédents, nous pensons que, comme cette thématique évolue, les définitions mêmes ont évolué encore en 2020, il est donc pertinent de mettre à jour nos connaissances et de fournir un état de l'art actualisé sur les définitions et les technologies utilisées dans la recherche reproductible car elles ont évolué récemment. Nous voulons offrir un point de vue supplémentaire, axé sur le calcul à hautes performances (que nous noterons HPC pour « High Performance Computing » en anglais). Alors que les travaux précédemment cités mettent l'accent sur la reproductibilité computationnelle et sont plus axés sur la méthode scientifique globale ou sur les workflows comme Ivie et Thain (2018), avec le HPC, nous sommes en première ligne, là où les problèmes de reproductibilité spécifiques sont le plus susceptible d'apparaître. Nous avons publié un livre en français pour couvrir l'état de l'art de la recherche reproductible (Antunes et Hill 2024).

I.2. Définitions de la recherche reproductible

I.2.1. Evolution des terminologies

Trois termes principaux sont employés dans le domaine de la recherche reproductible : Reproductibilité, réplicabilité et répétabilité. Bien que définir la reproductibilité semble quelque chose d'évident, la réalité est qu'une définition consensuelle parmi les chercheurs et les différents domaines de recherche n'a été atteinte que très récemment.

Voici un peu d'histoire récente 2020, « l'Association for Computing Machinery » (ACM) définissait (anciennes définitions) ces termes comme suit :

- Répétabilité : même équipe, même protocole expérimental.
- Reproductibilité : équipe différente, protocole expérimental différent.
- Réplicabilité : équipe différente, même protocole expérimental.

Dans ces définitions, comme l'a déclaré Drummond (2009), « Reproducibility requires change, replicability avoids it », qui peut se traduire par « la reproductibilité exige des changements, la réplicabilité les évite ». La reproductibilité implique qu'une équipe différente, appliquant une méthode ou une expérience différente pour la même question scientifique, obtienne les mêmes conclusions scientifiques, renforçant ainsi la découverte. D'autre part, la réplicabilité vise à ce qu'une équipe différente atteigne les mêmes résultats avec une certaine précision, en utilisant les artefacts de l'article de la première équipe. Dans la littérature, les auteurs utilisaient parfois le mot « reproductibilité » pour se référer à la « réplicabilité », et vice versa. Conseillée par la NISO (National Information Standards Organization), l'ACM a changé ses définitions après 2020, échangeant les définitions entre reproductibilité et réplicabilité. La principale raison en était le besoin d'une meilleure correspondance avec la pratique des autres domaines de recherche.

Le fait que l'informatique était l'un des seuls domaines à utiliser ces définitions était un argument pour suivre les conseils de standardisation de la NISO; l'ACM a donc changé ses définitions de reproductibilité et de réplicabilité. Voici les définitions de 2020, et actuelles, pour les deux termes :

- *Reproductibilité (Équipe différente, même expérience) : la mesure peut être obtenue avec la précision déclarée par une équipe différente utilisant la même procédure de mesure, le même système de mesure, dans les mêmes conditions de fonctionnement, au même endroit ou dans un endroit différent lors de plusieurs essais. Pour les expériences*

informatiques, cela signifie qu'un groupe indépendant peut obtenir le même résultat en utilisant les propres artefacts de l'auteur.

- *Réplicabilité (Équipe différente, expérience différente) : la mesure peut être obtenue avec la précision déclarée par une équipe différente, un système de mesure différent, dans un endroit différent lors de plusieurs essais. Pour les expériences informatiques, cela signifie qu'un groupe indépendant peut obtenir le même résultat en utilisant des artefacts qu'ils développent complètement indépendamment.*
- *Répétabilité (Même équipe, même expérience) : la mesure peut être obtenue avec la précision déclarée par la même équipe en utilisant la même procédure de mesure, le même système de mesure, dans les mêmes conditions de fonctionnement, au même endroit lors de plusieurs essais. Pour les expériences informatiques, cela signifie qu'un chercheur peut reproduire de manière fiable son propre calcul.*

Pour illustrer ce problème de non-consensus des définitions, nous pouvons examiner la littérature. Nous nommons D1 le jeu de définitions proposées par l'ACM avant 2020, et D2 le jeu de définitions adoptées par l'ACM après 2020. La plupart des articles ne reprennent pas l'ensemble des trois termes : reproductibilité, répétabilité et réplicabilité.

Dans un article discutant de la reproductibilité en informatique (Hinsen 2014), Konrad Hinsen se rapproche des définitions D1. Pour l'article décrivant la création de leur journal ReScience, Rougier *et al.* (2017) utilisent des définitions en phase avec le jeu D2. Dans une présentation en 2017 (Hinsen 2017), Konrad Hinsen modifie sa vision des termes présentée en 2014 pour passer au sens proposé par le jeu D2. Dans leur article, Stanisic *et al.* (2015) se rapporte aux définitions D1. L'article de Cohen-boulakia *et al.* (2017) sur le thème des sciences de la vie est dans la catégorie D1. Drummond (2009) plébiscite le jeu de définitions D1. Le papier étudiant la reproductibilité de Collberg et Proebsting (2016) applique les définitions D1. Gundersen et Kjensmo (2018) utilise le jeu D1, il en est de même pour Bajpai *et al.* (2019) dans le domaine des réseaux. La National Academies of Sciences Engineering, and Medicine (2019) utilise le jeu de définitions D2 avec un fort impact (plus de 600 citations à ce jour). L'état de l'art le plus récent sur le thème de la reproductibilité des workflows en calcul scientifique (Ivie et Thain 2018) utilise également le jeu de définitions D1. Stodden a utilisé en 2011 (Stodden 2011) le jeu D1, puis est passée au jeu D2 en 2014 (Stodden *et al.* 2014). Plusieurs articles utilisent les trois termes reproductibilité, répétabilité et réplicabilité, mais la plupart utilisent la notion de « recherche reproductible » sans apporter explicitement de nuances aux différentes notions. C'était le cas dans notre article de 2013 (Hill *et al.* 2013), nous utilisions sans

distinction le terme de reproductibilité au sens large de la philosophie des Sciences. Avec plus d'expérience dans le domaine, nous avons utilisé le jeu de définitions D1, dans (Hill 2015) et (Hill *et al.* 2017), où la reproductibilité fait référence à l'obtention de la même conclusion scientifique en réalisant potentiellement une autre expérience et la répétabilité le fait d'avoir les même résultats bits à bits pour pouvoir déboguer. Enfin dans (Hill 2022), nous utilisons le jeu de définitions D2 en conformité avec les avancées de l'ACM et de la « National Academies of Sciences, Engineering, and Medicine ».

Dans notre petit échantillon, qui comprend principalement des articles publiés par des auteurs actifs du domaine et bien cités, nous avons identifié huit articles qui utilisent les définitions fournies par l'ACM avant 2020 (D1) et qui ont été publiés entre 2009 et 2019. Quatre articles utilisent les nouvelles définitions de l'ACM (D2) dans un intervalle de temps similaire. De notre point de vue en tant qu'informaticiens, il semble que la majorité des travaux publiés que nous pouvons voir s'appuient sur les définitions obsolètes de l'ACM (avant 2020, D1). Une étude très intéressante de Barba (2018) a étudié quel champ scientifique utilisait plutôt une définition ou l'autre. Avec cette étude, nous réalisons que l'informatique était l'un des rares domaines à utiliser les définitions données au début de cette section (D1).

	Définitions ACM avant 2020 (anciennes)	Définitions ACM après 2020 (actuelles)
Reproductibilité	Equipe différente, expérience différente	Equipe différente, même expérience
Replicabilité	Equipe différente, même expérience	Equipe différente, expérience différente
Répétabilité	Même équipe, même expérience	Même équipe, même expérience
Classification des définitions utilisées par des auteurs du monde de la recherche reproductible dans leurs articles	(Hinsen 2014), (Stanisic *et al.* 2015), (Cohen-boulakia 2017), (Drummond 2009), (Collberg and Proebsting, 2016), (Gundersen and Kjensmo, 2018), (Bajpai, Brunstrom, *et al.* 2019), (Ivie and Thain, 2018), (Stodden 2011), (Hill *et al.* 2013), (Hill 2015), (Hill 2019), (Hill *et al.* 2017)	(Rougier *et al.* 2017), (Hinsen 2017), (National Academies of Sciences 2019), (Stodden *et al.* 2014), (Hill 2022)

Table 1: Tour d'horizons des définitions au sein de la recherche reproductible

Les informations précédentes sont résumées dans la Table 1. Néanmoins, pour continuer définir précisément un concept, il est impératif de considérer des perspectives au-delà de son

domaine immédiat (pour nous, l'informatique). Certaines définitions de la recherche reproductible sont spécifiques à chaque domaine, ce que nous voulons éviter dans le contexte de la reproductibilité, puisque nous voulons standardiser autant que possible entre les disciplines. Nous observons une tendance à adopter les nouvelles définitions parmi les auteurs contribuant activement au domaine de la recherche reproductible, tel que Konrad Hinsen ou Victoria Stodden, dans cet objectif de standardisation.

Les nouvelles définitions de l'ACM correspondent avec les définitions de l'Académie Nationale des Sciences, de l'Ingénierie et de la Médecine, qui définissent « La reproductibilité est l'obtention de résultats cohérents en utilisant les mêmes données d'entrée; étapes de calcul, méthodes et code; et conditions d'analyse », et « La réplicabilité est l'obtention de résultats cohérents à travers des études visant à répondre à la même question scientifique, chacune ayant obtenu ses propres données » (National Academies of Sciences 2019). Les articles édités avec le soutien de l'ACM peuvent obtenir des « badges », suivant le niveau de disponibilité des artefacts et de reproductibilité de l'article (Badges ACM).

Le dernier terme à discuter est la répétabilité. Il a suscité moins de controverse que les deux autres. Cependant, dans les articles de recherche en informatique, nous trouvons parfois une confusion entre répétabilité et reproductibilité. Nous discuterons de son importance plus tard.

Cependant, d'un point de vue philosophique, la reproductibilité est le seul terme qui peut englober toutes les différentes notions. Karl Popper, le célèbre philosophe des sciences, a discuté de la logique derrière la découverte scientifique au début du 20ème siècle, d'abord en allemand (1934/35), puis en anglais en 1959. Nous disposons maintenant d'une réédition récente (Popper 2005). Popper a défini la reproductibilité comme un critère permettant de distinguer la science et la pseudo-science (Hill 2019 ; Bisgambiglia et Hill 2022). L'ajout d'autres termes tel que réplicabilité ou répétabilité est toujours utile, car cela permet de décrire avec une granularité plus fine ce dont nous parlons dans nos articles. Plus centré sur le terme de reproductibilité, Goodman *et al.* (2016) ont proposé trois autres définitions qui pourraient convenir et englober la définition standard :

- La reproductibilité des méthodes : se réfère à la capacité de reproduire fidèlement les procédures expérimentales et computationnelles, en utilisant les mêmes données et outils, afin d'obtenir des résultats cohérents. Cela s'aligne avec la définition révisée de la reproductibilité de l'ACM, ou peut-être de la répétabilité.

- La reproductibilité des résultats : concerne la génération de résultats cohérents dans une nouvelle étude grâce à l'utilisation des mêmes méthodes expérimentales. Cela correspondrait également à la nouvelle définition de la reproductibilité de l'ACM. Cependant, dans la définition de la reproductibilité, nous utilisons exactement les mêmes matériaux (artefacts) pour générer les mêmes résultats.
- La reproductibilité inférentielle : implique d'atteindre des conclusions qualitativement similaires soit par une réplication indépendante d'une étude, soit par la ré-analyse des résultats de l'étude originale. Cela correspondrait à la nouvelle définition de la réplicabilité de l'ACM.

Avec ces définitions, nous conservons le terme principal de reproductibilité, mais en ajoutant un contexte à son utilisation. Cependant, nous considérons que les définitions de l'ACM post 2020 sont désormais la norme à laquelle les auteurs devraient se conformer, dans un objectif de standardisation. On constate néanmoins que de nombreuses définitions ont encore émergé pour essayer de définir ce qu'est la reproductibilité.

I.2.2. A propos de l'importance de la recherche reproductible

L'une des origines du mouvement de la recherche reproductible en science computationnelle a été l'initiative de Claerbout en 1992 (Claerbout et Karrenbach 1992). Buckheit et Donoho, en 1995, l'ont synthétisé avec la célèbre citation : « Un article en informatique dans une publication scientifique n'est pas en soi la recherche, c'est simplement une publicité pour la recherche. La véritable recherche est l'environnement complet de développement logiciel et l'ensemble complet des instructions qui ont généré les figures ». Marwick, en 2015, a souligné le défi posé par nos programmes informatiques : ils agissent comme des boîtes noires. Aujourd'hui, presque toute recherche en science implique l'utilisation d'un ordinateur. Dans un passé pas si lointain, il était nécessaire de faire soi-même tous les calculs, toutes les transformations, l'utilisation des données, etc., et décrire précisément la procédure dans un article de recherche que l'on allait publier. Cela permettait de reproduire les articles assez simplement. Aujourd'hui, des couches logicielles plus ou moins complexes cachent de nombreux éléments. Nous ne savons souvent pas exactement ce qui se passe, et il devient beaucoup plus compliqué de reproduire des résultats publiés par d'autres, si nous n'avons pas la même machine avec la même pile logicielle disponible. L'ordinateur devient un instrument de recherche à part entière, et bien plus encore... Par conséquent, il devrait subir le

même contrôle de qualité (un travail méticuleux de métrologie) que les biologistes ou physiciens appliquent à leurs instruments.

Comme le dit Popper, la reproductibilité est obligatoire pour l'avancement de la science. Nous devrions être capables d'utiliser les artefacts (code, données, etc.) d'un papier publié pour rejouer l'expérience et obtenir les mêmes résultats. Nous avons besoin des artefacts du papier de recherche pour prévenir les erreurs ou, beaucoup moins fréquemment, les fraudes. De plus, la reproductibilité implique qu'une équipe de recherche indépendante mène une expérience uniquement basée sur les informations fournies par l'équipe de recherche originale. La capacité à faciliter la reproduction d'un article que l'on publie augmentera la confiance dans les résultats publiés. Notamment, d'autres chercheurs peuvent maintenir un niveau d'objectivité plus élevé, car ils n'ont aucun intérêt à exagérer la performance d'une méthode qu'ils n'ont pas développée eux-mêmes. En outre, ces chercheurs peuvent ne pas partager les mêmes préconceptions et connaissances tacites de l'équipe initiale qui a effectué la recherche. De plus, les variations dans les configurations matérielles et logicielles parmi différents chercheurs aident également à contrôler les variables de bruit associées au matériel et aux logiciels auxiliaires, ainsi qu'aux connaissances implicites et préconceptions. Mais ce dernier point n'est vrai que si l'on considère que la reproductibilité est atteinte avec une précision imparfaite, et non avec des résultats identiques au bit près (dans le cadre de l'informatique), qui pourraient être l'objectif de certains auteurs et aussi une exigence pour le débogage.

Concernant la répétabilité, la définition peut encore varier parmi les domaines scientifiques. En effet, en informatique, nos machines sont conçues pour être déterministes (à l'exception des machines quantique, qui sont par « essence » stochastiques). En informatique, nous voulons obtenir des résultats identiques au bit près d'une exécution à l'autre lorsque nous sommes sur la même machine pour le même programme dans le même environnement logiciel avec les mêmes jeux de données. Ce point est considéré comme acquis par de nombreux scientifiques, et malheureusement ce n'est pas toujours le cas, en particulier lorsqu'il s'agit de calcul haute performance. Cependant, la répétabilité bit à bit reste essentielle pour le débogage et pour la confiance dans l'utilisation de nos ordinateurs déterministes. La répétabilité est une véritable préoccupation pour les chercheurs qui sont encore conscients du débogage car ils continuent à faire du développement informatique, mais cette quête peut être particulièrement difficile avec le calcul parallèle, à tel point qu'on la compare parfois à une quête du « Graal ». Assurer un débogage parallèle fiable nécessite une répétabilité avec des résultats identiques au bit près. Et en ce sens, nous ne sommes pas entièrement d'accord avec la définition de l'ACM

qui ajoute que les résultats sont identiques avec une marge de précision. Cela est dû au fait que la définition de la répétabilité de l'ACM provient du vocabulaire international de métrologie. À notre avis, cette définition est parfaitement correcte pour l'informatique quantique, mais pas pour l'informatique déterministe classique où nous avons vraiment besoin de résultats identiques au bit près pour déboguer correctement. Dans le cas de l'informatique déterministe classique, la définition est valide en précisant que la marge de précision est nulle afin de n'autoriser aucune différence.

Enfin, la réplicabilité est également obligatoire pour la science. En effet, l'idée que plus une hypothèse scientifique est répliquée dans le monde entier (avec différentes équipes de recherche et différentes méthodes ou expériences), plus l'hypothèse devient forte, et plus elle sera acceptée par la majorité les scientifiques.

Nous pouvons voir les trois termes des définitions de l'ACM comme se situant à différents niveaux. La répétabilité se situe au niveau de l'auteur, qui doit déboguer ou refaire sa propre expérience. La reproductibilité se situe au niveau de l'article : d'autres chercheurs pourraient vouloir s'appuyer sur ce papier pour construire leur propre recherche, et poursuivre l'objectif d'améliorer les connaissances en évitant de « réinventer la roue ». Atteindre cet objectif implique que les articles soient publiés avec tous leurs artefacts, et cela améliorera la confiance dans les résultats publiés. Enfin, la réplicabilité se situe au niveau de la Science : différentes équipes de recherche, réalisant différentes expériences, mais obtenant la même conclusion scientifique. Ceci est obligatoire pour valider une hypothèse scientifique.

Nous pouvons voir que toutes ces notions sont cruciales pour la communauté scientifique. Il y a une montée d'intérêt pour la recherche reproductible qui peut désormais être observée dans les conférences et les revues scientifiques. La création des journaux dédiés à la reproduction des résultats des articles publiés est également un grand pas pour favoriser la recherche reproductible. Dans son article (Drummond 2018), Drummond, même s'il ne l'approuve pas, constate le fait que de nombreuses conférences (AAAS, AMP, ENAR, NSF, SIAM-CSE, SIAM-Geo) et journaux, dans le domaine de l'apprentissage artificiel, se préoccupent de plus en plus de la recherche reproductible. Stodden et ses collègues évaluent l'efficacité d'une politique de journal exigeant des auteurs qu'ils rendent disponibles les données et le code après publication (Stodden *et al.* 2018). Sur un échantillon aléatoire de 204 articles scientifiques publiés dans la revue à fort impact (Science), après la mise en œuvre de cette politique, les artefacts n'ont été obtenus que pour 44 % de l'échantillon. Ensuite, les résultats ont été reproduits avec succès pour 26 %. Cette politique des journaux apporte une amélioration

par rapport à l'absence de celle-ci, mais elle reste cependant insuffisante pour garantir la reproductibilité. Stodden et ses collègues estiment que les incitations des conférences et des revues améliorent la reproductibilité, et qu'il faut continuer de travailler dans cette direction.

I.3. Pourquoi pouvons-nous perdre la reproductibilité ?

Dans notre exploration des défis liés à la reproductibilité dans la recherche informatique, nous constatons que les raisons de pertes de reproductibilité sont multifactorielles, telles qu'illustrées dans la Figure 1. La perte de reproductibilité peut provenir de divers facteurs qui sont catégorisés sous les rubriques de l'informatique scientifique et du calcul à hautes performances, chacun influencé par le contexte scientifique et le chercheur en lui-même. Dans l'informatique scientifique, les problèmes de reproductibilité surviennent principalement, dans un contexte scientifique « indépendant » du chercheur, en raison de différences dans les environnements logiciels, les workflows, une culture scientifique encourageant ou non certaines bonnes pratiques et le degré d'ouverture de la science. Ces facteurs sont complétés par des problèmes pouvant venir du chercheur en lui-même tel que son niveau technique en génie logiciel, la qualité de la documentation et des statistiques.

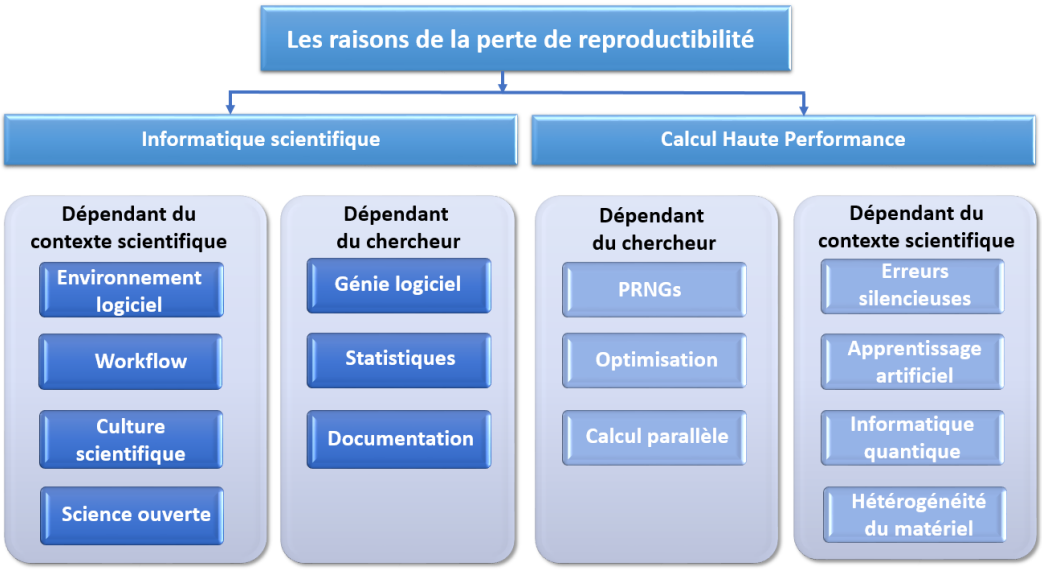


Figure 1: Les différentes raisons de pertes de la reproductibilité pour les sciences ayant recours à l'informatique

De l'autre côté, le calcul à hautes performances fait face à son propre ensemble de défis techniques, indépendamment de l'engagement des chercheurs, nous pensons aux erreurs silencieuses (particules cosmiques etc.), au bruit actuel sur les ordinateurs quantiques, mais

également aux compétences techniques que doivent acquérir les chercheurs pour maîtriser l'utilisation parallèle de générateurs de nombres pseudo-aléatoires (Hill *et al.* 2013) ou encore les techniques d'optimisation et les subtilités du calcul parallèle liées à l'évolution des architectures matérielles. Ces éléments soulignent collectivement la nature complexe et stratifiée de la perte de reproductibilité, qui peut à la fois dépendre du niveau technique du chercheur, de sa volonté à mettre en place certaines bonnes pratiques favorisant la recherche reproductible, mais également un contexte scientifique facilitant ou non la mise en place d'une recherche reproductible, certains facteurs étant extérieurs au chercheur.

I.3.1. La science ouverte

Une des raisons principales sous-jacente à l'incapacité de reproduire les articles scientifiques est la réticence ou l'échec des auteurs à partager leurs artefacts (codes, scripts, données, etc.). Sans partage de codes et de données, la reproductibilité d'un travail scientifique utilisant un ordinateur pour produire une partie de ses résultats est impossible (soit une grande partie de la recherche, que ce soit en informatique, mais également en biologie, et physique, et autres). L'essor de l'informatique parmi tous les domaines scientifiques a contribué à la crise de la reproductibilité, comme l'a bien décrit Ben Marwick (Marwick 2015). La plupart des auteurs qui plaident en faveur de la recherche reproductible soulignent l'importance du partage des codes et des données. En effet, de nombreuses études citées précédemment ont indiqué que les problèmes de reproductibilité proviennent principalement de l'absence de partage de ces éléments.

L'utilisation de logiciels propriétaires peut également entraîner des problèmes de reproductibilité. Dans (Nüst *et al.* 2020), les auteurs rappellent que nous avons besoin de « l'open source » pour pouvoir avoir une science reproductible. L'Académie Nationale des Sciences, de l'Ingénierie et de la Médecine (National Academies of Sciences 2019) déclare que les agences de financement devraient financer la science ouverte. Un autre aspect a son importance : la propriété intellectuelle du code publié. Sans licence explicite, les droits de propriété intellectuelle par défaut s'appliquent, comme décrit dans (Halchenko et Hanke 2015), ce qui peut freiner la possibilité de reproduire certains articles.

Enfin, d'autres raisons sont mentionnées de manière informelle et personnelle par les auteurs eux-mêmes dans le sondage mené par Collberg et Proebsting (2016). Par exemple, certains auteurs hésitent à publier leur code car ils estiment qu'il devrait être nettoyé ou modifié. Ils considèrent que leur code n'est pas assez propre pour être partagé, ce qui est regrettable.

Parfois, le code n'était pas initialement destiné à être partagé, et le rendre facilement accessible nécessiterait trop d'effort de la part de l'auteur qui n'est pas toujours à jour avec les meilleures pratiques en génie logiciel. Les problèmes de licence peuvent également empêcher les auteurs de publier leur code, en particulier lorsqu'il s'agit de logiciels non open source. De plus, une différence dans l'accessibilité du code a été observée entre le domaine public et le domaine privé. Fait intéressant, certains auteurs ne peuvent plus utiliser leur propre code parce que la seule personne qui savait l'utiliser a quitté le laboratoire ! Cela souligne l'importance de mettre en œuvre de bonnes pratiques de génie logiciel pour éviter de tels écueils. De surcroît, plus conventionnellement, le code peut être perdu. Cela peut être dû à des plantages ou à de mauvaises politiques de sauvegarde, parfois le langage de programmation ou le style peuvent être trop obscurs ou dépassés pour être utilisés par quelqu'un d'autre. La non-divulgation peut aussi être intentionnelle pour atténuer les risques tels que la découverte d'une vulnérabilité de sécurité, et d'empêcher son exploitation par quiconque. Les grands modèles de langages peuvent scruter les codes et aider des « hackers » mal intentionnés.

Le principal moyen pour accroître le mouvement de la science ouverte est d'en parler. De nombreux articles sur la recherche reproductible mettent en effet en lumière l'importance de la science ouverte. Cependant, il existe des exemples concrets d'actions entreprises pour renforcer l'ouverture de la science. L'un des premiers projets de science ouverte était peut-être le Stanford Exploration Project de Jon Claerbout. Plus récemment, le CERN a développé Zenodo, dans le cadre de la science ouverte. L'Inria propose Software Heritage pour archiver l'ensemble des versions de codes sources afin d'archiver la connaissance (Di Cosmo et Zacchiroli 2017). Les journaux réagissent également, comme le montre la politique de PLOS (PLOS one n.d.). Les États, qui ont un pouvoir de financement, doivent également s'impliquer dans la science ouverte, comme l'Institut National de la Santé (Collins et Tabak 2014) ou le gouvernement français (Ministère de l'enseignement supérieur et de la recherche 2021) qui soutient fortement l'initiative l'archive ouverte HAL. Enfin, l'intégration de l'enseignement de la science ouverte et de la reproductibilité dans le cadre éducatif des jeunes chercheurs pourrait permettre de faire avancer cette cause (Janz 2016), c'est la voie qu'ont choisi Konrad Hinsen, Arnaud Legrand, et Christophe Pouzat avec leur proposition de MOOC sur la recherche reproductible (Pouzat *et al.* 2018) qui se décline maintenant en deux niveaux.

I.3.2. Documentation

La documentation, dans le contexte de la recherche scientifique, fait référence à la sauvegarde et à la fourniture d'informations détaillées sur le code, les logiciels, les méthodes et les procédures expérimentales utilisées dans une étude. Dans le contexte de la recherche reproductible en informatique, l'importance de la documentation lors du partage du code est primordiale. Une documentation précise et complète est essentielle pour faciliter la réplication des résultats de recherche et promouvoir la transparence et la crédibilité au sein de la communauté scientifique. Comme le souligne l'étude (Boettiger 2015), une documentation incomplète ou imprécise sur la manière d'installer et d'exécuter le code peut être un obstacle significatif à la reproduction, en particulier pour les chercheurs qui ne sont peut-être pas familiers avec les outils spécifiques et les gestionnaires de paquets impliqués. Ce problème est encore souligné dans (Kitzes *et al.* 2018), où les auteurs discutent de la disponibilité des données et des logiciels, y compris l'importance d'une documentation adéquate, des pratiques open-source, des techniques d'ingénierie logicielle et des considérations sur les droits d'auteur.

Les meilleures pratiques, telles que définies dans (Wilson *et al.* 2014), soulignent la nécessité de documenter la conception et l'objectif du code plutôt que de se concentrer uniquement sur ses mécanismes. En réalisant cela, les chercheurs permettent à d'autres de comprendre la fonctionnalité du code et ses objectifs, facilitant ainsi la reproductibilité. L'étude dans (Boettiger 2015) souligne également l'impact d'une documentation imprécise sur la reproduction des analyses, un nombre significatif d'expériences utilisant des logiciels populaires n'étant pas reproductibles en raison d'une documentation incomplète.

De plus, dans le domaine en évolution rapide de l'intelligence artificielle (IA), les pratiques de documentation sont cruciales pour garantir la reproductibilité, comme le souligne Gundersen et Kjensmo (2018). De nombreuses études de recherche en IA manquent de méthodes et de codes bien documentés, rendant difficile la reproduction des résultats rapportés et freinant les progrès dans le domaine. Cependant, l'article reconnaît également que les pratiques de documentation se sont améliorées au fil du temps, soulignant l'importance de poursuivre les efforts pour promouvoir et maintenir des normes de documentation rigoureuses.

Une documentation efficace des logiciels et du code est un aspect fondamental de la recherche reproductible en informatique. Les chercheurs doivent s'efforcer de fournir une documentation claire et complète, incluant les procédures d'installation, les descriptions des paramètres et la justification de la conception. Une documentation adéquate permet non

seulement de reproduire les résultats de recherche, mais favorise également la collaboration, le partage des connaissances et l'avancement de la science.

I.3.3. Statistiques

En 2005, l'article de Ioannidis précédemment cité (Ioannidis 2005) a fortement impacté le monde de la recherche reproductible. Il affirmait que ses simulations démontrent que les affirmations des publications scientifiques sont plus susceptibles d'être fausses que vraies, soulevant des questions sur la qualité des résultats de recherche et leurs implications pour le développement scientifique. L'importance des statistiques en recherche est en effet majeure. La statistique joue un rôle crucial dans la détection et l'évitement de résultats faux-positifs, qui sont une préoccupation significative dans la recherche scientifique et dans le domaine médical en particulier. Plusieurs problèmes en statistique peuvent conduire à une perte de reproductibilité dans les articles, ce qui est plus, dans ce cas, une question de réplicabilité (la capacité à obtenir la même conclusion scientifique en réalisant une autre expérience).

Le « P-hacking », défini comme la manipulation des analyses statistiques pour obtenir des résultats statistiquement significatifs, est l'un des principaux facteurs contribuant à la crise de la reproductibilité. Les pratiques de P-hacking, telles que le choix sélectif des données ou la réalisation de multiples tests statistiques jusqu'à l'obtention d'un résultat significatif, peuvent conduire à des conclusions biaisées et peu fiables. Comme décrit dans (Forstmeier *et al.* 2017), ce problème est exacerbé par la pression lié à la culture scientifique de ne publier que des résultats statistiquement significatifs, ce qui peut donc créer un biais dans la recherche scientifique, puisque toutes les études trouvant un résultat négatif ne seront pas publiées, celles trouvant un résultat positif apparaitront comme correcte sans nuance. Ils présentent le fait que la diminution de la taille de l'échantillon (parfois pour des raisons éthiques d de préservation des animaux de laboratoires), la pression impliquant la recherche d'innovation et la réalisation de tests multiples peuvent tous augmenter la probabilité de conclusions positives erronées. Pour aborder ces problèmes, il est essentiel d'adopter des pratiques rigoureuses en recherche. Positionner les observateurs en aveugle pendant la collecte et l'analyse des données, et rapporter tous les résultats, qu'ils soient significatifs ou non, sont quelques-unes des stratégies qui peuvent améliorer l'objectivité de la recherche scientifique. De plus, réallouer les efforts de la recherche de nouveauté et de découverte vers la reproduction de résultats de recherche importants peut bénéficier à la communauté scientifique. Il est crucial d'évaluer la recherche sur la base de la rigueur scientifique plutôt que de se fier uniquement aux métriques liées aux facteurs d'impact

des journaux. La dépendance aux P-values comme mesure de preuve et de véracité est un autre aspect critique à considérer dans le contexte de la recherche reproductible. Les P-values sont souvent utilisées pour déterminer la force des preuves dans les résultats de recherche, mais des études ont remis en question leur fiabilité et leur objectivité (Nuzzo 2014). Les chercheurs ont noté que même de petits changements dans la significativité statistique peuvent entraîner des changements importants dans l'interprétation des résultats (Gelman et Stern 2006). Cette divergence peut conduire à des conclusions trompeuses et souligne la nécessité d'une réévaluation des méthodes statistiques (Nuzzo 2014).

Formuler des hypothèses après les résultats est nommé « HARKing », c'est une autre pratique courante dans la communication scientifique qui peut compromettre la reproductibilité (Kerr 1998). Le HARKing implique de présenter des hypothèses après-coup comme si elles étaient des hypothèses a priori dans les rapports de recherche. Cette pratique peut conduire à des interprétations biaisées et inexactes des données, car elle permet aux chercheurs de choisir sélectivement des hypothèses qui s'alignent avec leurs résultats. Bien que les motivations derrière le HARKing soient variées, il est largement considéré comme inapproprié et a des implications négatives pour l'intégrité scientifique.

Il est important de faire attention à ne pas surestimer l'importance de résultats qui semblent significatifs du point de vue statistique. Gelman et Stern (2006) ont expliqué que des variations importantes dans les niveaux de significativité statistique pourraient en réalité représenter des changements mineurs et non significatifs dans les données réelles. Cette tendance à sur-interpréter les résultats est une erreur différente d'autres problèmes communs en statistique, comme confondre la significativité statistique avec l'importance pratique, diviser les résultats en catégories « significatif » ou « non significatif », et choisir des seuils de significativité de façon arbitraire. Cette erreur est fréquente et il est donc crucial que les étudiants et les professionnels en soient conscients pour éviter de mal interpréter les données.

Les erreurs de type I et de type II sont des concepts dans les tests d'hypothèses liés à un rejet incorrect ou à un échec à rejeter une hypothèse nulle. Une erreur de type I, également connue sous le nom de faux positif, se produit lorsque l'hypothèse nulle est vraie, mais est rejetée à tort. Par exemple, conclure qu'un médicament est efficace alors qu'il ne l'est pas. Une erreur de type II, ou faux négatif, se produit lorsque l'hypothèse nulle est fausse mais échoue à être rejetée par erreur, comme conclure qu'un médicament est inefficace alors qu'il fonctionne réellement. La puissance statistique est la probabilité de rejeter correctement une fausse hypothèse nulle, ce qui signifie qu'elle mesure la capacité d'un test à détecter un effet lorsqu'il

y en a un. La significativité statistique, d'autre part, indique la probabilité que la différence ou l'association observée dans les données ne soit pas due au hasard, souvent évaluée par une « valeur p ». Une puissance statistique élevée et un résultat significatif statistiquement (typiquement une valeur p inférieure à 0,05) contribuent à la fiabilité des résultats des tests. Ce seuil se révèle être souvent bien insuffisant et J.P.A. Ioannidis a proposé de le ramener à 0,005 pour réduire le taux potentiel de faux positifs des résultats obtenus (Ioannidis 2005), l'article initial très cité a été récemment corrigé en 2022 (erreur mineure, une paire de parenthèse manquante dans la table 2) (Ioannidis 2022).

La puissance statistique est une question significative dans des domaines comme la psychologie. Dans le cadre hybride des tests de significativité de l'hypothèse nulle (NHST) prévalent dans la science psychologique, la puissance statistique est définie comme la probabilité de rejeter correctement une fausse hypothèse nulle, évitant ainsi une erreur de type II. À mesure que la puissance statistique augmente, la probabilité d'une erreur de type II diminue. Cependant, la puissance statistique a tendance à être sous-estimée par rapport au niveau de significativité. Typiquement, les chercheurs fixent un seuil de 5% pour les erreurs de type I mais acceptent souvent un taux plus élevé de 20% pour les erreurs de type II, indiquant une plus grande préoccupation pour éviter les faux positifs que les faux négatifs. Malgré le seuil conventionnel de 20% pour les erreurs de type II, l'attention réelle à la puissance statistique est souvent insuffisante, conduisant à un taux d'erreur de type II effectif qui pourrait être aussi élevé que 80%. Ce problème est évident dans divers domaines, soulignant une lacune critique dans les pratiques statistiques, comme décrit dans (Button et Munafò 2017), et cela peut conduire à des problèmes de reproductibilité (Clayson *et al.* 2019). Une solution serait d'augmenter la taille de l'échantillon et d'accroître la précision de la variable pour diminuer l'erreur de mesure, donc nécessiterait plus de calculs (Button et Munafò 2017). Dans (Strong et Alvarez 2019), les auteurs abordent la crise de la réplication dans la recherche psychologique en proposant l'utilisation du « bootstrap resampling » combiné avec des données pilotes ou synthétiques pour améliorer la puissance statistique et la reproductibilité des expériences. Cette méthode permet une analyse personnalisée des conceptions expérimentales, en tenant compte de facteurs tels que la taille de l'échantillon, le nombre d'essais et les critères d'exclusion, souvent négligés dans les outils traditionnels d'analyse de puissance. L'article explique comment le « bootstrap resampling » aide à estimer à la fois la variabilité de l'échantillonnage et de la mesure en simulant des expériences plusieurs fois, ce qui améliore la précision des prédictions de puissance. En utilisant une boîte à outils MATLAB fournie par les auteurs, les chercheurs

peuvent appliquer ces méthodes de simulation à leurs données pilotes pour concevoir des expériences qui sont plus susceptibles de produire des résultats reproductibles.

Certains outils sont conçus pour aider avec les statistiques, comme R Markdown, un outil conçu pour simplifier l'analyse statistique reproductible, le rendant adapté à la fois pour la recherche avancée et les cours d'introduction à la statistique (Baumer *et al.* 2014), ou bien Pernet *et al.* (2013) qui ont introduit une boîte à outils Matlab open-source conçue pour des analyses de corrélation robustes. La méthode de corrélation de Pearson, couramment utilisée en psychologie, a une portée limitée car elle ne détecte que les relations linéaires et est très influencée par les valeurs extrêmes, qui peuvent donner une image faussée des données. Pour pallier ces limites, une boîte à outils Matlab en accès libre propose deux alternatives : « percentage-bend correlation and skipped-correlations ». La première méthode réduit l'impact des valeurs extrêmes en leur attribuant moins de poids, tandis que la seconde les élimine directement de l'analyse. Ces approches offrent une meilleure approximation des véritables relations entre les données, en maintenant un équilibre entre la minimisation des erreurs positives et le maintien de la capacité à détecter des associations réelles.

I.3.4. Culture scientifique

Le paysage actuel de la recherche scientifique est affecté par plusieurs problèmes qui entravent la reproductibilité, essentielle pour l'avancement de la science. Un problème majeur est le manque d'incitations de la part des journaux et des conférences à fournir tous les artéfacts liés à un article de recherche, tels que les ensembles de données, le code ou des méthodes détaillées, qui sont essentiels pour une transparence et une reproductibilité complète. Nosek a écrit : « En raison de fortes incitations à l'innovation et de faibles incitations à la confirmation, la reproduction directe est rarement pratiquée ou publiée », et « Les découvertes innovantes rapportent des récompenses de publication, d'emploi et de titularisation ; les découvertes répliquées produisent un haussement d'épaules. » (Collaboration 2012), citation présente dans (Collberg & Proebsting, 2016). Cela crée une culture qui décourage les efforts de reproduction, amenant souvent les chercheurs en début de carrière à se détourner de telles entreprises au profit d'activités qui renforcent leurs profils académiques. Ce problème est aggravé par la mentalité enracinée du « publish or perish » qui met la pression sur les chercheurs pour produire continuellement de nouvelles découvertes, parfois au détriment d'une recherche rigoureuse et de qualité. Le processus de révision par les pairs n'est pas non plus sans failles. Souvent, il ne privilégie pas la reproductibilité des études, ce qui peut conduire à la publication de résultats

qui ne peuvent pas être vérifiés indépendamment. Enfin, le biais de publication en faveur des résultats positifs entraîne souvent une sous-représentation des résultats négatifs ou nuls, faussant davantage le paysage de la recherche. Collectivement, ces problèmes posent des défis importants à l'intégrité et à la fiabilité de la recherche scientifique, appelant à des changements systémiques dans la manière dont nous menons et communiquons la science. Bajpai *et al.* (2017) discutent de trois éléments qui posent des problèmes pour la reproductibilité : le manque d'incitations des journaux (qui privilégient uniquement les articles innovants), le processus de révision en double aveugle qui oblige les auteurs à cacher des informations ou des données potentiellement cruciales, et les relecteurs (reviewers) qui ne testent pas la reproductibilité. Le papier de Bajpai *et al.* propose d'inclure une section sur la reproductibilité dans les articles pour encourager les auteurs, promouvoir les articles reproductibles et améliorer le processus de relecture. Baker (2016) présente 14 facteurs qui peuvent contribuer à la perte de reproductibilité, principalement liés à des problèmes statistiques, à la pression de publication et au code non disponible, entre autres. Il propose 11 solutions qui abordent directement les problèmes soulevés, comme l'amélioration de la formation statistique des chercheurs. Munafò *et al.* (2017) identifient les biais cognitifs, les améliorations des méthodes, une collaboration accrue entre les chercheurs et une relecture par les pairs améliorée comme problèmes et solutions pour améliorer la reproductibilité. Ten Hagen (2016) soutient que les journaux privilégient excessivement la nouveauté, ce qui mine la recherche reproductible, car cela décourage les chercheurs de tenter de répliquer des résultats précédemment publiés. Fanelli (2010) montre que la pression des financements et de réussir à sortir une publication pousse les chercheurs à publier uniquement des résultats positifs, limitant la publication de résultats négatifs qui pourraient également contribuer aux progrès scientifiques en révélant ce qui ne fonctionne pas ou n'est pas vrai, y compris les résultats négatifs de reproduction d'articles précédemment publiés.

I.3.5. Environnement logiciel

L'environnement logiciel est un aspect crucial de la recherche reproductible, car l'utilisation des ordinateurs est devenue omniprésente dans le monde scientifique. Un grand nombre de publications utilisent désormais du code, des scripts ou des données en entrée pour générer certains résultats.

Même si vous partagez votre code et vos données, il est très peu probable que la personne tentant de reproduire les résultats ait exactement le même environnement logiciel que vous. Par

conséquent, elle pourrait ne pas être capable d'exécuter correctement le code en raison d'incompatibilités potentielles, de différentes bibliothèques, ou de logiciels non disponibles (Hinsen 2013). Pour résoudre ce problème, Hinsen suggère d'utiliser des bibliothèques fiables et éprouvées, d'écrire du code clairement (tout en tenant compte de la performance, en particulier dans le contexte du calcul à hautes performances), de documenter les formats utilisés, d'évaluer les dépendances et de fournir des exemples prêts à l'emploi pour faciliter la prise en main. En 2015, Hinsen continue de fournir des informations précieuses, conseillant la prudence concernant l'ajout de données d'entrée non mentionnées dans les workflows, et les éventuels bugs logiciels ou matériels qui en découlent (Hinsen 2014). Desquilbet *et al.* (2019) discutent également de ces différents défis dans leur livre. La couche de l'environnement logiciel englobe différents niveaux. Tout d'abord, nous avons le système d'exploitation. Les calculs peuvent différer à travers l'utilisation de différents systèmes d'exploitation, car l'utilisation du matériel peut différer. L'utilisation de systèmes basés sur Linux (open source) est obligatoire pour la recherche reproductible, car nous avons besoin de logiciels open source (contrairement à Windows ou MacOS qui ne sont pas open source, donc rendent plus difficile l'obtention de la même configuration pour assurer la reproductibilité). Linux est fortement conseillé car il est largement utilisé, entièrement open source et offre de nombreux outils pour mener à bien une recherche reproductible. Deuxièmement, comme mentionné, les codes utilisent presque toujours des bibliothèques. Ainsi, lorsqu'ils sont partagés, si la machine de destination n'a pas accès aux mêmes bibliothèques, avec les mêmes versions, le code ne fonctionnera pas. Chaque bibliothèque peut également avoir des dépendances avec d'autres. Cela est connu sous le nom de « Dependency Hell » (Boettiger 2015). Des outils de haut niveau, comme des cadriciels (frameworks par la suite) ou des bibliothèques, peuvent également agir comme des boîtes noires, qui ne permettent pas à un autre chercheur d'utiliser le code pour sa propre recherche. C'est ce que Hinsen a appelé "code réutilisable VS code rééditable" (Hinsen 2018).

En calcul à hautes performances, les compilateurs sont des logiciels importants. Le calcul intensif nécessite d'éviter le gaspillage de d'énergie et de temps de calcul, or les langages compilés sont connus pour être plus efficaces que les langages interprétés. Ainsi, les codes exécutés sur des clusters de calcul ou des superordinateurs sont principalement produits par des compilateurs C, C++ ou Fortran. Malheureusement, de nombreux scientifiques qui ne sont pas spécialisés en informatique, des biologistes ou des chimistes, peuvent par exemple trouver plus facile de travailler avec Python ou R, qui sont des langages utiles avec de nombreuses

bibliothèques que ce soit pour les aspects statistiques ou encore pour développer des modèles d'apprentissage automatique. Différentes versions des compilateurs, ou du langage de programmation, peuvent également entraîner une perte de reproductibilité. L'importance de l'ensemble de la pile logicielle a déjà été prêchée par Claerbout selon la citation de Donoho que nous avons mentionnée plus tôt (Buckheit et Donoho 1995).

I.3.6. Workflow

Le problème du workflow (littéralement flux de travail ou processus métier) dans la recherche reproductible tourne autour de la nécessité de capturer et de communiquer l'ensemble du processus d'analyse des données, d'expérimentation et d'exécution du code de manière claire et organisée, dans un ordre précis. En effet, la plupart du temps, un article n'utilise pas un seul code ou script, mais une succession de codes s'appliquant à différentes données. Un workflow reproductible devrait permettre à d'autres chercheurs de vérifier et de reproduire de manière indépendante les résultats présentés dans une étude.

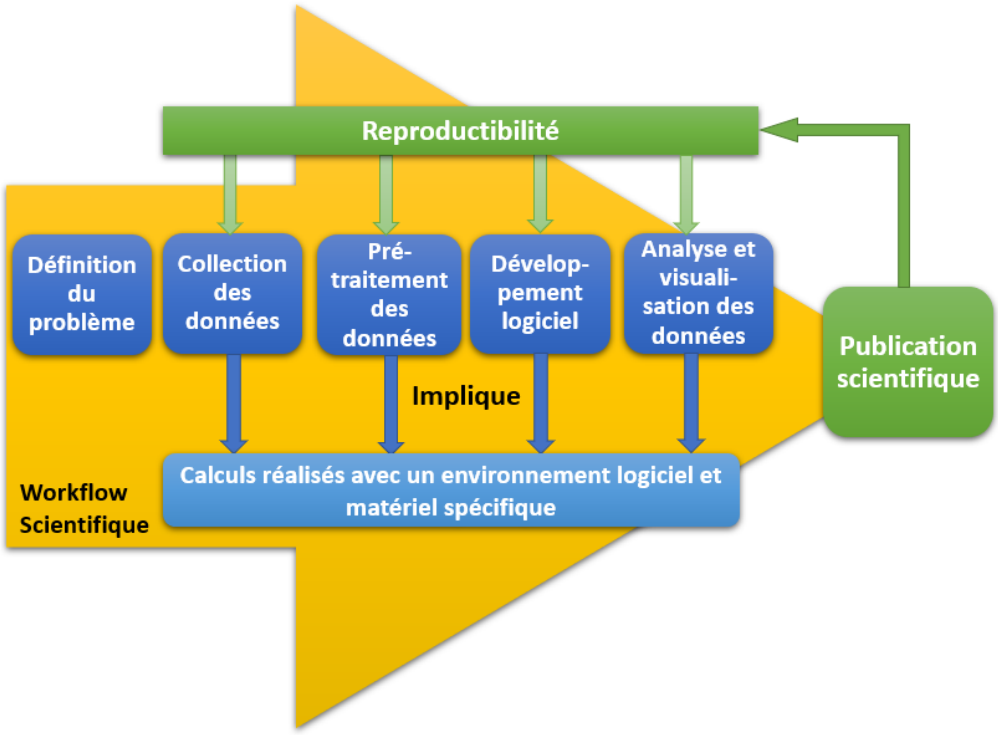


Figure 2: Exemple de workflow scientifique, amenant à une publication

Cependant, en pratique, les workflows manquent souvent de transparence, rendant difficile pour les autres de comprendre les étapes suivies, les paramètres utilisés et les transformations

de données appliquées. Ce manque de clarté peut venir d'une documentation incomplète ou ambiguë, rendant difficile la reproduction de la séquence exacte d'opérations qui a conduit aux résultats rapportés. De plus, les workflows peuvent impliquer plusieurs outils, bibliothèques et dépendances logicielles, et la gestion des interactions entre ces composants ajoute de la complexité au processus de reproductibilité. Le problème est aggravé lorsque les workflows sont répartis sur différents scripts, notebook ou même différents langages de programmation, il devient alors plus difficile de s'assurer que chaque détail est capturé de manière précise et cohérente. De plus, le contrôle de version devient crucial dans la gestion des changements apportés au workflow au fil du temps. Sans un suivi de version approprié, il peut être difficile de retracer l'état exact du workflow lorsqu'un résultat particulier a été obtenu.

Aborder le problème du workflow en recherche reproductible nécessite d'adopter les meilleures pratiques pour documenter les étapes, fournir des explications claires sur la raison des prises de décisions, et assurer la disponibilité de toutes les données et codes nécessaires. Sarah Cohen-Boulakia et ses collègues ont réalisé une étude assez exhaustive sur les workflows, axée sur les sciences de la vie (Cohen-Boulakia *et al.* 2017). Stanisic *et al.* (2015) ont également travaillé sur ce thème. Nous discuterons des outils utilisables par la suite. La figure 2 présente une représentation visuelle d'un concept de workflow d'une publication scientifique. Cette illustration met en évidence un workflow standard comprenant des étapes telles que la collecte de données, la préparation des données, l'élaboration de solutions logicielles et la conduite d'analyses de données, aboutissant à la création d'un article scientifique. Pour reproduire une telle étude, il est nécessaire qu'un scientifique soit capable de rejouer cette séquence entière, appelée le workflow. Le résultat précis de chaque étape dépend d'un environnement logiciel particulier, ce qui est une contrainte supplémentaire.

I.3.7. Génie logiciel

Nous avons mentionné que l'ingénierie logicielle est sujet important pour la recherche reproductible et qu'il peut être souvent négligé. En effet, avec l'omniprésence de l'informatique, de nombreux domaines scientifiques utilisent la programmation pour mener leurs recherches. Désormais, vous utilisez du code pour préparer des données, les transformer, les visualiser, réaliser vos statistiques, concevoir vos expériences, etc. Pour le calcul à hautes performances, vous aurez principalement besoin de compétences de bas niveau (réalisation de scripts au niveau du système d'exploitation, connaissance des systèmes d'exploitation et de langages informatiques efficaces comme C, C++ ou Fortran). Pour des objectifs plus généraux et

l'analyse de données, les scientifiques utilisent principalement Matlab, Python ou R, ainsi que certains frameworks, bibliothèques et autres outils de haut niveau. Toutes ces compétences peuvent être difficiles à acquérir pour certains d'entre eux, notamment pour ceux qui n'ont pas suivi un cursus classique en informatique (qu'ils soient biologistes, physiciens, sociologues ou chimistes…). Alors que les technologies de haut niveau offrent un moyen plus facile de gérer le calcul, elles sont également moins stables et plus obscures que les technologies de bas niveau, car elles cachent la complexité dans des boîtes noires, rendant la reproductibilité plus difficile. Ce sont les « défauts » des technologies modernes de haut niveau.

Lors de la publication, fournir des artefacts est une bonne chose, mais la qualité logicielle du code produit est essentielle pour avoir un code maintenable, lisible et évolutif. De plus, pour aider à garantir la reproductibilité, plusieurs outils ont été développés. Cependant, ils nécessitent souvent certaines compétences en informatique. Les non-informaticiens utilisent des ordinateurs comme un outil, sans avoir la connaissance suffisante de ce que fait réellement l'ordinateur (et peuvent supposer par exemple le déterminisme absolu de leurs machines). Dans (Hinsen 2018), Konrad Hinsen présente un exemple de pourquoi il est important de savoir ce qui se passe en coulisse. Dans un exemple, il charge un jeu de données sur la santé avec le logiciel Pandas à partir d'un fichier csv (il a pris cinq ans de données). Pandas essaie automatiquement de trouver les types de données appropriés pour les données, Hinsen a remarqué un biais. Alors que le type utilisé pour ses données était « int64 », lorsqu'il a chargé l'ensemble du jeu de données (33 ans contre 5 ans), les données ont été reconnues comme étant désormais du type « objet » (et non plus « int64 »). Dans la documentation de Pandas, il est mentionné une « conversion intelligente des données tabulaires », mais les règles utilisées ne sont pas explicites. Bien sûr, vous pouvez gérer cela avec un peu d'effort, mais cela montre qu'il y a des complexités cachées auxquelles un non-expert pourrait ne pas penser. L'article souligne également le fait qu'un code réutilisable et rééditable ne sont pas la même chose. Si vous fournissez du code avec votre article, et dans votre analyse, vous utilisez une fonction complexe d'une bibliothèque de haut niveau, d'autres chercheurs pourront-ils éditer cette fonction pour leur propre usage ? Les fonctions de haut niveau peuvent agir comme une boîte noire, et toute mise à jour de la bibliothèque peut vous faire perdre la reproductibilité au bit près. Un code rééditable sera plus compréhensible pour la majorité.

Comme discuté précédemment, il existe des cas où un groupe de chercheur pourrait cesser d'utiliser leur propre logiciel car la seule personne compétente connaissant son fonctionnement ne fait plus partie de l'équipe. Cela souligne la nécessité d'adhérer à des principes robustes

d'ingénierie logicielle pour prévenir ce type de scénarios. De plus, dans une situation plus typique, un logiciel peut devenir inaccessible en raison d'une défaillance matérielle rendant le code inutilisable (suppression par exemple), ou lorsque le langage de programmation utilisé pour développer le programme pourrait devenir si rare ou si archaïque qu'il est peu pratique pour d'autres chercheurs de l'adopter.

I.4. Les problèmes de reproductibilité propre au calcul haute performance

I.4.1. Exécution de programmes en parallèle

Le calcul parallèle implique l'exécution simultanée de multiples tâches pour un traitement plus rapide de la charge de travail. Avec le calcul parallèle viennent de nouveaux défis, tels que les changements dans l'ordre d'exécution d'opérations en virgule flottante, qui peut introduire des résultats numériques non reproductibles (Goldberg 1991). La combinaison du comportement non déterministe dans les programmes parallèles et de la non-associativité des opérations en virgule flottante pose des défis significatifs en termes de reproductibilité, comme le montre l'exemple de la Figure 3.

```
>>> (pow(10,-3)+1)-1
0.0009999999999998899
>>> pow(10,-3)+(1-1)
0.001
```

Figure 3: Exemple de la non-associativité des opérations à virgule flottante

Dans (Hunold 2015), cet état de l'art nous apprend qu'une majorité de chercheurs en informatique utilisant le calcul à hautes performances sont préoccupés par la reproductibilité de leurs articles. Une grande majorité, 94%, croient que la reproductibilité des articles devrait être améliorée dans le domaine du calcul parallèle. Dans les mêmes proportions, ils pensent que les articles de recherche actuels dans le domaine du calcul parallèle sont très peu reproductibles. En calcul parallèle, l'ordre d'exécution des tâches peut varier en raison de facteurs tels que l'ordonnancement des tâches, l'équilibrage de charge et le multithreading. Dans (Chohra *et al.* 2016), les auteurs supposent que le non-déterminisme peut provenir de l'ordonnancement dynamique des données, de la disponibilité des ressources physiques et de différents ensembles d'instructions du processeur. Ce comportement non déterministe peut conduire à différents résultats lorsque le même programme est exécuté plusieurs fois dans les mêmes conditions. Cela pose des défis pour la reproductibilité, car les résultats peuvent varier de manière

imprévisible et cela rend le débogage et la validation des résultats particulièrement difficiles. De plus, la non-associativité de l'arithmétique en virgule flottante exacerbe le problème. Les opérations en virgule flottante telles que la multiplication et l'addition ne sont pas associatives dans l'espace des fractions Q où sont situés les nombres réels en virgule flottante. Cela signifie que changer l'ordre des opérations peut donner des résultats différents. Lorsque ces opérations non associatives sont combinées avec le calcul parallèle, atteindre des résultats identiques au niveau binaire, ce qui peut être un aspect critique du débogage, devient un défi encore plus significatif dans le calcul Exascale (Demmel et Nguyen 2013b). Les systèmes Exascale, se référant aux superordinateurs capables d'effectuer 10^{18} opérations en virgule flottante par seconde, présentent une puissance de calcul immense mais posent également des défis significatifs pour la reproductibilité. Parmi les deux opérations en virgule flottante non associatives, l'addition est plus sensible. Cela implique la production d'algorithmes de compensation, comme des sommes compensées par exemple, qui sont particulièrement utile pour la phase de réduction. Il s'agit d'une étape cruciale dans les algorithmes parallèles où les résultats partiels calculés par différentes unités de traitement en parallèle (par exemple, des threads, des processeurs, des nœuds de calcul) sont combinés ou agrégés pour produire un résultat global. Pour les opérations flottantes, l'ordre et la façon dont on organise les calculs pour compenser les erreurs d'arrondis sont importants.

Nous présenterons des solutions pour les problèmes de reproductibilité de ces calculs parallèles dans la section 7. Avec le nouveau superordinateur Frontier (Top500) et de nombreux autres superordinateurs proposés depuis plus d'une décennie, nous avons dépassé le million de cœurs capables de réaliser des calculs en parallèle.

L'utilisation de grilles de calcul (« Grid Computing ») ou de bibliothèques parallèles telles que Message Passing Interface (MPI), avec le passage de messages asynchrones sur de grandes simulations, peut fréquemment conduire à une exécution où l'on change l'ordre des opérations dans les calculs sur des nombres en arithmétique flottante. Dans ce cas, nous avons une perte de répétabilité, ce qui augmentera considérablement la difficulté de débogage.

I.4.2. Simulations de Monte-Carlo et nombres pseudo-aléatoires

Lors de notre recherche bibliographique de la recherche reproductible pour le calcul à hautes performances, nous avons souvent vu l'affirmation selon laquelle les simulations de Monte Carlo ne sont pas déterministes. Bien qu'il soit vrai que les simulations de Monte Carlo peuvent être non déterministes pour les mêmes raisons que d'autres programmes, telles que

l'arithmétique à virgule flottante désordonnée, le parallélisme, etc., nous soutenons que qualifier les simulations de Monte Carlo de non déterministes en raison de l'utilisation d'une source pseudo-aléatoire peut être trompeur. En fait, lors de la production de résultats scientifiques, la méthode de Monte Carlo utilise précisément des modèles déterministes de hasard appelés « Générateurs de Nombres Pseudo Aléatoires » (PRNG). C'est l'approche scientifique utilisée pour simuler le hasard en le maîtrisant et en fournissant de très bonnes propriétés statistiques. C'est un aspect important dans la conception des générateurs de nombres pseudo-aléatoires, car il est alors possible d'obtenir la répétabilité pour le débogage de chaque expérience « indépendante ». Lors de l'exécution de modèles stochastiques, tels que les simulations de Monte Carlo, les scientifiques expérimentés font un usage intelligent des états des PRNGs. Dans le cas des anciens générateurs il s'agissait de simple graines (seed), afin d'assurer la répétabilité des résultats. La génération de nombres pseudo ou quasi aléatoires est totalement déterministe, ce qui est un atout fort pour les études scientifiques. Cependant cela suppose un usage correct des PRNGs pour obtenir un calcul stochastique reproductible, particulièrement lorsqu'il s'agit de Monte Carlo parallèle. Hellekalek a justement averti les chercheurs en simulation lors de la conférence sur l'informatique parallèle et distribuée de 1998 : « Ne faites pas confiance au Monte Carlo parallèle » (Hellekalek 1998). Nous avons également trop souvent vu le mauvais conseil d'initialiser votre PNRG avec « time(NULL) » pour permettre un « vrai aléatoire » y compris dans un ouvrage sur le calcul parallèle ! C'est un mauvais conseil qui ne devrait pas être appliqué pour pouvoir déboguer ou faire un travail statistiquement correct et reproductible. Cette approche utilisable pour des jeux, ou pour produire rapidement quelques jeux d'essais sur un programme débogué est encore trop souvent enseignée. Pour permettre la reproductibilité avec les PRNGs, vous devez maîtriser et sauvegarder méthodiquement les états initiaux et sélectionner une technique de parallélisation avant d'exécuter simultanément des instances « indépendantes » de simulations de Monte Carlo (Hill *et al.* 2013). La figure 4 présente une méthode de parallélisation de flux aléatoires grâce à l'utilisation d'un espacement aléatoire, permettant l'exécution concurrente de calculs parallèles indépendants. Cette approche repose sur le fait que les flux ne se chevauchent pas afin de maintenir leurs opérations distinctes. La méthode d'espacement aléatoire est particulièrement efficace et simple pour les PRNGs ayant de très grandes périodes, où la probabilité de chevauchement des flux est extrêmement faible lors de la sélection de flux aléatoires sur l'ensemble de la période du PRNG. Dans les cas où le risque chevauchement entre les flux est statistiquement trop élevé, des méthodes alternatives comme le fractionnement de séquences sans recouvrement peuvent être utilisées. De plus, il est crucial pour les chercheurs dans la communauté scientifique de comprendre que le terme

historique de graine (seed) est différent de l'état complet d'un PRNG moderne. L'état du PRNG dicte les valeurs qu'il produit. Inversement, utiliser une graine implique l'application d'une fonction spécifique pour transformer la graine en un état complet. Il est important de noter que ce processus de transformation peut varier selon les différentes plateformes et technologies, de sorte que la même graine peut conduire à différents flux de nombres aléatoires pour le même algorithme supposé de PRNG, dans différents langages de programmation ou frameworks, ceci impact en particulier les outils d'apprentissage sur lesquels bien des études d'intelligence artificielle se basent.

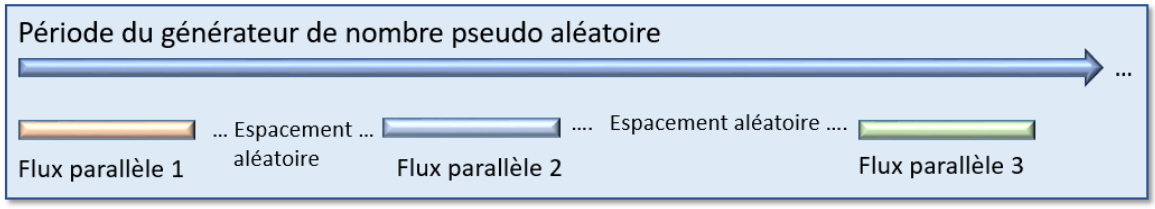


Figure 4: Exemple de partition de flux stochastiques en utilisant la méthode d'espacement aléatoire (random spacing)

Nous pouvons rencontrer des problèmes en traitant la parallélisation des PRNGs, cependant, nous disposons désormais de PRNGs de haute qualité qui peuvent être parallélisés correctement. Si vous utilisez les bonnes méthodes adaptées aux meilleurs PRNGs, comme décrit dans (Traoré et Hill 2001 ; Hill *et al.* 2013), il est possible de paralléliser les simulations de Monte Carlo et d'obtenir une répétabilité avec des résultats identiques au niveau binaire pour toutes les expériences stochastiques (Hill 2015). (Reuillon *et al.* 2008) proposent un outil de parallélisation des flux. Voici une courte liste de certains PRNGs de bonne qualité utilisés pour le calcul parallèle, à partir de la sélection retenue dans (Antunes et Hill 2023) :

- Philox et Threefry, proposés par Salmon et ses collègues lors de la Conférence Supercomputing de 2011 (Salmon *et al.* 2011). Ils reposent sur des techniques cryptographiques comme AES (Rijmen et Daemen 2001). Ils proposent une technique de paramétrisation solide et facile pour résoudre le problème de distribution de flux stochastiques « indépendants » dans les applications parallèles. En raison de ses contraintes cryptographiques, on peut le considérer comme un générateur lent, mais statistiquement, ce type de générateur est très bon même si nous avons noté certains problèmes de reproductibilité (Hill 2015). Les PRNGs sécurisés d'un point de vue cryptographique sont également connus sous le nom de CS-PRNG (CS pour Crypto-Secure).

- MRG32k3a, est un générateur récursif combiné, il a été minutieusement sélectionné par L'Ecuyer par force brute pour répondre aux tests statistiques les plus rigoureux qu'il a lui-même développés (L'ecuyer 1999). Ce générateur facilite le fait d'avoir de nombreux flux parallèles, c'est un atout dans le calcul parallèle. Cependant, il peut être jusqu'à 19 fois plus lent que le Mersenne Twister dans sa mise en œuvre en C/C++, le rendant moins attrayant pour le calcul parallèle intensif. MRG32k3a est le choix par excellence pour la plus haute qualité statistique, tandis que Mersenne Twister offre de meilleures performances avec une solidité statistique légèrement moindre.
- WELL, développé par Panneton, L'Ecuyer, et Matsumoto (Panneton *et al.* 2006), améliore le Mersenne Twister d'un point de vue statistique. Provenant des registres à décalage à rétroaction linéaire (LFSR), WELL n'a pas connu une utilisation ou un développement des techniques de parallélisation aussi importante que Mersenne Twister.
- PCG, un PRNG récent, a été créé en 2014 par O'Neill (O'neill 2014). Il est plébiscité pour ses propriétés statistiques supérieures par rapport aux autres générateurs. La documentation de Numpy recommande son utilisation en raison de ces qualités, mais nous avons observé des failles en faisant un grand nombre de tests avec la bibliothèque de tests statistique TestU01 (discutée plus tard). D'autre part, la documentation de numpy mentionne que des faiblesses statistiques ont été repérées lors de l'utilisation de PCG64 dans un contexte de parallélisme massif. De fait ils ont introduit la version nommée PCG64DXSM que nous n'avons pas encore testée.
- Mersenne Twister (MT), introduit par Matsumoto et Nishimura (Matsumoto et Nishimura 1998), est rapidement devenu connu en raison de sa période gigantesque de 2^{19937} - 1. C'était le premier d'une famille de générateurs, certains conçus pour les GPU et les Field Programmable Gate Array (FPGA). La version SFMT utilise les possibilités vectorielles des processeurs modernes (Single Instruction Multiple Data), et est plus rapide que le MT original, il a de meilleures propriétés statistiques et propose une période encore plus grande allant jusqu'à 2^{216091} - 1, mais il est beaucoup moins connu que le MT original (Saito et Matsumoto 2006). Cependant, aucun des PRNGs de cette famille n'est adapté aux applications cryptographiques. Les failles connues ne posent pas de problèmes aux applications de simulation et son implémentation optimisée est particulièrement rapide contrairement à ce que l'on peut lire parfois.
- La dernière version de Xoshiro par (Blackman et Vigna 2021), basée sur un principe « XOR, shift, rotate ». Les auteurs proposent l'applications de « scramblers » non-linéaire

sur les sorties du générateur Xoshiro (ainsi que sur Xoroshiro – Xor Rotate Shift Rotate de la même famille). Les auteurs affirment, que tout en maintenant la rapidité du générateur, ils ont fourni des améliorations pour éviter les échecs dans les tests statistiques liés à la linéarité tels que MatrixRank et LinearComp de TestU01 (sur lequel échoue le Mersenne Twister, un échec au test MatrixRank peut également survenir pour MT qui n'est pas un générateur de qualité cryptographique).

L'utilisation de certaines versions de ces algorithmes n'est pas toujours explicitement spécifiée par les langages, les bibliothèques ou les frameworks (cadriciels). Par conséquent, cela peut entraîner des différences dans la qualité statistique et des différences dans le flux de nombres aléatoires générés, et ceci peut conduire à des difficultés pour reproduire le travail d'autres personnes, qui se basent simplement sur le nom des PRNGs utilisés. Par exemple, les langages R et Python, en considérant le module random de la bibliothèque standard, utilisent tous les deux le générateur Mersenne Twister de Matsumoto et Nishimura (générateur par défaut pour R et unique générateur pour Python). Par contre, R utilise la première version (1998) de MT qui renvoie un entier de 32 bits divisé par 2^{32} (et utilise donc seulement 32 des 53 bits de la mantisse d'un double) et qui connais des problèmes lors d'initialisation dans certains états. Il s'agit d'un ancien défaut connu sous le nom de « zero excess initialization state ». Il se manifeste lorsque l'état du générateur est initialisé avec un excès de zéros, affectant ainsi la qualité statistique des nombres générés. Ce problème a été corrigé par la suite. Python utilise la version de 2002 qui a corrigé ce défaut dans la version initiale de MT et dans ses variantes (Saito et al, 2008). Cette version combine deux tirages successifs d'entiers de 32 bits afin d'obtenir un double pour lequel tous les bits de la mantisse sont utilisés (ce dernier point n'est malheureusement pas documenté dans Python). Cet exemple illustre bien comment différentes versions d'un même algorithme peuvent entraîner des variations dans les flux de nombres aléatoires générés ainsi que dans la qualité statistique de ces derniers.

Une autre famille de nombres aléatoires existe : les vrais nombres aléatoires. Les vrais nombres aléatoires sont des nombres générés de manière imprévisible et non déterministe (contrairement aux nombres pseudo-aléatoires évoqués précédemment). Un générateur de vrai nombres aléatoire (TRNG, pour True Random Number Generator) est un dispositif ou une méthode qui génère de véritables nombres aléatoires. Contrairement aux PRNGs, qui utilisent des algorithmes déterministes et un état initial pour produire des séquences de nombres qui semblent aléatoires, les TRNGs s'appuient sur des processus physiques imprévisibles ou des sources de hasard pour générer de vrais nombres aléatoires. Certaines sources de hasard

couramment utilisées dans les TRNGs peuvent être le bruit électronique, la désintégration radioactive, le bruit atmosphérique ou le bruit optique. Pour faire de la science reproductible avec des TRNGs, vous devez sauvegarder chaque nombre aléatoire que vous utilisez. Lorsqu'on traite des simulations de Monte Carlo lourdes, comme en physique des hautes énergies, utiliser des TRNGs n'est pas une option viable, car la sauvegarde de millier de milliards de nombres utiliserait trop de ressources matérielles. De plus, les vrais nombres aléatoires sont souvent lents à générer, comparés aux nombres pseudo-aléatoires.

Une autre catégorie de générateurs de nombres aléatoires est connue sous le nom de générateurs de nombres quasi aléatoires (QRNG – Quasi Random Number Generator), qui sont conçus pour générer des séquences de nombres qui minimisent l'écart à l'uniformité de façon déterministe. Cette approche est intéressante pour le calcul intégral ou pour les problèmes de « space-filling » que l'on utilise notamment dans la réalisation de plans d'expériences très utiles en calcul à hautes performances.

I.4.3. Optimisation

Dans le monde du calcul haute performance, l'optimisation est une notion importante. Lorsque nous parlons d'optimiser l'exécution d'un processus sur un CPU, nous pouvons penser aux options de compilation –O2 et –O3 (pour C, C++ et Fortran), et nous pouvons également penser aux opérations Fused Multiply-Add (FMA), Advances Vector Extension (AVX), et à l'utilisation de cartes accélératrices avec des « Tensor Cores » capables de réaliser des multiplications de petites matrices en un cycle. Le FMA est une opération arithmétique qui combine la multiplication et l'addition en une seule instruction. La nature fusionnée de cette opération signifie que la multiplication et l'addition sont effectuées à l'intérieur du processeur au niveau matériel en une étape « unique », améliorant l'efficacité computationnelle des deux opérations suivantes : « a * b + c ». AVX est une extension du jeu d'instruction de l'architecture « x86 » qui permet des opérations SIMD (Single Instruction, Multiple Data). L'architecture SIMD permet au processeur de traiter plusieurs éléments de données en parallèle avec une seule instruction vectorielle, augmentant ainsi le débit de calcul. AVX introduit des registres vectoriels plus larges (par exemple, 256 ou 512 bits), permettant à une seule instruction vectorielle d'opérer simultanément sur plusieurs éléments de données en simple ou double précision. Cela est particulièrement bénéfique dans les applications spécifiques, où la même opération doit être effectuée simultanément sur plusieurs points de données. Par exemple, avec la norme AVX-512, un registre vectoriel de 512 bits peut contenir cent vingt-huit nombres à

virgule flottante de simple précision (32 bits) ou soixante-quatre nombres à virgule flottante de double précision (64 bits). Lors de la compilation avec l'optimisation agressive –O3, toutes ces fonctionnalités d'optimisation peuvent conduire à une perte de répétabilité. Nous avons observé une perte de répétabilité d'une exécution à l'autre avec un contexte identique sur des superordinateurs comme décrit par le Prof. Thomas Ludwig, directeur du DKRZ lors de sa présentation à l'atelier reproductibilité de la conférence Européenne de super calcul de Francfort en 2019 (Ludwig 2019). La suppression des optimisations vectorielles « agressives » permet de retrouver un comportement normal avec une performance moindre.

Avec la généralisation des exécutions dynamiques à l'intérieur des CPU (également connues sous le nom de « out-of-order », des changements de l'ordre d'exécution des instructions), créées pour mieux alimenter la pile d'exécution des CPU, nous observons depuis plus de problèmes avec l'arithmétique à virgule flottante en HPC. De plus, comme le montrent Mytkowicz *et al.* (2009), même l'évaluation de la performance d'une fonctionnalité d'optimisation comme –O3 peut être non reproductible.

I.4.4. Hétérogénéité du matériel

Dans un environnement de calcul haute performance, tel que les grilles de calcul utilisées par le CERN, les ressources matérielles peuvent être très hétérogènes. Cette hétérogénéité peut avoir des implications significatives pour la reproductibilité, car elle introduit des variations en termes de performance, de précision et de comportement d'exécution entre différents systèmes (Boyer 2022). En effet, l'hétérogénéité potentielle des plateformes matérielles et des réseaux soulève des préoccupations quant à la reproductibilité. La planification dynamique nécessaire pour s'adapter aux ressources et charges changeantes rend difficile l'exécution constante des opérations dans le même ordre lors de différentes exécutions sur une plateforme à mémoire distribuée. La figure 5 illustre un scénario de charge de travail standard pour les expériences LHCb du CERN. Dans cette représentation, divers nœuds de travail sont engagés dans la récupération et le traitement des tâches. Notamment, ces nœuds sont équipés d'un large éventail de composants matériels, illustrant le concept de calcul en grille dans un environnement de calcul à hautes performances.

Nous pouvons également rencontrer une variabilité de performance en raison de l'hétérogénéité du matériel. Différentes architectures matérielles présentent des capacités de traitement variables, telles que la vitesse d'horloge des CPUs, le nombre de cœurs, la bande passante de la mémoire, les configurations du BIOS et la taille du cache. En conséquence, le

même code exécuté sur différentes machines peut présenter des caractéristiques de performance différentes. Cette variabilité peut entraîner des écarts dans le temps nécessaire pour achever les calculs, rendant difficile la comparaison et la reproduction cohérente des résultats à travers diverses configurations matérielles.

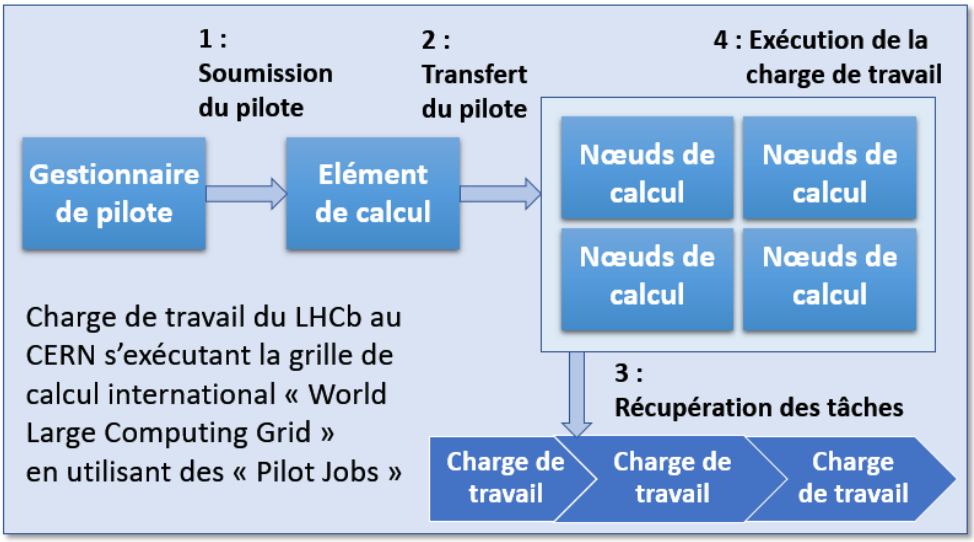


Figure 5: Charge de travail typique pour les expériences du LHCb

De plus, les CPUs peuvent également être très différents au niveau de leur conception, mettant en œuvre différentes technologies et ensembles d'instructions, y compris SIMD (par exemple, AVX dont nous venons de parler), qui peuvent accélérer certains calculs. Cependant, le code utilisant des ensembles d'instructions spécifiques peut produire des résultats différents sur des CPU qui ne prennent pas en charge ces instructions. Cette divergence peut entraîner des différences de performance et, dans certains cas, une divergence numérique dans les résultats. Cela peut également affecter la précision en virgule flottante (par exemple, double précision, précision étendue). L'utilisation de différentes précisions peut entraîner des variations dans la précision et l'exactitude numérique des résultats finaux. Dans certains cas, ces différences peuvent être négligeables, mais dans d'autres, lorsqu'elles sont répétées intensivement, elles peuvent affecter de manière significative le résultat final.

Toujours concernant le matériel, les mémoires peuvent varier, et avec elles, leurs performances. Des hiérarchies de mémoire diverses, telles que différentes tailles de cache et latences d'accès à la mémoire, peuvent affecter l'efficacité des applications intensives en

mémoire (certaines applications étant limitée en performance par leurs accès à la mémoire). Cela aussi peut entraîner des performances variables.

Sur les grilles de calcul, nous cherchons également non seulement la parallélisation, mais aussi l'équilibrage de charge, et cet aspect est difficile. Un code optimisé pour une architecture parallèle spécifique peut ne pas utiliser efficacement d'autres architectures, conduisant à des schémas différents de parallélisation et, par conséquent, à différentes performances et résultats numériques.

Enfin, la pile logicielle peut être différente, comme les compilateurs, les bibliothèques, le système d'exploitation, etc. Toutes ces variations peuvent conduire à un code généré ou exécuté différent, modifiant potentiellement la performance et le comportement numérique de l'application.

Pour toutes ces raisons, nous pouvons faire face à des bogues spécifiques à la plateforme. De tels problèmes peuvent conduire à des résultats non reproductibles sur certains systèmes, rendant difficile l'identification et la correction de la cause principale.

I.4.5. Informatique quantique

Une nouvelle branche du calcul haute performance émerge avec le calcul quantique. Cela est prometteur pour l'optimisation de la résolution de problèmes (Hogg et Portnov 2000). Des outils sont développés pour faciliter l'utilisation du calcul quantique, qui peut rester obscur pour les personnes habituées à l'informatique déterministe classique. (Shaydulin *et al.* 2021) proposent un toolkit Python nommé QAOAKit, conçu pour la recherche d'Algorithme d'Optimisation Quantique Approximatif (QAOA). Il sert de dépôt unifié de paramètres QAOA pré-optimisés et de générateurs de circuits quantiques. En intégrant des paramètres connus pour le problème MaxCut et en fournissant des outils de conversion pour divers cadres de simulation quantique, QAOAKit facilite la reproductibilité, la comparaison et l'extension des résultats de recherche en optimisation quantique, abordant des questions ouvertes sur la performance et le comportement de l'algorithme.

IBM est à l'avant-garde de l'accès aux machines quantiques avec leur plateforme IBM Quantum Experience. Cette plateforme permet aux chercheurs et aux passionnés de réaliser des expériences sur les processeurs quantiques d'IBM, qui peuvent avoir jusqu'à 127 qubits depuis 2023. Cet accès remarquable est facilité par Qiskit, un framework de développement de logiciels de calcul quantique open-source, qui permet aux utilisateurs d'écrire des algorithmes

quantiques en Python. Il y a également QASM (Quantum Assembly Language), qui offre un moyen d'exprimer des circuits quantiques à un niveau inférieur, permettant un contrôle plus fin des opérations quantiques.

De plus, les méthodes de création de qubits sont variées et font l'objet de recherches actives. En général, les qubits peuvent être réalisés en utilisant plusieurs systèmes physiques. Les méthodes les plus utilisées incluent les ions piégés, où des ions individuels sont confinés et manipulés avec des champs électromagnétiques; les circuits supraconducteurs qui exploitent les propriétés mécaniques quantiques de circuits électroniques macroscopiques refroidis à près de zéro absolu; et les qubits à semi-conducteurs, qui sont fabriqués en utilisant des principes similaires à ceux des puces informatiques conventionnelles. Chacune de ces méthodes a ses propres avantages et défis, notamment en termes de passage à l'échelle, de temps de cohérence et de taux d'erreurs. Les chercheurs essayent continuellement d'améliorer ces systèmes, dans le but de créer des qubits plus fiables et durables qui pourraient ouvrir la voie à un calcul quantique plus utile.

Cependant, avec le calcul quantique pour l'optimisation haute performance, de nouveaux défis de reproductibilité apparaissent. Nous pouvons voir que la reproductibilité dans le calcul quantique est l'une des principales préoccupations, nous avons besoin de milliers d'excellents qubits pour obtenir un seul qubit « parfait ». Dans (Mauerer et Scherzinger 2022), ils restent à un niveau élevé, proposant des solutions pour gérer les paquets et les dépendances, ce qui est un problème qui n'est pas seulement propre au calcul quantique, au contraire. Si nous nous penchons plus profondément sur l'aspect technique du calcul quantique, nous pouvons en effet comprendre qu'il a des problèmes intrinsèques (Dasgupta et Humble 2021a; Dasgupta et Humble 2021b; Dasgupta et Humble 2022; Hill *et al.* 2023) en raison du comportement non déterministe, et du fait qu'il est physiquement difficile d'avoir des qubits fiables en raison de la décohérence, du bruit, du manque de correction d'erreur, etc. IBM offre un accès gratuit pour utiliser ses machines quantiques, et ainsi, beaucoup de recherches sont menées sur leurs machines. Dans (Hill *et al.* 2023), les auteurs ont mis en œuvre un algorithme de Grover à trois qubits. L'oracle est codé (avec des portes quantiques) pour trouver la propriété « être égal à cinq ». Ils ont exécuté l'algorithme sur un simulateur quantique, montrant qu'ils devraient obtenir la bonne solution avec une probabilité de 94%; le calcul quantique étant stochastique par essence, les résultats sont donnés par probabilités. Mais lors de l'exécution sur différentes machines quantiques réelles, les résultats n'étaient pas si bons et ils ont trouvé beaucoup de variabilité. Au mieux, ils ont obtenu la bonne solution avec 60% (ce qui est bien moins que ce qui était

prévu selon la simulation, mais est encore utilisable). Le problème de reproductibilité est survenu lors de l'exécution de cette expérience plusieurs fois sur la même machine quantique, ou sur différentes machines quantiques. Avec exactement le même code et les mêmes paramètres, ils ont obtenu à chaque fois des résultats statistiquement différents, la plupart n'étant pas ceux attendus. Cela remet en question l'utilisabilité des ordinateurs quantiques actuels, mais il faut noter des améliorations rapides lorsque nous avons testé le même algorithme sur 3 et 5 qubits un an plus tard (Antunes et Hill 2024). Les simulateurs ou les machines quantiques hybrides sont intéressants pour développer des compétences et dans un but d'apprentissage, mais nous ne pourrons produire des résultats scientifiques approfondis sans une reproductibilité fiable.

Le calcul quantique est une technologie jeune et passionnante. Il est probable que nous verrons bientôt des technologies innovantes et obtiendrons des machines quantiques générales avec des qubits très utilisables dans les prochaines décennies.

I.4.6. Apprentissage artificiel

Une utilisation récente des ressources de calcul à hautes performances traite de l'apprentissage artificiel avec des données massives (Big Data). L'apprentissage artificiel implique l'utilisation de l'aléatoire pour entraîner les modèles. Cependant, les bonnes pratiques pour la reproductibilité dans le monde de l'apprentissage artificiel ne sont pas si bien répandues. Il peut être difficile d'obtenir des séquences de nombres pseudo-aléatoires reproductibles pour les scientifiques utilisant des frameworks de haut niveau (Onose 2023). De plus, les frameworks d'apprentissage profond eux-mêmes subissent des mises à jour et des changements de version, ce qui peut introduire des écarts lors de la comparaison des résultats obtenus avec différentes versions de logiciels.

L'apprentissage artificiel est souvent associé au big data, qui peut également présenter des défis pour la reproductibilité, principalement en raison de la variabilité des ensembles de données ou de l'échantillonnage des données. À mesure que la taille des ensembles de données augmente, les techniques de filtrage et d'échantillonnage deviennent de plus en plus importantes. Différentes approches d'échantillonnage peuvent entraîner des variations dans les sous-ensembles de données utilisés pour l'analyse, et cela peut potentiellement impacter la reproductibilité des résultats. Lorsque nous considérons la classification, les résultats numériques exacts sont beaucoup moins importants. Cependant, le traitement parallèle et les systèmes distribués, couramment utilisés dans l'analyse de masses de données, peuvent

contribuer à un comportement non déterministe, rendant la reproductibilité difficile. Les algorithmes parallèles peuvent présenter des comportements différents en fonction de facteurs tels que l'ordre d'exécution ou l'allocation des ressources informatiques. Par conséquent, obtenir les mêmes résultats à travers différentes exécutions parallèles, simplement à des fins de débogage, peut devenir très difficile. De plus, lorsqu'on traite d'une quantité massive de données, il devient plus difficile de partager de telles données, et cela affecte la capacité d'autres chercheurs à reproduire les résultats. Avec (Collberg et Proebsting 2016), nous avons vu que les recherches financées par des institutions publiques sont plus facilement reproductible car le partage de code est une pratique plus courante. Les entreprises privées ont un poids lourd dans le monde de l'apprentissage automatique et sont moins enclines à partager leur code et leurs données puisqu'elles sont en concurrence pour obtenir un avantage industriel.

Enfin, beaucoup de ressources informatiques sont maintenant proposées par les unités de traitement graphique de type GPU (Graphical Processing Unit) qui sont devenues généraliste (GP-GPU : General Purpose-GPU). Ce matériel est largement utilisé dans le domaine de l'IA, en particulier pour l'entraînement de modèle, car les GPU modernes intègrent un grand nombre de « Tensor cores ». Lors de l'utilisation des GPU pour le calcul à hautes performances, la reproductibilité peut faire face à plusieurs défis. La précision en virgule flottante est moins importante car l'entraînement peut souvent être réalisé en demi-précision (FP16) ou parfois moins, les opérations de classification n’ayant pas les mêmes besoins en termes de précision. Cependant, des erreurs d'arrondi non déterministes peuvent survenir lors des calculs numériques. Cela peut entraîner de légères variations dans les résultats même lorsque le même code et les mêmes données sont utilisés. De plus, différentes architectures de GPU peuvent produire des résultats incohérents, rendant la reproductibilité difficile sur diverses configurations matérielles. Les bibliothèques et pilotes de GPU contribuent également aux défis de reproductibilité, leur maintenabilité est réduite dans le temps (parfois seulement 5 ans). De nombreuses bibliothèques d'optimisation dans l'apprentissage artificiel et le Big Data sont spécifiques au fournisseur de matériel, s'appuyant sur des interfaces propriétaires et des fonctionnalités fournies par les fabricants de GPU. Les incompatibilités ou les différences entre les versions de bibliothèques ou les pilotes de GPU peuvent introduire des variations dans les résultats et entraver la reproductibilité. L'exécution parallèle sur les GPU peut également introduire des comportements « non déterministes » qui affectent davantage la reproductibilité. Les interférences entre les noyaux de GPU, les variations dans la planification des threads et les conditions de favorisation de la vitesse lors de l'exécution parallèle peuvent entraîner des

sorties différentes à travers différentes exécutions, même avec les mêmes données d'entrée et le même code (Taufer *et al.* 2010; Jézéquel *et al.* 2015).

I.4.7. Erreurs silencieuses (ou « soft errors »)

Rarement mentionnées, les erreurs silencieuses jouent un rôle important dans la fiabilité du calcul à hautes performances. Les perturbations de nature électrique ou magnétique au sein d'un système informatique peuvent entraîner le basculement involontaire d'un bit de la mémoire vive dynamique (DRAM : Dynamic Random Access Memory) vers son état opposé. Initialement, ce phénomène était principalement attribué aux particules alpha émises par les impuretés dans l'emballage des puces. Les particules alpha ou la radiation alpha se composent de deux protons et deux neutrons, elles sont généralement émises pendant le processus de désintégration alpha, mais peuvent aussi être produites d'autres manières (Rutherford et Royds 1908). Cependant, Normand (1996) a démontré que la principale cause d'erreurs isolées et temporaires dans les puces DRAM est liée au rayonnement ambiant. Les neutrons, provenant principalement des interactions secondaires des rayons cosmiques, ont été identifiés comme une source clé de ce rayonnement, capable d'altérer le contenu d'un ou plusieurs bits de mémoire ou de perturber les circuits responsables de leurs fonctions de lecture ou d'écriture. Une vaste étude, portant toujours sur la DRAM, a tenté de déterminer la fréquence des erreurs de mémoire (Schroeder *et al.* 2009). Dans cette étude, le comportement des erreurs de DRAM dans des scénarios réels s'écarte considérablement des hypothèses courantes. Notamment, les taux d'erreur observés de la DRAM sont nettement plus élevés que ce que l'on pensait auparavant, allant de 25 000 à 70 000 erreurs par milliard d'heures d'appareil par Mbit, et plus de 8% des modules de mémoire en ligne double (DIMM) connaissent des erreurs annuellement. Contrairement aux croyances antérieures, il est démontré que les erreurs matérielles, plutôt que les erreurs silencieuses (aussi appelées « soft errors »), dominent le paysage des erreurs de mémoire. Étonnamment, la température, qui est un facteur influent dans des environnements contrôlés, a un impact minimal sur le comportement des erreurs dans des contextes pratiques lorsqu'on tient compte d'autres variables.

Les CPU ne sont pas à l'abri de ce problème. Dans le domaine du calcul avancé, l'augmentation de la surface, la réduction des tailles de fabrication (gravures au nanomètre et en dessous dans les laboratoires) et la densité accrue des composants ont conduit à une augmentation des renversements de bits, qui peuvent causer des erreurs silencieuses. Bien que de telles anomalies soient supposées se produire plus fréquemment dans la DRAM, d'autres

composants comme les portes logiques et les unités arithmétiques des CPUs peuvent également être vulnérables (Elliott *et al.* 2013). Cette étude vise à évaluer l'impact d'un renversement de bit unique sur des opérations en virgule flottante spécifiques. L'auteur se penche sur les erreurs résultant du renversement de bits particuliers dans la représentation en virgule flottante IEEE, évitant de se fier à des informations propriétaires comme les taux de renversement de bits et les conceptions de circuits spécifiques aux fournisseurs. Dans (Dixit *et al.* 2021), ils considèrent également les CPU dans le cadre de la gestion d'erreurs matérielles. Ils argumentent : « *Par exemple, lorsque vous effectuez 2*3, le CPU peut donner un résultat de 5 au lieu de 6 silencieusement sous certaines conditions micro-architecturales, sans indication de la mauvaise opération dans les journaux d'événements ou d'erreurs du système. En conséquence, un service utilisant le CPU peut ne pas être conscient de l'inexactitude des calculs et continuer à consommer les valeurs incorrectes dans l'application*. »

De ces études, nous savons que la DRAM et les CPUs peuvent conduire à ce que nous appelons des erreurs silencieuses. Alors que la technologie évolue, en travaillant avec des puces toujours plus petites, nous augmentons la probabilité d'avoir des renversements de bit. Maintenant, à l'échelle Exascale, nous avons des superordinateurs comme Frontier qui possède 8 699 904 cœurs. À cette échelle, le temps moyen avant défaillance tombe à quelques heures de calcul seulement. Dans la section suivante, nous verrons les solutions proposées pour pallier ces problèmes.

I.5. Des oppositions au mouvement de la recherche reproductible

Cependant, tous les auteurs ne sont pas en accord avec le mouvement de la recherche reproductible. Dans son article de 2018 (Drummond 2018), Drummond affirme que le partage du code source utilisé pour produire un article n'est pas nécessaire. Il pense que les chercheurs sont forcés de le faire pour éviter d'être mal vus, mais que cela n'a aucun intérêt pour la Science. Pour lui, le mouvement de la recherche reproductible n'est pas basé sur des faits mais seulement sur des sentiments. Il ajoute que l'obligation de fournir le code source conduira à l'acceptation, par les journaux ou les conférences, d'articles sur la base de critères techniquement faibles et que, pour lui, la fraude a toujours existé et n'a jamais posé de problème significatif. Cependant, Drummond soutient toujours le concept de science ouverte. Nous ne sommes pas d'accord avec les déclarations de Drummond : partager le code et les données est maintenant devenu simple avec la multitude d'outils à notre disposition. De plus, pourquoi devrions-nous automatiquement

faire confiance à un article publié ? Nous devrions pouvoir vérifier que ce qui a été publié n'est pas entaché d'erreurs. Un cas dont nous avons précédemment discuté a montré comment une erreur sur Excel (au minimum) dans un article de recherche invité publié par des scientifiques de renom, a impacté les politiques économiques de nombreux pays (Herndon *et al.* 2014). S'il est heureusement très rare d'observer des fraudes, les humains peuvent toujours commettre des erreurs. Il existe par ailleurs un potentiel pour que les auteurs sélectionnent volontairement les données qui les arrangent, ou commettent des erreurs involontaires. En effet, un nombre accru de relecture et de reproduction ne peut qu'améliorer la détection de telles erreurs. Un code sans bug est un code qui n'a pas été suffisamment testé. Enfin, concernant l'affirmation selon laquelle les fraudes n'ont pas posé de problèmes significatifs, nous pensons que cela est sujet à débat. De plus, en ce qui concerne la recherche financée par des fonds publics, vraisemblablement financée par les contribuables, il semble y avoir un impératif éthique pour assurer une accessibilité complète pour les citoyens aux résultats. Dans un deuxième article, Drummond (2019) critique fortement la priorisation des reproductions d'articles par rapport à la nouveauté. Cela va à l'encontre du sentiment prédominant, et par conséquent, il s'oppose aux politiques actuelles des journaux et des conférences qui tendent vers la volonté de publier des articles plus reproductibles. Il y a aussi Fanelli (2018) qui affirme que la crise de la reproductibilité est largement exagérée, voire fausse, et que pour lui, ce récit est nuisible car il démotive les jeunes chercheurs. Pourtant, être rigoureux fait partie de notre travail, et nous ne voyons pas pourquoi cela dissuaderait les jeunes chercheurs.

Un grand nombre de chercheurs sont fortement en désaccord avec les voix s'opposant au mouvement de la recherche reproductible, bien qu'utiles pour remettre en question la pertinence de telle ou telle approche, que nous avons citées ci-dessus (Fanelli et Drummond en 2018 et 2019). Stodden *et al.* (2013), Bajpai *et al.* (2017), Pouzat (2022) dans son dialogue humoristique, Rougier, Hinsen et Barba, pour ne citer qu'eux, via la création de leur journal dédié à la reproduction d'articles (Rougier *et al.* 2017), National Academies of Sciences (2019), Ten Hagen (2016), et d'autres, défendent l'idée que les journaux devraient encourager la recherche reproductible. Barba (2018) est beaucoup moins indulgente envers Drummond, déclarant que l'écart entre les définitions de la reproductibilité et de la réplicabilité du domaine de l'informatique par rapport aux autres domaines scientifique était dû à Drummond, après son article de 2009 : « Ils ont [les chercheurs en informatique], à leur tour, basé leurs définitions sur le travail emphatique mais essentiellement erroné de Drummond (2009). ». Nous croyons que le scepticisme est une marque de fabrique d'un scientifique compétent. Promouvoir une culture

du doute et de scepticisme est crucial, mais il est nécessaire de faire preuve de prudence, car l'utilisation abusive du doute « scientifique » a également été employée pour entraver la reconnaissance de découvertes révolutionnaires, comme ce fut le cas avec le lien entre les cigarettes et le cancer du poumon.

I.6. Conclusion

Nous avons vu les différentes raisons pour lesquelles nous pourrions perdre la reproductibilité, en détail dans le cadre de l'informatique générale, utilisé dans d'autres sciences, et dans le cadre du calcul à haute performance, qui peut également être utilisé dans d'autres sciences.

En examinant les définitions et les nuances de la recherche reproductible, nous avons exploré la crise multifacette de la reproductibilité qui s'étend à divers domaines scientifiques, mettant en évidence les défis spécifiques et les opportunités présentées par le HPC. Nous avons analysé les facteurs contribuant à la perte de reproductibilité, de la science ouverte et de la documentation à l'ingénierie logicielle et à la complexité des workflows. De plus, nous avons commencé une réflexion sur les problèmes de reproductibilité uniques au HPC : les problèmes liés au calcul parallèle, la génération de nombres aléatoires dans les simulations de Monte Carlo parallèles, l'optimisation, l'hétérogénéité matérielle, et les domaines émergents du calcul quantique et de l'apprentissage automatique. Nous allons maintenant voir quelles solutions existent déjà pour faire face aux problèmes soulevés.

II. Exemple de solutions mises en place pour répondre à la crise de la reproductibilité en informatique

II.1. Introduction

Dans ce chapitre, nous explorerons les solutions conçues pour améliorer la reproductibilité, telles que le suivi de versions, la programmation lettrée et la gestion avancée des workflows, tout en abordant également les défis spécifiques au HPC tels que la reproductibilité des calculs avec des nombres à virgule flottante et la gestion des erreurs silencieuses.

Les outils de suivi de gestion et d'archivage de code sont des outils omniprésents dans le monde du génie logiciel. Dans cette introduction, nous pouvons évoquer Git comme outil incontournable de la gestion de versions, et Zenodo ou Software Heritage pour l'archivage des versions du code et des données. La programmtion lettrée et la gestion des workflows ont également mené au développement de nombreux outils. Nous pouvons citer Jupyter et Org-mode, deux outils extrèmement plébicités et utilisé par les chercheurs dans ce contexte. Il est également nécessaire de contrôler l'environnement logiciel afin de pouvoir le reproduire. Cela passe par des outils simples comme « apt » ou « pip », et de manière plus rigoureuse avec des outils comme des machines virtuelles, des conteneurs, ou des outils plus encore plus spécifiques et performants en termes de reproductibilité comme Guix.

Les problématiques de reproductibilité propres au calcul à haute performance ont également donné lieu au développement de solutions. Concernant la reproductibilité des calculs avec des nombres à virgule flottante, des solutions algorithmiques ont été développées, ainsi que des solutions logicielles, avec des outils permettant d'enregistrer et de rejouer l'execution d'un programme, initialement conçu pour déboguer, cela peut aussi concerner la répétabilité. Enfin, nous évoquons les erreurs silencieuses qui provoquent par exemple le changement de valeur de certains bits de façon inopinée suite à la collision avec une particule chargée énergétiquement. Les mémoires avec codes de correction d'erreur et plus globalement, la redondance du matériel, permettent de contrôler ce genre d'erreurs.

II.2. Quelles solutions pour améliorer la reproductibilité ?

II.2.1. Gestion des versions et archivage

Pour garantir la reproductibilité des parties computationnelles d'un article, il est nécessaire d'effectuer un suivi de version, d'archiver, de documenter et de partager le code. Tout d'abord, le suivi de version est effectivement très important lorsqu'il s'agit de codes et de scripts. Imaginez un scénario avec une expérience informatique, aboutissant à des figures intéressantes. Ensuite, vous continuez à mettre à jour votre code. Plus tard, vous voulez retravailler sur ces figures, peut-être parce que vous avez soumis un article de recherche et que les relecteurs vous l'ont demandé, ou peut-être juste dans votre processus de recherche afin de mieux comprendre les figures produites. Comme vous avez modifié le code, vous êtes désormais incapable de générer les mêmes figures, vous n'obtenez plus les mêmes résultats intéressants et vous n'avez pas l'ancienne version du code. Cette situation n'est pas rare et aide à comprendre le besoin d'outils pour contrôler les versions du code. Deuxièmement, il est crucial d'archiver votre code de manière permanente. Le code d'erreur HTML « 404 not found » apparaît bien plus souvent qu'il ne le devrait, nous devons faire attention à l'endroit où nous stockons nos différentes versions de code. Troisièmement, vous devez documenter votre code afin que d'autres personnes puissent l'utiliser et le comprendre. Et enfin, vous devez partager votre code pour que les gens puissent y accéder (Kitzes *et al.* 2018).

Concernant les outils de suivi de version, deux des plus importants sont Git et Apache Subversion. Git est devenu l'outil dominant dans ce domaine et est largement utilisé pour le suivi de version. Git offre de nombreuses fonctionnalités, dans le cadre d'une recherche reproductible, nous préconisons son intégration dans le processus de développement, bien que les débutants puissent avoir besoin d'une phase d'apprentissage initiale. En essence, adopter Git représente une des bases de bonnes pratiques en génie logiciel.

Plusieurs plateformes sont disponibles pour l'archivage du code. Github et Gitlab sont parmi les plus courantes. Ils fonctionnent comme des dépôts, où l'on peut stocker son code. Lors du développement d'un code pour un projet de recherche, l'utilisation de l'un de ces outils, en conjonction avec Git, est presque obligatoire. Cependant, ces outils n'ont pas été initialement conçus pour la recherche reproductible. Il convient de noter que ces plateformes, étant des entités commerciales (dans le cas de Github), ne garantissent pas l'accessibilité perpétuelle du code. Des alternatives comme Zenodo et Software Heritage offrent des solutions plus robustes pour l'archivage du code (Hinsen 2020b). Développés respectivement par le CERN et Inria,

Zenodo et Software Heritage fournissent des moyens d'archiver du code, des données ou toute ressource numérique associée à un article, chacun identifié par un ID unique (soit DOI ou SWHID). Zenodo est davantage orienté vers l'archivage du code et des données d'un article, tandis que Software Heritage est un projet plus vaste, comprenant le stockage de toutes les versions existantes de code open source dans le monde, telles que les bibliothèques et non seulement les artefacts d'articles de recherche.

II.2.2. Programmation lettrée et documentation

Pour résoudre le problème de la documentation (rendre le code aussi clair que possible) et, peut-être aussi un peu, pour résoudre le problème des workflows, la programmation lettrée a été introduite. Il est essentiel, lors du partage de code, de le rendre accessible (dans le sens de la facilité d'utilisation). À cette fin, le concept de Programmation lettrée (Literate Programming) a émergé en 1984 avec Knuth (1984). Le principe est d'entrelacer le code avec du texte en langage naturel afin de rendre l'ensemble du contenu plus compréhensible. L'application actuelle de ce principe prend la forme de Notebooks : des documents interactifs qui permettent de combiner du texte avec des blocs de code. Ils se présentent sous différentes formes pour différents langages de programmation. Pour faciliter la rédaction de ce type de fichier, des langages de composition de texte sont utilisés : principalement LaTeX et Markdown, ce dernier étant beaucoup plus léger et compréhensible sans interpréteur, comme discuté par Pouzat (2021). Knuth avait développé l'outil initial pour la programmation lettrée, appelé WEB, qui comprenait deux programmes principaux, Tangle et Weave (Knuth 1984). Ce système était adapté pour le langage de programmation Pascal et générait des documents formatés en utilisant TeX. Plus tard, Knuth et Levy (1994) ont créé une version pour le langage C, nommée cweb. Une évolution contemporaine de ces outils est noweb (Johnson et Johnson 1997), conçue pour être adaptable à différents langages. Ses programmes principaux, notangle et noweave, sont tous deux codés en C. Les documents produits via noweave peuvent être formatés en utilisant TeX, LaTeX ou troff, ou même être affichés dans un navigateur web sous forme HTML. Des utilitaires logiciels comme WEB, cweb et noweb permettent aux auteurs de créer à la fois du contenu écrit et du code, mais ils ne disposent pas de mécanismes pour exécuter directement le code dans les documents. Au lieu de cela, le code destiné à l'exécution est extrait, résultant en des fichiers de code source qui sont ensuite transmis à un compilateur ou interpréteur. Un des outils le plus largement adopté pour réaliser une recherche reproductible était Sweave, qui voit une utilisation croissante au sein de la communauté de programmation R. L'utilisation de Sweave pour la recherche reproductible a donné naissance à des outils analogues comme

SASweave et Statweave. Certains de ces outils sont destinés à des langages statistiques autres que R et sont compatibles avec d'autres systèmes d'écriture de documents que LaTeX, englobant des formats comme les formats OpenDocument et Microsoft Word (Lenth et Højsgaard 2007; Baier et Neuwirth 2007; Lenth 2012). Aujourd'hui, parmi les outils de programmation littéraire les plus connus, nous pouvons mentionner :

- Jupyter (Kluyver *et al.* 2016), un notebook principalement utilisé avec le langage Python.
- Rstudio (Allaire 2012), développé principalement pour le langage R.
- Org-mode (Schulte *et al.* 2012), un outil beaucoup plus polyvalent, utilisable avec tous les langages.

Ces trois outils sont présentés en détail dans le MOOC de recherche reproductible de l'Inria (Pouzat *et al.* 2018) animé par Arnaud Legrand, Konrad Hinsen et Christophe Pouzat. L'utilisation de notebook et, plus largement, la programmation lettrée, est vivement encouragée par la communauté scientifique. Stanisic *et al.* (2015) proposent une solution de workflow pour la recherche reproductible basée sur l'utilisation de Git et Og-mode. Desquilbet *et al.* (2019) discutent de l'utilisation des notebooks dans leur livre sur la recherche reproductible. Stodden *et al.* (2013) compilent une liste d'outils pour la recherche reproductible, les notebooks en faisant partie intégrante. Ragan-Kelley *et al.* (2018) proposent même un outil nommé Binder, permettant l'exécution directe des dépôts de notebooks Jupyter sur le web sans installation, en reliant à un dépôt Git. Delescluse *et al.* (2012) mentionnent également Org-mode et Sweave pour la recherche reproductible, dans le contexte du langage R.

Avec l'augmentation de l'utilisation du langage de programmation Python, Jupyter est probablement le notebook le plus utilisé et le plus facile à prendre en main. Cependant, pour sa polyvalence, Org-mode est probablement l'outil le plus plébiscité, en particulier pour les amateurs de Emacs, car Org-mode se base sur cet éditeur de texte (Stallman 1981).

Lorsque l'on traite de grands projets logiciels complexes, on ne peut pas les coder entièrement en utilisant la programmation lettrée. Dans ce cas, le projet sera codé en utilisant un schéma de développement plus classique (génie logiciel), et on utilisera la programmation lettrée en finalité pour l'exécution ou l'analyse des résultats et des données. Par conséquent, pour traiter de tels projets, vous aurez besoin d'autres outils de documentation que la simple programmation lettrée. Il n'y a pas qu'une seule manière de faire une bonne documentation (Sommerville 2001), et vous aurez besoin de bonnes pratiques issues du génie logiciel. Une

manière répandue de documenter du code orienté objet est de produire des diagrammes pour modéliser le code en utilisant le langage de modélisation unifié (UML) (Booch *et al.* 1996).

En plus de la programmation lettrée et d'une documentation plus complète avec des outils de modélisation, l'adhésion aux principes FAIR peut contribuer à renforcer la reproductibilité dans la recherche scientifique. Ces principes plaident en faveur de la facilitation de la recherche d'éléments numériques tels que les données, les algorithmes et les outils, en les rendant facilement trouvables, accessibles, interopérables et réutilisables. La mise en œuvre de ces principes garantit que les résultats de la recherche sont identifiables grâce à des identifiants uniques et des métadonnées riches, accessibles via des protocoles bien définis, interopérables grâce à l'utilisation de formats et de vocabulaires standard, et réutilisables avec des licences d'utilisation claires. L'utilisation des principes FAIR facilite la reproduction des résultats de recherche et favorise également la collaboration et l'innovation en permettant aux chercheurs de partager facilement et de s'appuyer mutuellement sur leur travail (Wilkinson *et al.* 2016).

II.2.3. Workflow

Le workflow d'un article de recherche est une partie importante de la recherche reproductible. Il est essentiel de comprendre l'ordre dans lequel les opérations doivent être exécutées. L'un des premiers outils à émerger est le fichier makefile. L'outil make associé a permis l'automatisation de la construction d'un exécutable, mais pas seulement. Le premier article à le considérer à cette fin est paru en 2000 (Schwab *et al.* 2000), avec Jon Claerbout, que nous avions déjà vu comme l'un des premiers contributeurs de la recherche reproductible en 1992. Schwab *et al.* décrivent comment utiliser « make » et des outils associés, ainsi que des conventions de nommage pour les fichiers, afin de garantir que les lecteurs ainsi que les auteurs puissent être capable de reproduire les calculs. Avec (Cohen-Boulakia *et al.* 2017), nous pouvons voir que la biologie est l'un des domaines qui possède le plus d'outils de gestion des workflows pour améliorer la reproductibilité, peut-être parce que la biologie est un domaine qui produit et analyse beaucoup de données. Dans (Oinn *et al.* 2004), nous pouvons lire « les expériences in silico en bioinformatique impliquent l'utilisation conjointe d'outils informatiques et de stockage d'informations. Un nombre croissant de ces ressources est mis à disposition avec un accès sous forme de services Web. Les scientifiques en bioinformatique devront orchestrer ces services Web dans des workflows dans le cadre de leurs analyses ». De nombreux outils de workflow ont été créés depuis le fichier makefile en 2000 (plus de 300) (Amstutz *et al.* 2024). Ils visent à être faciles d'utilisation, souvent en fournissant une interface graphique (GUI).

Taverna (Oinn *et al.* 2004) était un système de gestion de workflow scientifique open source qui fournit un environnement pour la conception de workflows et un moteur d'exécution de ces workflows. Taverna optimise la structure du workflow, relie les services Web et les activités, et prend en charge l'exécution sur des grilles de calcul ou du cloud computing. Cependant, le projet est désormais abandonné. Galaxy (Giardine *et al.* 2005) est un système de workflow populaire dans la communauté bioinformatique qui utilise depuis peu un format de spécification de workflow JSON ou YAML. Galaxy fut compatible avec Common Workflow Language (CWL), mais ce n'est plus le cas (Amstutz *et al.* 2016). Il fournit une interface graphique pour explorer et partager les exécutions de workflows. Galaxy gère les dépendances des activités pour la parallélisation, génère et surveille les tâches, et utilise une planification dynamique pour la distribution des tâches. OpenAlea (Pradal *et al.* 2015) est un système de workflow scientifique open source lié à un outil de modélisation basé sur le langage Python. Il permet aux utilisateurs d'exporter des workflows dans un format CWL. Cependant, il présente des limites en termes de complexité, de manque d'un dépôt centralisé et de difficulté à visualiser la provenance des données. Nextflow (Di Tommaso *et al.* 2014) est un système de workflow en ligne de commande pour la gestion de workflows scientifiques parallèles complexes. Il utilise une spécification basée sur du texte et prend en charge des workflows de processeurs et opérateurs. Il présente des limites en termes de gestion des dépendances des outils et de manque d'informations de provenance de données. Pegasus, développé par Deelman *et al.* en 2004 (Deelman *et al.* 2004), est un outil utilisé dans plusieurs domaines scientifiques. Il offre plusieurs caractéristiques importantes comme la capacité de s'adapter à différents environnements (portabilité), l'utilisation d'algorithmes avancés pour organiser les tâches de manière optimale, le passage à l'échelle, la gestion efficace des informations sur l'origine des données, et la tolérance aux pannes. Pegasus est composé de cinq éléments principaux : un moteur qui associe les tâches aux ressources disponibles, un système pour exécuter les tâches sur l'ordinateur local, un ordonnanceur de tâches, un moteur d'exécution à distance, et un dispositif pour surveiller le processus. Ensemble, ces éléments permettent de créer et d'exécuter un workflow efficace sur différentes plateformes informatiques. Swift (Zhao *et al.* 2007), tout comme Pegasus, est utilisé dans de nombreuses disciplines et se spécialise dans les workflows qui nécessitent une grande quantité de données. Il gère ces workflows en plusieurs étapes : définition du programme, planification, exécution, gestion de la provenance et approvisionnement. Swift supporte la division des workflows, la résilience aux erreurs, et l'allocation dynamique de ressources sur divers sites d'exécution. Kepler (Altintas *et al.* 2004) est un système de gestion de workflow qui permet différentes méthodes d'exécution. Il intègre

une interface graphique. Kepler est capable de planifier de manière statique ou dynamique, offre une tolérance aux pannes et peut exécuter des workflows sur des services Web, des systèmes basés sur des grilles de calcul ou utilisant le framework Hadoop (Borthakur 2007). Chiron (Ogasawara *et al.* 2013) adopte une approche basée sur une base de données pour gérer l'exécution parallèle de workflows scientifiques intensifs en données. Il utilise un modèle algébrique et des opérateurs pour gérer les données et les activités du workflow. Chiron assure le suivi du workflow, gère différents types de parallélisme (par les données, indépendant, en pipeline) et propose une planification dynamique. Il stocke les informations d'exécution et de provenance dans une base de données structurée. Triana (Taylor *et al.* 2007) est un outil doté d'une interface graphique, développé initialement pour l'analyse de données dans le projet GEO 600. Triana est conçu pour les applications distribuées. Enfin, Snakemake est un moteur de workflow avancé offrant un langage de définition de workflow basé sur Python, facile à lire. Il permet une exécution flexible, allant d'un seul cœur à la distribution de calculs parallèles sur des clusters avec de nombreux cœurs, sans nécessiter de modifier le workflow. Cet outil est particulièrement réputé dans le domaine de la bioinformatique (Köster et Rahmann 2012).

Taverna, Chiron, Triana et Galaxy ont des fonctionnalités telles que l'interface graphique pour la conception des workflows, la prise en charge du parallélisme, la planification dynamique et l'exécution de workflows dans des environnements de grille de calcul et de cloud computing. Taverna prend en charge le partage d'informations sur les workflows, tandis que Galaxy se spécialise dans l'exécution de workflow en bioinformatique. Sequanix propose une interface graphique dynamique pour Snakemake (Desvillechabrol *et al.* 2018). Un état de l'art sur la gestion des systèmes de workflow est disponible (Liu *et al.* 2015), et un article sur les workflows plus axés sur les sciences de la vie a également été publiée en 2017 (Cohen-Boulakia *et al.* 2017), tandis que dans (Ivie et Thain 2018), Ivie et Thain proposent un état de l'art plus général sur la gestion des workflows en recherche reproductible.

II.2.4. Environnement logiciel

II.2.4.1. Gérer les dépendances

Lorsque vous écrivez un programme, vous vous reposez toujours sur votre environnement logiciel. À la fin, lorsque vous publierez vos résultats, d'autres chercheurs pourraient rencontrer des obstacles pour relancer votre programme. Il se peut qu'ils ne puissent pas reproduire vos résultats car ils n'ont pas le même environnement logiciel que vous. Avoir une bonne

documentation ne suffit pas, tout comme décrire le workflow ou archiver l'historique des versions du code.

Plusieurs solutions existent pour gérer les dépendances logicielles. Par exemple, l'utilisation de « apt-get » sur les distributions Linux, ou l'utilisation de « pip » lors de travaux avec Python. Apt-get est un système de gestion de paquets qui permet l'installation, la mise à jour et la suppression facile de paquets logiciels. Il résout automatiquement les dépendances, garantissant que toutes les bibliothèques et composants logiciels nécessaires sont installés. Pip, quant à lui, est un installateur de paquets pour Python qui simplifie la gestion des bibliothèques Python et de leurs dépendances. Il permet aux utilisateurs d'installer, de mettre à jour et de supprimer facilement des paquets Python à partir de l'Index des paquets Python (PyPI). Mais ils nécessitent tous les deux l'installation manuelle des paquets (même s'ils peuvent gérer par eux-mêmes les dépendances entre ces paquets). Même si de tels outils sont bons lorsqu'ils sont utilisés seuls, ils ne sont pas adaptés à la recherche reproductible. Le langage Python est désormais adopté pour de nombreuses expérimentations scientifiques. Il existe des outils spécifiques pour Python tels que Conda, qui est un gestionnaire de paquets et d'environnements couramment utilisé dans la communauté scientifique Python. Conda permet l'installation à la fois de paquets Python et de paquets non-Python, ce qui le rend utile pour gérer des environnements logiciels complexes avec des dépendances qui vont au-delà des bibliothèques Python. Enfin, une fonctionnalité plus avancée vous permet de créer un environnement virtuel. Venv est un outil qui aide à créer des environnements Python isolés. Il permet aux utilisateurs de créer des environnements distincts pour différents projets, évitant ainsi les conflits entre les dépendances. Lorsqu'un nouvel environnement virtuel est créé, il est livré avec sa propre copie de l'interpréteur Python et une copie locale de l'outil pip. Cela garantit que toutes les bibliothèques installées dans l'environnement virtuel sont isolées de l'environnement Python global, ce qui facilite la gestion et la reproduction des dépendances spécifiques au projet. Venv est particulièrement utile lorsque l'on travaille sur plusieurs projets ou que l'on teste différentes versions de bibliothèques.

Bien que des outils tels que apt-get, pip, Conda et Venv soient largement utilisés et offrent des moyens pratiques de gérer les environnements logiciels et les dépendances, ils présentent des limites par rapport à des outils plus spécifiques conçus pour la reproductibilité. Tout d'abord, comme ils ne sont pas conçus pour la reproductibilité, la stabilité des environnements logiciels peut être difficile à atteindre avec ces outils, car les versions exactes et les configurations des paquets ne sont pas forcément explicitement capturées. Cela peut entraîner

une perte de reproductibilité. Par exemple, si vous utilisez une bibliothèque telle que Pandas ou Numpy en Python, les développeurs pourraient modifier certaines parties du code sans changer le numéro de version, de sorte que des outils comme pip (et ceux qui en dépendent) ne pourraient pas clairement identifier les deux versions du code. Une autre faiblesse de ces outils est qu'ils nécessitent souvent des privilèges d'administration pour installer des paquets à l'échelle d'un système d'exploitation, ce qui n'est pas adapté à un environnement de calcul à hautes performances, où vous n'aurez pas les privilèges administrateurs. Venv offre un certain niveau d'isolation, mais il se limite aux bibliothèques Python uniquement.

Pour ces raisons, certains outils ont émergé pour fournir une solution satisfaisante pour améliorer la reproductibilité de la partie computationnelle d'un article de recherche. CDE (Guo et Engler 2011) est un outil conçu pour résoudre les problèmes de dépendance. Il regroupe le code, les données et l'environnement. Aucun accès root n'est nécessaire. Voici comment fonctionne l'outil : il utilise ptrace et fait un paquetage automatisé du code, des données et de l'environnement pour s'exécuter sur une machine Linux. Un problème connu est qu'il pourrait ne pas inclure toutes les dépendances, qui devraient être ajoutées manuellement. De plus, il ne fonctionne qu'avec des noyaux Linux compatibles. Il fonctionne de la manière suivante: Alice lance la commande « cde script.py data.dat ». Cette action exécute le script Python, capture des informations via ptrace et crée un répertoire « cde-package ». Alice compresse le répertoire et l'envoie à Bob. Bob exécute ensuite la commande « cde-exec script.py data.dat ». Au lieu de rechercher des bibliothèques dans le chemin de Bob, cde-exec utilise ptrace ou strace pour modifier l'utilisation des bibliothèques et utilise celles stockées dans le dossier envoyé par Alice. Il peut y avoir une perte de performance potentielle allant de 2% à 28%. Cette solution peut ne pas être adaptée à un environnement HPC en raison de la perte potentielle de performance. L'outil Sumatra (Davison 2012) améliore la recherche reproductible en fournissant un système de capture automatisée des métadonnées. La bibliothèque principale de Sumatra met en œuvre des fonctionnalités telles que la capture de l'environnement matériel et logiciel, la capture des données d'entrée et de sortie, et la capture du contexte scientifique. Cette fonctionnalité de base peut être utilisée par différentes interfaces qui couvrent les différentes méthodes de travail (ligne de commande, interface graphique, etc.). Les outils Sumatra n'ont pas nécessairement besoin de capturer toutes les informations pour être efficaces, plus il y a d'informations enregistrées, plus il est facile de reproduire les résultats à l'avenir. Sumatra est conçu pour être facile à utiliser et ne pas ralentir le workflow habituel des scientifiques. Les outils de Sumatra se composent d'une bibliothèque de base implémentée en Python, d'un outil

en ligne de commande et d'une interface basée sur un navigateur web. L'outil en ligne de commande permet de capturer le contexte de calcul, les entrées et les sorties, ainsi que de visualiser les calculs précédents. L'interface basée sur un navigateur web offre des fonctionnalités supplémentaires pour visualiser, rechercher et annoter les enregistrements de calcul. De même que CDE, ReproZip (Chirigati *et al.* 2016) est un outil qui vise à rendre les expériences computationnelles reproductibles sur différentes plates-formes et ceci longtemps après leur création. Il capture automatiquement la provenance d'une expérience en traçant les appels système, en utilisant ptrace, et utilise ces informations pour créer un package reproductible léger qui inclut uniquement les fichiers nécessaires à sa reproduction. ReproZip ajoute une fonctionnalité pour être compatible non seulement avec les systèmes Linux, mais aussi avec Windows ou MacOS. Ses limites concernent les environnements distribués, tels que les clusters MPI ou Hadoop. Cependant, ni Sumatra ni ReproZip ne fournissent une analyse des performances, comme le fait CDE. Comme ReproZip utilise ptrace de même que CDE, une hypothèse plausible serait que les performances devraient être similaires. En fin de compte, ces outils sont conçus pour la recherche reproductible, mais ne sont probablement pas adaptés au calcul à hautes performances.

Étant donné que la manipulation de tels outils nécessite une phase d'apprentissage, l'objectif final serait de trouver un outil qui puisse être utilisé par le plus de monde possible et qui soit adapté au calcul à hautes performances. Pour gérer les dépendances de bibliothèques, l'outil que nous recommandons est Guix (Courtès et Wurmus 2015). Guix est un outil de gestion de logiciels qui adopte une méthode fonctionnelle pour gérer les dépendances des logiciels. Selon cette méthode, les étapes de construction et d'installation des logiciels sont considérées comme des fonctions pures. Autrement dit, les résultats de ces étapes dépendent uniquement des données fournies en entrée. Cette technique permet de stocker efficacement les résultats sur le disque dur, assurant que les mêmes données d'entrée produiront toujours les mêmes résultats. Pour appliquer cette méthode, Guix exerce un contrôle rigoureux sur l'environnement de construction. Inspiré par Nix (Dolstra *et al.* 2004), Guix utilise un service système (daemon) avec des privilèges administrateurs pour créer des logiciels dans des conteneurs Linux isolés, notamment dans un environnement chroot. Ces conteneurs disposent de leurs propres identifiants utilisateurs et espaces de noms isolés pour différents aspects tels que la gestion des processus, la communication entre processus, le réseau, etc. L'environnement chroot se limite aux répertoires spécifiquement déclarés, empêchant ainsi l'accès à des outils ou bibliothèques non autorisés durant la construction. Les espaces de noms isolés empêchent également toute

communication avec l'extérieur durant la construction. Les résultats de chaque construction sont enregistrés dans un espace de stockage commun, habituellement situé dans /gnu/store. Chaque élément dans ce stockage porte un nom comprenant un hash basé sur toutes les données d'entrée utilisées lors de la construction. Ce hash inclut non seulement les compilateurs et les bibliothèques, mais aussi les scripts de construction et les variables d'environnement. Ce processus tient compte de l'ensemble des dépendances de chaque logiciel, incluant récursivement les outils et bibliothèques utilisés. Cette méthode garantit la reproductibilité des constructions, permettant ainsi au système de retracer l'intégralité du réseau de dépendances pour chaque logiciel construit.

Nous avons constaté que Guix est probablement le meilleur outil pour gérer les dépendances. De plus, cet outil ne fournit pas seulement un fichier « binaire » illisible à exécuter pour reproduire n'importe quel traitement (mais dans ce cas, nous ne savons pas vraiment ce que nous exécutons), mais il offre une vision claire et complète de l'environnement logiciel que nous utilisons. Chaque version est identifiée par un hash unique, avec la source disponible sur un dépôt en ligne Guix. Un tutoriel clair et facile à utiliser a été proposé en 2023, il permet de produire des articles de recherche reproductibles. Nous recommandons la lecture de ce tutoriel. Les inconvénients de Guix sont qu'il n'est pas encore très répandu sur les clusters de calcul, et que le « daemon » s'exécutant en fond nécessite des privilèges administrateurs, de sorte que la première configuration et l'installation de Guix doivent être effectuées par un administrateur système du cluster. Lorsque vous travaillez localement avec Guix, il vous permet, si les futurs utilisateurs n'ont pas Guix, de générer une image Docker ou une image Apptainer (nouveau nom du container Singularity adapté au calcul à hautes performances). Nous présenterons ces deux outils par la suite.

II.2.4.2. Machines virtuelles

Pour rendre la partie informatique de vos recherches plus reproductible, un autre type d'outil existant est celui des machines virtuelles (VM). Une des premières mentions détaillées de ce concept remonte à une cinquantaine d'années (Goldberg 1974). Avec les machines virtuelles modernes, nous utilisons un hyperviseur comme plateforme de virtualisation permettant à plusieurs systèmes d'exploitation de fonctionner simultanément sur une seule machine physique. Il existe actuellement deux types d'hyperviseurs, comme décrit dans la Figure 6, natifs et hébergé (hosted). L'hyperviseur natif, également connu sous le nom de « bare metal », est un logiciel qui s'exécute directement sur le matériel; l'hyperviseur natif interagit directement avec le matériel de l'hôte, fournissant une plateforme sur laquelle plusieurs

systèmes d'exploitation peuvent être exécutés par l'hyperviseur. Ce type d'hyperviseur est léger et optimisé pour le noyau de l'hôte. A contrario, l'hyperviseur hébergé est un logiciel qui s'exécute à l'intérieur d'un autre système d'exploitation.

Une machine virtuelle offre un système d'exploitation indépendant, spécialement conçu pour vos expérimentations. Cette isolation permet de capturer et de reproduire fidèlement tous les processus et paramètres impliqués dans vos recherches. Cette caractéristique est particulièrement utile pour permettre à d'autres chercheurs de reproduire vos expériences dans un environnement identique. Toutefois, l'usage de machines virtuelles peut entraîner une réduction des performances, ce qui peut être un frein pour les applications nécessitant une grande puissance de calcul. Malgré cela, pour la plupart des applications, les avantages de reproductibilité et d'isolation offerts par les machines virtuelles compensent largement ce désavantage. Dans (Stodden *et al.* 2013), les auteurs mentionnent VirtualBox et VMWare comme outils permettant la création et l'utilisation de machines virtuelles pour la recherche reproductible. De même dans (Ruiz *et al.* 2014). Ces deux outils utilisent le deuxième type d'hyperviseur (hosted) qui est beaucoup plus facile à utiliser (la facilité d'utilisation étant un critère important pour la recherche reproductible). Mais celui peut aussi entraîner plus de perte de performance (ce qui n'est pas adapté au HPC). Certaines publications étudient les performances de la virtualisation de type 1, tandis que d'autres se penchent sur le type 2, souvent dans le contexte de l'informatique en nuage (ou « cloud computing »). Dans ce contexte, les machines virtuelles sont omniprésentes pour fournir un système d'exploitation utilisable à de nombreuses personnes en se basant sur le même matériel sous-jacent. Le but principal n'est donc pas ici la recherche reproductible, mais simplement proposer de l'accès à des machines via le « cloud ».

Concernant les hyperviseurs de type 1, Gilbert *et al.* (2005) évaluent les performances de la virtualisation dans le contexte des applications de physique de hautes énergies au CERN dans un environnement de grille de calcul. Leurs résultats ont montré que la virtualisation peut avoir jusqu'à 15 % de perte de performance par rapport au temps d'exécution. Ils travaillaient avec un hyperviseur de type 1, ESX VMware. Dans (Matthews *et al.* 2007), les auteurs n'ont pas trouvé que des outils de virtualisation tels que Xen ou VMware génèrent une perte de performance, tandis que dans (Padala *et al.* 2007), les auteurs trouvent que Xen génère beaucoup de perte de performance, probablement en raison d'un nombre plus élevé de perte de données dans le cache de niveau 2 (L2 cache miss). Dans (Acharya *et al.* 2018), leurs résultats montrent que sur les

processeurs x86, l'hyperviseur (en utilisant KVM) fonctionne moins bien que d'autres technologies. Cependant, ce n'est pas le cas sur les processeurs ARM.

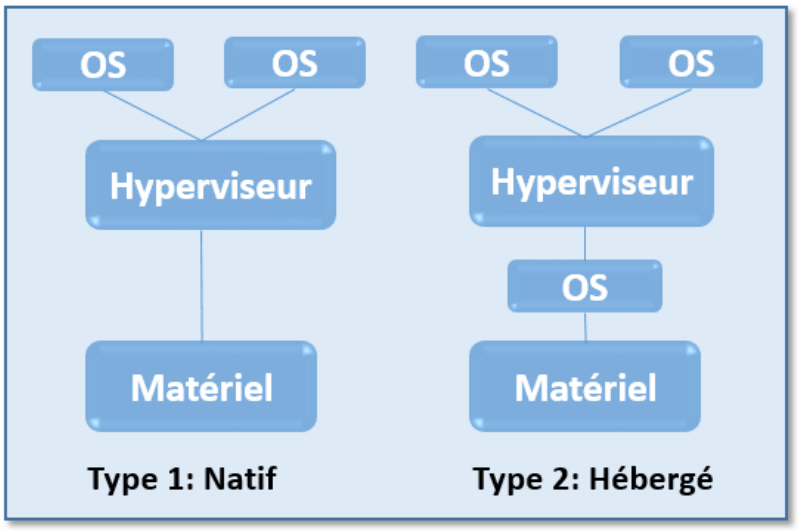


Figure 6: Hyperviseur type 1 et hyperviseur type 2

Tous les articles cités précédemment évaluent les performances des hyperviseurs natif (type 1). Et nous pouvons constater que ce n'est probablement pas une solution adaptée pour le calcul à hautes performances en raison de la perte de performance induite par leur utilisation, mais ce n'est également pas adapté à la recherche reproductible en raison de sa complexité de mise en place. Nous n'avons trouvé aucun article sur la recherche reproductible évoquant l'utilisation des technologies et outils des hyperviseurs de type 1. Cependant, pour les hyperviseurs hébergés (type 2), tels que VirtualBox et VMware (dans sa version hébergée), il existe des publications du monde de la recherche reproductible qui évoquent leur utilisation pour rendre les articles plus facilement reproductibles.

Dans (Đorđević *et al.* 2021), un article récent de 2021, les auteurs étudient les performances de VirtualBox par rapport à une exécution native, et concluent que « en ce qui concerne VirtualBox, la chute de performance est notable ». Dans (Beserra *et al.* 2015), les auteurs se situent dans le contexte du calcul haute performance, contrairement aux travaux précédents. Ils ont cherché à analyser les performances de KVM (hyperviseur de type 1) et de VirtualBox (hyperviseur de type 2). Pour le traitement, ils ont effectué trois tests différents (HPL, DGEMM et FTT). Sur le premier, KVM a atteint des performances proches des performances natives et a obtenu 14% de performance de plus que VirtualBox. Sur le deuxième, KVM a eu des performances 5% inférieures à une exécution native, mais a performé 10% mieux que VirtualBox. Sur le troisième test (FFT), KVM et la performance native ont performé 30% mieux

que VirtualBox. En ce qui concerne l'accès à la mémoire, VirtualBox a performé environ 25% moins bien que KVM et l'exécution native. Dans l'ensemble, on peut clairement voir que les hyperviseurs de type 2 ne conviennent pas au calcul à hautes performances. Un document fusionnant l'utilisation de la virtualisation et du cloud computing dans le cadre de la recherche reproductible a été proposé par Bill Howe (Howe 2012). Il a proposé d'utiliser les services d'Amazon pour collaborer via le cloud. Cependant, l'utilisation d'une entreprise privée en tant que tiers n'est pas toujours adaptée à la science ouverte internationale.

Un autre inconvénient de l'utilisation de machines virtuelles pour la recherche reproductible est leur opacité et leur demande en ressources. La distribution de gros fichiers de machines virtuelles sur les réseaux peut être fastidieuse, et le contenu d'une machine virtuelle reste obscur jusqu'à son exécution. Bien qu'elles permettent une certaine reproductibilité, les machines virtuelles ne sont pas optimales en termes de passage à l'échelle, d'adaptabilité et de performance, engendrant par ce dernier aspect un coût énergétique supplémentaire dans le contexte où cette consommation est très sensible (calcul intensif).

II.2.4.3. Conteneurs

Les conteneurs sont un autre type de virtualisation plus léger que les machines virtuelles et qui fonctionne au niveau du système d'exploitation. Les conteneurs et les technologies de virtualisation matérielles sont similaires en ce qu'ils permettent à plusieurs applications isolées de s'exécuter sur une seule machine hôte. Cependant, il existe des différences significatives dans la manière dont ils y parviennent. La virtualisation matérielle implique l'exécution d'un système d'exploitation complet dans un environnement isolé. Cela nécessite une quantité importante de ressources, notamment de la mémoire et de l'espace de stockage. Le démarrage d'un système d'exploitation invité peut prendre plusieurs minutes. En revanche, les conteneurs partagent le noyau sous-jacent du système d'exploitation hôte et isolent les processus s'exécutant à l'intérieur de ceux-ci des autres processus s'exécutant sur l'hôte. Cela signifie que les conteneurs utilisent directement les fonctionnalités de l'hôte, ce qui se traduit par une utilisation plus efficace des ressources. Plusieurs conteneurs peuvent être exécutés sur un seul hôte, et le démarrage d'un nouveau conteneur est rapide, similaire au démarrage d'une application native. Les conteneurs sont donc en quelque sorte des paquets autonomes et légers qui incluent une application et toutes ses dépendances, partageant le noyau du système d'exploitation hôte. Comparés aux machines virtuelles, ils sont beaucoup plus petits et plus efficaces en termes de ressources, ce qui permet un démarrage rapide avec une utilisation réduite des ressources. Ils ont été initialement conçus pour un déploiement consistant dans

divers environnements, du développement à la mise en production. Les conteneurs offrent une solution plus efficace et légère pour l'exécution d'applications isolées que la virtualisation traditionnelle évoquée précédemment.

Actuellement, les conteneurs sont une des solutions les plus plébiscitées pour assurer la reproductibilité dans les articles de recherche utilisant une partie informatique. Les conteneurs encapsulent une application et ses dépendances dans une unité autonome, légère et portable. Ces conteneurs fournissent un environnement isolé pour exécuter des applications sur différents systèmes d'exploitation et environnements informatiques, garantissant un comportement cohérent et la reproductibilité des résultats, de manière similaire à ce qui a été obtenu avec les machines virtuelles, mais de manière plus efficace (Boettiger 2015).

La première technologie de conteneurisation étudiée est Docker (Merkel 2014). Plusieurs articles ont plaidé en faveur de Docker dans le cadre de la recherche reproductible (Boettiger 2015; Nüst *et al.* 2020). Étant donné que Docker était la première technologie étudiée, elle a été comparée à la virtualisation basée sur un hyperviseur concernant le partage de l'environnement logiciel. Dans (Felter *et al.* 2015), les auteurs ont travaillé avec Kernel Virtual Machine (KVM) (Kivity *et al.* 2007), qui est un hyperviseur de type 1 souvent étudié. Ils ont exécuté le benchmark Linpack sur Linux natif, Docker et KVM. Une exécution de Linpack passe la majeure partie de son temps à effectuer des opérations mathématiques en virgule flottante. Les performances étaient presque identiques sur Linux et Docker, mais les performances de KVM étaient nettement moins bonnes. Ils ont effectué plusieurs tests différents, pas seulement des opérations arithmétiques en virgule flottante, et ont conclu que Docker égalait ou dépassait les performances de KVM dans tous les cas. Chae *et al.* (2019) ont comparé Docker à KVM, et leurs résultats montrent que Docker est plus rapide que KVM, en constatant que Docker performe mieux que KVM en termes d'utilisation du CPU, du disque dur et de la RAM. Rad *et al.* (2017) ont également constaté que Docker performe mieux que les machines virtuelles. Une fois de plus, dans (Chung *et al.* 2016), les auteurs ont constaté que Docker génère moins de surcharge que les machines virtuelles. Potdar *et al.* (2020) ont comparé Docker et les machines virtuelles en termes de performances CPU, de débit mémoire, d'E/S disque, de test de charge et de mesure de vitesse d'opération, et ont observé que Docker performe mieux que les machines virtuelles dans tous les tests. Ruiz *et al.* (2015) confirme également que les technologies de virtualisation utilisant des conteneurs performent mieux.

À partir de cet ensemble d'études, nous constatons que les conteneurs comme Docker sont de meilleurs outils à utiliser que les machines virtuelles dans le contexte de la recherche

reproductible. Cependant, nous n'avons pas encore répondu à la question de savoir si Docker induit une surcharge de performance par rapport à l'exécution native, et s'il convient au contexte du calcul à hautes performances, c'est donc ce que nous abordons ci-après.

Bien que Docker puisse sembler être une bonne option, il ne convient pas au calcul à hautes performances. La première raison est que Docker nécessite des privilèges administrateur. Au sein des clusters de calcul, aucun utilisateur ne devrait avoir de privilèges administrateur. Pour cette raison, Docker n'est pas utilisable dans un contexte de calcul à hautes performances. Le deuxième point est que nous avons trouvé des mentions de préoccupations en matière de sécurité dans diverses études sur Docker (Jacobsen et Canon 2015; Priedhorsky et Randles 2017; Zhou *et al.* 2022).

D'autres conteneurs ont été développés pour pouvoir être utilisé dans le contexte du calcul à hautes performances : Charliecloud (Priedhorsky et Randles 2017), Shifter (Gerhardt *et al.* 2017), Singularity (Kurtzer *et al.* 2017) (renommé Apptainer), Sarus (Benedicic *et al.* 2019) et Podman (Gantikow *et al.* 2020). Charliecloud exécute des conteneurs sans nécessiter de droit administrateur ou de « daemon ». Il convertit les images Docker en fichiers tar et les décompresse sur les nœuds HPC. Il est considéré sécurisé et supporte également MPI, une norme pour la communication entre les processus dans les systèmes parallèles. De plus, Charliecloud offre une solution pour les problèmes de compatibilité des bibliothèques logicielles. Il le fait en intégrant des fichiers provenant du système hôte directement dans les images de conteneurs. Cette méthode assure une meilleure compatibilité des applications en conteneur avec l'environnement dans lequel elles sont exécutées. Shifter est un moteur de conteneurs pour le calcul haute performance (HPC) créé par NERSC. Il utilise Docker pour construire des images, les transformant en format « ext4 » puis les compressant en « squashfs » pour le stockage sur des systèmes de fichiers parallèles. Il prend en charge MPI et gère la compatibilité des pilotes GPU en remplaçant le pilote GPU au démarrage du conteneur. Apptainer est un outil populaire dans l'enseignement supérieur et l'industrie pour le HPC. Il fonctionne sans privilèges administrateur et ne requiert pas de processus « daemon », supportant les GPU, MPI, et particulièrement la technologie InfiniBand, essentielle pour l'interconnexion dans les systèmes HPC. Apptainer permet l'exécution portable d'applications via un fichier image unique au format SIF et propose des modèles hybrides pour les applications MPI. Sarus, un autre moteur de conteneurs HPC, s'appuie sur « runc » pour créer des conteneurs conformément à la spécification Open Container Initiative (OCI). Il se compose d'un répertoire de système de fichiers racine et d'un fichier de configuration JSON. Podman exécute des

conteneurs sans élévation des privilèges, en utilisant l'espace de noms utilisateur. Il prend en charge le même runtime que Sarus et Docker (runc), ainsi que le runtime plus rapide crun. Il introduit le concept de pod pour regrouper des conteneurs pour des applications complexes. Toutes ces technologies reposent sur une fonctionnalité de base de Linux appelée Linux Containers (LXC) (Senthil Kumaran 2017).

La recherche a montré qu'aucun des conteneurs (virtualisation au niveau du système d'exploitation) n'induit une surcharge significative (n'affecte pas négativement les performances) ou a au moins des performances proches de celles d'une exécution native. Younge *et al.* (2017) montre que Singularity fonctionne à des performances natives dans le cadre du calcul à hautes performances. Xavier *et al.* (2013) montre que la virtualisation LXC offre des performances natives. Une étude plus complète de 2019 (Torrez *et al.* 2019) a étudié les performances de Shifter, Charliecloud et Singularity. Ils n'ont trouvé aucune différence significative de performances entre les trois environnements et l'exécution native, à l'exception possible d'une légère variation de l'utilisation de la mémoire. Young *et al.* (2018) ont étudié les performances de Singularity et Docker, et les résultats montrent que Singularity n'entraîne aucune perte de performance, mais Docker entraîne une légère dégradation. Hale *et al.* (2017) ainsi que Le et Paz (2017) confirment également que Singularity n'entraîne aucune perte de performance. Abraham *et al.* (2020) étudie davantage le débit d'entrée/sortie sur Docker, Charliecloud, Singularity et Podman sur un système de fichiers Lustre (un système de fichiers distribué utilisé dans des environnements de calcul à hautes performances au-delà de 40 nœuds). Les résultats « montrent une surcharge de temps de démarrage pour Docker et Podman, ainsi qu'une surcharge réseau au démarrage pour Singularity et Charliecloud. Nos évaluations d'E/S montrent qu'avec une parallélisation croissante, Charliecloud génère une surcharge importante sur le Metadata Server (MDS) et le Object storage server (OSS) de Lustre. De plus, nous constatons que le débit observé des conteneurs sur Lustre est comparable à celui des conteneurs exécutés à partir du stockage local ». Casalicchio et Perciballi (2017) ont constaté une légère surcharge pour Docker, d'environ 5 % à 10 % pour la charge CPU et de 10 % à 30 % de surcharge d'E/S disque. Axés sur des scénarios de calcul à hautes performances, Hu *et al.* (2019) n'ont trouvé aucune surcharge pour Singularity sur les applications parallèles MPI et GPU.

Sur la base de ces connaissances, nous constatons que Apptainer (Singularity) et d'autres technologies de conteneurs, à l'exception de Docker, conviennent au calcul haute performance et à la recherche reproductible. Nous pensons qu'Apptainer est la meilleure solution à utiliser et à apprendre actuellement, car il est l'outil le plus répandu sur les clusters de calcul. Cependant,

comme Apptainer a récemment changé (évolution à partir de Singularity), une étude à jour pourrait être utile pour confirmer qu'Apptainer reste parfaitement adapté aux performances natives ou proches de celles-ci.

Un inconvénient des conteneurs est qu'ils sont initialement conçus pour le déploiement, et non pour l'archivage. Bien qu'il soit possible de créer un Dockerfile qui spécifie les étapes exactes pour construire une image de conteneur, il n'y a toujours aucune garantie que l'image résultante sera entièrement reproductible. Cela signifie que même si deux personnes utilisent le même Dockerfile, elles peuvent obtenir des images de conteneur légèrement différentes. Une des raisons de ce manque de reproductibilité est que les images de conteneurs sont des binaires, ce qui signifie qu'elles sont compilées et non facilement lisible ou modifiables. Cela rend difficile la compréhension de toutes les dépendances et configurations à l'intérieur d'une image de conteneur. Si quelqu'un souhaite reproduire les calculs effectués à l'intérieur d'un conteneur, il devrait recréer l'environnement de conteneur exact. Cependant, en raison de la non-reproductibilité des images de conteneurs, ce processus peut ne pas donner des résultats identiques. Par conséquent, bien que les conteneurs soient efficaces pour le déploiement d'applications, ils peuvent ne pas être la solution idéale pour obtenir une reproductibilité totale dans le contexte de la recherche scientifique. Cependant, dans la pratique, les conteneurs sont les outils les plus utilisés et restent adaptés à la recherche reproductible. Comme mentionné précédemment, Guix pourrait actuellement être la meilleure solution que nous connaissons pour la reproductibilité computationnelle. Des états de l'art complets des conteneurs pour le calcul à hautes performances ont été récemment publiés en 2022 et 2023, respectivement (Zhou *et al.* 2022) et (Keller Tesser et Borin 2023).

II.2.5. Calcul parallèle

II.2.5.1. La reproductibilité des calculs flottants

L'arithmétique en virgule flottante n'est pas associative. Par conséquent, lorsqu'on traite des opérations désordonnées, on peut perdre la reproductibilité. Une méthode pour résoudre ce problème consiste à mettre en œuvre directement des solutions algorithmiques pour gérer les calculs en virgule flottante. Demmel et Nguyen ont largement contribué à ce sujet. En 2013, Demmel et Nguyen (Demmel et Nguyen 2013a) ont abordé le défi de parvenir à une reproductibilité avec les sommes en virgule flottante, en particulier face à la planification dynamique des processeurs qui peuvent modifier l'ordre d'exécution des opérations et à la non-associativité des opérations en virgule flottante dans les environnements de calcul parallèle. Ils

proposent une technique de somme en virgule flottante qui est reproductible indépendamment de l'ordre de la somme, en utilisant l'algorithme Rump (Rump *et al.* 2010), plus efficace que l'utilisation d'une arithmétique avec haute précision. Cette solution fait un compromis entre efficacité et précision. Les performances de cette solution sont améliorées avec l'algorithme OneReduction, toujours de Demmel et Nguyen (Demmel et Nguyen 2014), en utilisant des nombres en virgule flottante indexés et en nécessitant une seule opération de réduction pour réduire le coût de communication sur les plateformes parallèles à mémoire distribuée. Cependant, ces solutions n'améliorent pas la précision. Le résultat calculé, même s'il est reproductible, reste exposé à des problèmes de précision, en particulier lorsque l'on traite un problème où une petite modification des données d'entrée peut entraîner de grandes variations dans le résultat (Chohra *et al.* 2016). Selon Stodden (Stodden *et al.* 2013), l'utilisation du type long-double ou de la somme de Kahan peut augmenter la reproductibilité sans augmenter beaucoup le temps de calcul. En ce qui concerne les algorithmes de somme compensée rapides et précis, les travaux récents de Blanchad *et al.* (2020) et de Lange (2022) sont intéressant. L'arithmétique par intervalles est également une solution envisagée (Revol et Théveny 2014).

Certains outils ont également été développés pour améliorer la reproductibilité de l'arithmétique en virgule flottante. La bibliothèque Intel MKL présente le CNR (Reproductibilité Numérique Conditionnelle) (Rosenquist 2012). Cette fonctionnalité limite l'utilisation des extensions d'ensemble d'instructions pour garantir des résultats numériques reproductible sur différentes architectures. Cependant, cette approche a tendance à considérablement réduire les performances, notamment sur les architectures plus récentes. De plus, elle exige un nombre de threads identique d'une exécution à l'autre pour maintenir des résultats cohérents. Verrou (Févotte et Lathuilière 2016) est un outil construit sur l'outil Valgrind qui utilise l'arithmétique de Monte Carlo (MCA) pour surveiller la précision des opérations en virgule flottante dans les simulations numériques sans nécessiter de modification du code source ou de recompilation. Conçu pour les applications à petite échelle et les codes industriels complexes, Verrou aide à diagnostiquer les inexactitudes découlant des calculs en virgule flottante, contribuant au processus de vérification et de validation dans des industries telles qu'EDF (Électricité De France), qui reposent sur des simulations numériques pour la sécurité et l'efficacité de leurs centrales nucléaires.

Cependant, pour gérer la recherche reproductible en calcul haute performance, des solutions de niveau supérieur à celles qui travaillent directement sur l'arithmétique en virgule

flottante sont préférées. De bons algorithmes en virgule flottante ne peuvent pas résoudre tous les problèmes induits par la parallélisation complexe des simulations stochastiques.

II.2.5.2. Enregistrer et rejouer

Lorsqu'il s'agit de traiter des workflows et des exécutions non déterministes, tels que nous en trouvons en calcul parallèle, il peut être difficile de répéter même nos propres expériences et d'obtenir des résultats identiques bit à bit, bien que cela soit nécessaire pour le débogage. Les outils d'enregistrement et de relecture peuvent être utiles pour répéter ou reproduire un calcul non déterministe, comme suggéré par Chapp *et al.* (2018). Nous avons synthétisé ce dernier travail pour voir comment de tels outils ont été initialement proposés pour permettre le débogage de calculs parallèles sur des clusters. L'exécution dynamique modifiant l'ordre des opérations, avec des bibliothèques à hautes performances comme MPI, qui peut en effet rendre les calculs non déterministes d'une exécution à l'autre. MPI est une bibliothèque largement utilisée pour gérer les communications et la concurrence sur des machines distribuées.

Tout d'abord, certains outils d'enregistrement et de relecture sont conçus pour une architecture à mémoire partagée, où tous les processeurs travaillent sur la même mémoire. L'outil ODR (Altekar et Stoica 2009) est un outil logiciel conçu pour les programmes multiprocesseurs qui a pour objectif d'obtenir une relecture déterministe du résultat en enregistrant uniquement une partie des données d'exécution et en employant une stratégie de recherche lors de la relecture pour converger vers une exécution reproduisant les sorties originales du programme. En évitant le besoin d'une reproduction bit à bit de l'ensemble de l'exécution et en contournant les problèmes de concurrences des données, ODR peut reproduire les comportements dans des applications multiprocesseurs telles qu'Apache et MySQL avec un facteur de surcoût moyen d'enregistrement de 1,6 fois. PRES (Park *et al.* 2009) utilise l'approche « enregistrement et relecture » pour le débogage d'exécution parallèle multiprocesseurs en développant des « schémas d'exécution » pendant l'enregistrement. Ils explorent de manière itérative les chemins d'exécution potentiels qui correspondent à ces schémas pour reproduire étroitement une exécution sur laquelle nous avons eu un bogue. Selon eux, la plupart des bogues sont fidèlement répliqués en moins de 10 essais de relecture. Respec, proposé par Lee *et al.* (Lee *et al.* 2010) est un système de relecture déterministe qui exécute simultanément la relecture aux côtés de l'exécution surveillée, vérifiant périodiquement les divergences entre les deux et maintenant des points de contrôle des états mutuellement acceptés. En utilisant des techniques de journalisation spéculative et de relecture externe déterministe, Respec gère le surcoût et garantit la correction. Les évaluations révèlent un surcoût de 18% pour les programmes à deux

threads et de 55% pour les applications à quatre threads lorsqu'ils sont testés sur les benchmarks PARSEC et SPLASH-2. ScalaMemTrace (Budanur *et al.* 2011) fournit une représentation compressée de la trace mémoire, identifiant les motifs comportementaux récurrents à travers la hiérarchie de mémoire. Évalué avec des charges de travail axées sur le calcul haute performance, ScalaMemTrace maintient une taille de trace presque constante jusqu'à 64 threads. La fidélité de relecture de cet outil était reconnue comme ayant des limitations, mais atteignant une précision de 91% sur le benchmark AMG. PinPlay d'Intel (Patil *et al.* 2010), propose des capacités d'enregistrement et de relecture adaptables, telles que la relecture de sous-groupes, et assure la compatibilité avec d'autres outils associés à Pin, visant la polyvalence. Lorsqu'il est évalué sur des charges de travail de calcul à hautes performances, PinPlay enregistre un surcoût de temps d'exécution très significatif, étant de 10 à 18 fois durant la phase de « replay » de plusieurs applications MPI. Light (Liu *et al.* 2015) de Liu *et al.* détermine et journalise de manière succincte les traces de données essentielles pour une relecture précise, en se concentrant spécifiquement sur la dépendance de flux des accès à la mémoire partagée. Les évaluations sur un ensemble diversifié de benchmarks indiquent que Light présente un surcoût de journalisation de 44% et un surcoût d'espace de 10% par rapport aux techniques traditionnelles, ce qui le rend efficace. PORRidge (Utterback *et al.* 2017), est conçu pour les programmes Cilk Plus, qui sont une extension des langages de programmation C et C++ permettant de gérer le parallélisme des données et des tâches. Cette approche utilise une technique d'enregistrement et de relecture qui ne dépend pas du type de processeur. Elle se distingue en se concentrant sur les enregistrements associés à chaque verrouillage, plutôt qu'à chaque fil d'exécution (thread). Cette méthode influence le planificateur de Cilk pour qu'il se conforme aux ordres d'accès préalablement enregistrés, en utilisant une représentation améliorée du graphe acyclique dirigé. Des tests réalisés sur une gamme variée de benchmarks ont montré que le surcoût lié à l'enregistrement variait, atteignant parfois 3,39 fois le niveau normal, avec une moyenne de 1,62 fois. Rerun (Hower et Hill 2008) est un système conçu pour les architectures multiprocesseurs. Il introduit un mécanisme de relecture déterministe, c'est-à-dire qu'il permet de reproduire de manière fiable et précise le déroulement d'un programme. Au lieu de se concentrer sur l'enregistrement de chaque conflit de mémoire individuel, Rerun tire parti des « épisodes atomiques ». Ces épisodes sont des séquences d'instructions exécutées par un seul thread qui ne génèrent pas de conflits avec d'autres threads. L'avantage majeur de cette approche est qu'elle nécessite une quantité de mémoire relativement faible pour chaque cœur du processeur. QuickRec (Pokam *et al.*, 2013) est une extension pour l'architecture Intel, offrant des capacités d'enregistrement et de relecture assistées par matériel pour les programmes

multithreads sur des systèmes multicœurs. En utilisant la plateforme d'émulation QuickIA (Chitlur *et al.* 2012) et un noyau Linux modifié, QuickRec bénéficie d'une génération minimale de journaux mémoire et de surcoût de performances, bien que sa pile logicielle introduise un surcoût moyen d'environ 13%. Samsara (Ren *et al.* 2015) capitalise sur le « hardware-assisted virtualization » (HAV), en utilisant une méthode de relecture déterministe dans les systèmes multiprocesseurs sans nécessiter de modifications matérielles. En utilisant un schéma d'enregistrement basé sur les blocs qui évite toutes les détections d'accès mémoire (une source principale de surcoût dans les méthodes précédentes), les auteurs soutiennent que Samsara réduit considérablement la taille du fichier journal à 1/70ème et réduit le surcoût d'enregistrement d'environ 10 fois à une moyenne de 2,3 fois par rapport aux solutions logicielles classiques. Plus récemment en 2017, Castor (Mashtizadeh *et al.* 2017) offre une solution par défaut d'enregistrement et de relecture pour les applications multicœurs, mettant l'accent sur des surcoûts minimes et prévisibles. Notamment, bien que Castor atteigne un faible surcoût d'enregistrement pour la majorité des benchmarks PARSEC, le benchmark Radiosity voit des surcoûts atteignant 25% lors d'une exécution à 10 threads en raison des impacts sur le cache. Castor peut fonctionner avec les langages C, C++ et Go. Nous pouvons également citer (Congo *et al* 2018) qui proposent un outil permettant de sauvergarder toutes les opérations de calcul via un système de log. Tous les calculs, ainsi que leur résultat, sont stockés, permettant de plus facilement débuguer ou répéter une expérience.

D'autres outils sont conçus pour une architecture à mémoire distribuée (contrairement à la mémoire partagée vu précédemment), où chaque processeur dispose de sa propre mémoire privée, et les processeurs doivent communiquer en s'envoyant des messages les uns aux autres, le plus souvent en utilisant la bibliothèque MPI. Scala-H-Trace (Wu *et al.* 2011) utilise une compression de trace agressive, capturant les variations des paramètres de communication et d'E/S via des histogrammes probabilistes, garantissant des tailles de fichier de log presque constantes même avec des contextes variables. Scala-H-Trace rejoue de manière déterministe ces traces probabilistes sans blocages, ressemblant étroitement aux applications originales, avec des temps de relecture étant de 12% à 15% supérieurs aux temps d'exécution originaux. De manière similaire à Scala-H-Trace, Scalatrace II (Wu et Mueller 2013) utilise des techniques de compression de trace, efficaces pour les applications HPC aux comportements irréguliers, en employant un schéma de codage de bas niveau qui améliore l'adaptabilité et l'interprétation des données. Guermouche *et al.* (2011) introduisent un protocole de points de contrôle non coordonnés adapté aux applications HPC avec MPI, qui ne journalise que les messages

sélectionnés et n'exige pas un redémarrage complet du processus après une défaillance. Meneses *et al.* (2010) utilisent une technique conçue pour atténuer le surcoût mémoire inhérent à la journalisation des messages en mémoire en regroupant les processeurs en groupe, où seuls les messages inter-groupes nécessitent une journalisation. Xue *et al.* (2009) ont introduit MPIWiz, basé sur une méthode qu'ils ont nommée relecture reproductible de sous-groupe, une technique hybride de relecture déterministe pour les applications MPI qui équilibre les avantages et les limitations de la relecture des données et de la relecture des instructions dans l'ordre original. En utilisant MPIWiz, le système capture le contenu des messages entre les groupes de processus et seulement les ordres des messages au sein d'un groupe, montrant une augmentation de 27% du temps d'exécution lors de l'enregistrement et permettant une relecture en seulement 53% du temps d'exécution de base de l'application. Gioachin *et al.* (2010) ont présenté un mécanisme hybride de relecture en trois étapes où les premières passes adoptent une relecture d'ordre minimaliste, passant ensuite à des relectures de données plus intensives mais limitées à un nombre progressivement réduit de processus, facilitant ainsi la traçabilité ciblée des erreurs. Rex (Perianayagam *et al.* 2010), introduit par Perianayagam *et al.*, est un ensemble d'outils conçus pour enregistrer, archiver et rejouer de manière exhaustive des expériences logicielles, garantissant la fidélité de la reproduction malgré d'éventuels changements dans les ensembles de données externes, l'indisponibilité du logiciel original ou les paramètres d'entrée non documentés. Il ajoute un surcoût minimal en termes de temps d'exécution, d'environ 1,6 %, et un surcoût en termes d'espace d'archivage variant entre 5 et 7 Go. Un état de l'art complet des outils d'enregistrement et de relecture se trouve dans (Chapp *et al.* 2018).

Comme nous pouvons le voir, de nombreux outils ont été proposés assez récemment car le problème du non-déterminisme est omniprésent en calcul à hautes performances, et il s'intensifie avec l'avancement des technologies, notamment la diminution de la taille de gravure des processeurs et l'augmentation des puissances de calcul. Beaucoup de ces outils ont un effet négatif non négligeable sur les performances, ce qui réduit leur utilisation. Cependant, ce sont les dernières solutions possibles pour la mise au point des programmes. Il faut se rappeler sans cesse que garantir la répétabilité et la reproductibilité sont des aspects essentiels pour le débogage. En effet, bon nombre de ces outils ont été créés à des fins de débogage, et non pas pour la reproductibilité. Néanmoins, en calcul à hautes performances, il est intéressant de savoir que c'est une solution existante qui permet d'effectuer des expériences reproductibles.

II.2.6. Gestion des erreurs silencieuses

Les erreurs silencieuses sont courantes dans les systèmes de calcul à hautes performances, et plus encore à l'échelle des calculs « exaflopiques ». Ce phénomène a été étudié, et plusieurs solutions existent.

Au niveau de la mémoire, la technologie de correction d'erreur (ECC memory - Error Correction Code) a été conçue pour protéger les mémoires RAM. La mémoire ECC est une forme spécialisée de stockage de données informatiques qui utilise un code de correction d'erreur pour identifier et corriger les occurrences de corruption de données sur une certaine quantité de bits dans la mémoire. Elle est utilisée dans des situations où la corruption des données est inacceptable, pour le calcul scientifique mais aussi dans les systèmes de contrôle industriel, les bases de données critiques et les caches mémoire essentiels. La mémoire ECC est conçue pour empêcher les erreurs dues à un seul bit qui pourraient affecter l'intégrité de la mémoire, garantissant que les données lues correspondent aux données initialement écrites, même si un bit a été altéré involontairement. En revanche, la plupart des mémoires non-ECC classiques ne disposent pas de capacités de détection et de correction des erreurs, seules certaines configurations de mémoires non-ECC comportent une prise en charge de la parité, ce qui permet la détection des erreurs sans correction. Dans l'étude de Baumann (2005), Baumann a travaillé sur l'observation du taux d'erreurs pour une SRAM non protégée par ECC. « Ceci est calculé en utilisant un taux d'erreurs silencieuses (SER) dû à l'exposition aux radiations, ce qui donne un estimé de 50000 FIT (Failure-In-Time : un FIT équivaut à une défaillance sur 1 milliard d'heures de fonctionnement). Ils recommandent donc l'utilisation de l'ECC, qui divise le taux d'erreur par 1000 pour les SRAM. » (Dixit *et al.* 2021). D'autres contre-mesures matérielles pour gérer les erreurs silencieuses incluent la redondance et la parité. La redondance consiste à dupliquer des composants ou des données critiques de manière à ce que, en cas d'erreur sur l'un d'entre eux, le composant ou les données redondants puissent prendre le relais de manière transparente. La parité, une forme plus simple de vérification d'erreur, ajoute un bit supplémentaire à un ensemble de données pour garantir une parité paire ou impaire (la somme des bits étant paire ou impaire). En cas d'erreur, le bit de parité change, ce qui indique un problème.

Des solutions logicielles existent également. Fiala *et al.* (2012) propose une bibliothèque associée à MPI pour gérer les erreurs silencieuses, basée sur la redondance. Dans (Benson *et al.* 2015), les auteurs proposent de gérer les erreurs silencieuses avec un nouveau paradigme pour détecter de telles erreurs au niveau de l'application. Cette méthode consiste à comparer

régulièrement les valeurs obtenues par des calculs avec celles issues d'un processus de vérification peu coûteux. En analysant les écarts entre ces deux séries de résultats, des systèmes de détection d'erreurs sont établis. Les chercheurs emploient l'analyse numérique pour déterminer des procédures de vérification adaptées aux problèmes de valeurs initiales rencontrés dans les équations différentielles ordinaires et les équations aux dérivées partielles. Plus précisément, ils utilisent des techniques telles que les méthodes de Runge-Kutta et les méthodes linéaires à plusieurs étapes pour les équations différentielles ordinaires, ainsi que des approches de différences finies, tant implicites qu'explicites, et pour les équations aux dérivées partielles. Pour illustrer leur méthode, les auteurs se réfèrent à des exemples comme l'équation de la chaleur et les équations de Navier-Stokes. Il est intéressant de noter que, lors de tests incluant des erreurs introduites volontairement, la stratégie proposée a démontré une capacité à identifier presque toutes les erreurs significatives sans causer de baisse notable de la performance de calcul. Hoemmen et Heroux (2011) proposent une version résiliente de la méthode itérative GMRES. Cette version résiliente est conçue pour corriger les erreurs et améliorer la tolérance aux erreurs de la méthode. Cependant, Bronevetsky et de Supinski (2008) ainsi que Casas *et al.* (2012) suggèrent que des études empiriques ont montré que certaines méthodes itératives pourraient être vulnérables aux erreurs. Ces études indiquent que toutes les méthodes itératives ne corrigent pas intrinsèquement les erreurs. Huang et Abraham (1984) introduisent l'utilisation de méthodes de somme de contrôle pour améliorer l'intégrité de la multiplication de matrices, en détectant efficacement les erreurs dans le processus. Du *et al.* (2012) proposent l'application de méthodes de somme de contrôle pour améliorer la robustesse de la décomposition LU (Lower-Upper) à hautes performances, détectant ainsi les erreurs potentielles dans le processus de factorisation LU. Dans (Aupy *et al.* 2013), la solution proposée pour gérer les erreurs de corruption silencieuse des données repose sur la révision des stratégies traditionnelles de sauvegarde et de récupération par retour en arrière. L'accent est spécifiquement mis sur la résolution d'erreurs latentes qui ne sont pas immédiatement détectées. Ils introduisent deux modèles pour gérer de telles erreurs. Dans le premier modèle, les erreurs sont détectées après un certain délai, suivant une distribution de probabilité souvent représentée sous forme de distribution exponentielle. La solution calcule la période optimale pour la sauvegarde, dans le but de minimiser le temps perdu pendant que les nœuds ne sont pas engagés dans des calculs utiles. Étant donné que seul un nombre limité de points de sauvegarde peut être stocké en mémoire, il existe une possibilité de défaillance irrécupérable en raison de ressources limitées. Dans de tels cas, les auteurs déterminent la période minimale nécessaire pour un niveau acceptable de risque. Dans le deuxième modèle, les erreurs sont détectées grâce à un

mécanisme de vérification. Contrairement au premier modèle, il n'y a pas de risque de défaillance irrécupérable car le mécanisme de vérification garantit la détection des erreurs. Cependant, les coûts introduits par ce mécanisme de vérification comptent dans le calcul du temps perdu. L'objectif principal est de trouver une période de sauvegarde optimale qui minimise le temps perdu, en tenant compte du compromis entre la détection précoce des erreurs, le stockage limité des points de sauvegarde et les surcoûts introduits par les mécanismes de vérification. Les auteurs appliquent ces modèles à des scénarios réels et tiennent compte de divers paramètres d'application et d'architecture pour démontrer leur faisabilité et leur efficacité.

Des technologies plus récentes peuvent offrir de nouvelles solutions, telles que l'apprentissage automatique. Dans (Wang *et al.* 2018), ils proposent un détecteur basé sur un réseau de neurones capable de détecter les corruptions silencieuses des données, même plusieurs itérations après leur injection. L'efficacité de ce détecteur proposé est évaluée à l'aide de six applications Flash et de deux mini-applications Mantevo. Les résultats des expériences montrent que ce détecteur peut identifier avec succès plus de 89 % des corruptions silencieuses des données tout en maintenant un faible taux de faux positifs de moins de 2 %. Plus d'informations sur ce sujet peuvent être trouvées dans un état de l'art sur les techniques de modélisation et d'amélioration de la fiabilité des systèmes informatiques (Mittal et Vetter 2015).

II.3. Exemples pratiques de reproduction de résultats d'articles

Pour illustrer les principes théoriques et mettre en évidence les obstacles communs, nous fournissons dans cette section quelques exemples de tentatives de reproduction d'articles scientifiques.

En 2020, Hinsen a tenté de reproduire son propre travail intitulé « Structural flexibility in proteins - impact of the crystal environment » (Hinsen 2020a). Il a rencontré des défis significatifs pour reproduire les résultats de ses recherches informatiques en raison de l'évolution des environnements logiciels et des problèmes de gestion des données. Son travail original utilisait des versions antérieures de Python et des bibliothèques telles que le Molecular Modeling Toolkit (MMTK), ScientificPython et netCDF, qui ne sont plus compatibles avec les mises à jour logicielles récentes comme Python 3 et les dernières versions de NumPy qui ont cessé de supporter les interfaces obsolètes. La transition de Python 2 à Python 3 a été particulièrement problématique en raison de changements majeurs dans l'API C de Python et

une définition différente des divisions (en Python 2, « / » est un opérateur de division entière, tandis qu'il est un opérateur de division flottante en Python 3). De plus, des scripts cruciaux pour générer des graphiques et d'autres sorties n'ont pas été publiés ou ont été perdus en raison de défaillances matérielles, notamment concernant le stockage sur des CD-ROM et sur des bandes magnétiques illisibles (pour lesquelles il n'a pas été possible de trouver un lecteur fonctionnel). Ces problèmes ont été exacerbés par des changements dans les formats de données utilisés par des bases de données essentielles comme la Protein Data Bank, qui peuvent avoir mis à jour ou corrigé leurs entrées, affectant la reproductibilité des analyses basées sur les données originales. Bien que l'auteur ait pu obtenir des résultats assez proches de l'article original, il n'a pas pu reproduire exactement tous les résultats (comme les figures par exemple) en raison de la perte de certains codes comme précité.

En 2023, Legrand et Velho ont tenté de reproduire leur propre travail sur SimGrid (Legrand et Velho 2023). Dans leurs efforts pour répliquer leur précédent papier, ils ont fait face à une série d'obstacles importants, principalement en raison de l'indisponibilité de l'article original, des codes sources et des ensembles de données, exacerbés par les défis posés par les dépendances logicielles obsolètes. L'article original et les codes sources associés n'étaient pas facilement accessibles en raison de problèmes de droits d'auteur et de l'arrêt du logiciel GTNetS, crucial pour les simulations. Pour surmonter ces obstacles, les auteurs ont téléchargé les documents et codes nécessaires sur le dépôt en libre accès HAL, GitHub ou Software Heritage, garantissant ainsi que les chercheurs futurs pourraient accéder librement à ces matériaux. Ils ont également relevé le défi de recréer un environnement logiciel compatible en développant une image Docker qui émulait les anciens systèmes et en utilisant Debian snapshot archive, ce qui permet de fournir la plateforme nécessaire pour exécuter les dépendances obsolètes. Même si Paul Velho a fait des efforts pour documenter le workflow à l'époque, les auteurs concluent que d'autres scientifiques n'auraient probablement pas réussi à reproduire le travail original, principalement en raison de sa complexité et du fait que même si une partie du code source était publiquement disponible, elle restait cachée, et donc peu de scientifiques auraient pu le trouver. Les auteurs ont minutieusement documenté leur processus de reproduction, incluant des instructions détaillées de configuration et des modifications aux scripts et aux workflows de simulation pour garantir la compatibilité avec les systèmes contemporains. Cela met en évidence l'importance d'une documentation approfondie, de la préservation des artefacts numériques et de l'accès libre aux matériaux de recherche pour soutenir la reproductibilité académique.

Un dernier exemple sur l'apprentissage automatique illustre certains problèmes de HPC. Dans (Langezaal *et al.* 2023), les auteurs ont rencontré plusieurs défis dans leur tentative de reproduire l'étude « Label-free explainability for unsupervised models ». Un obstacle important était le temps considérable requis pour l'exécution des expériences, certaines prenant plus de 32 heures sur une machine à faible coût. Ce problème a imposé des exigences substantielles sur les ressources et de la patience. De plus, ils ont rencontré des complications dues à des bugs dans la base de code, qui ont ajouté de la complexité au travail de reproductibilité. Ces bugs ont transformé ce qui serait typiquement des exécutions de ligne de commande simples en une tâche plus laborieuse, compliquant davantage leurs efforts. Lorsqu'on considère des expériences qui consomment beaucoup de temps comme dans le domaine du HPC, un obstacle majeur à la reproductibilité pourrait simplement être le manque de ressources matérielles ou d'allocation de temps de calcul suffisant.

II.4. Quelques problèmes « ouverts » concernant la recherche reproductible en calcul haute performance

II.4.1. Portabilité des algorithmes, en particulier des générateurs de nombres pseudo-aléatoires

Vérifier la justesse des algorithmes en calcul à hautes performances est une tâche difficile. Un exemple frappant est l'incohérence observée lorsque l'on implémente un générateur de nombres pseudo-aléatoires dans différents langages de programmation ou technologies. Parmi les meilleurs PRNGs, Philox mentionné précédemment est un exemple où il faut être prudent sur la portabilité, MLFG est un autre exemple où nous avons rencontré des problèmes de portabilité. Lorsqu'un PRNG est initialisé avec le même état initial, nous nous attendons à obtenir une séquence identique de nombres quels que soient les différents environnements informatiques et bibliothèques (comme Numpy ou TensorFlow). Cependant, des variations apparaissent, jetant le doute sur la portabilité et la consistance de l'algorithme. Cela remet également en question la fiabilité des résultats scientifiques obtenus à partir de tels algorithmes. Les fondements de la science informatique déterministe reposent sur la prévisibilité et la fiabilité des algorithmes. Les disparités observées dans les sorties des PRNGs soulignent un problème plus large : sans mise en œuvre normalisée et vérification rigoureuse entre les plateformes, comment pouvons-nous garantir l'intégrité de nos résultats algorithmiques ? Cela appelle à un effort concerté pour développer et respecter la normalisation dans les

implémentations de PRNG. De plus, cela nécessite la création de logiciel de vérification robustes pouvant garantir la fidélité et la portabilité algorithmiques dans divers environnements informatiques, ce qui est essentiel pour l'avancement de la science.

II.4.2. Reproductibilité avec les nouveaux paradigmes de calcul

Les modèles de calculs émergents, notamment l'ordinateur quantique, posent des défis sans précédent en matière de reproductibilité. L'ordinateur quantique, par exemple, fonctionne selon un ensemble de principes différents de l'informatique classique, ce qui entraîne de nouveaux types d'erreurs et d'incertitudes. Le domaine en est encore à ses débuts, de nombreuses options techniques étant explorées pour créer des qubits fiables pour les circuits quantiques. Nous avons souvent besoin d'un millier de qubits excellents pour modéliser un seul qubit parfait, et les expériences qui fonctionnent correctement sur des simulateurs quantiques sont souvent non reproductibles sur de véritables machines quantiques (Hill *et al.* 2023). La nature probabiliste de l'informatique quantique rend de fait la répétabilité insaisissable et la reproductibilité encore plus essentielle. Les outils et pratiques actuels ne sont pas adaptés pour faire face à ces nouveaux problèmes, ce qui nécessite un effort de recherche pour vérifier les résultats des calculs quantiques bruités et comprendre comment surmonter les obstacles à la reproductibilité dans ce paradigme de calcul disruptif.

II.5. Conclusion

Atteindre la reproductibilité en calcul à hautes performances présente des défis. Certains outils choisissent de privilégier la facilité d'utilisation au détriment de l'optimisation des performances, comme on peut l'observer avec les principes FAIR conçus pour des disciplines telles que la biologie (Wilkinson *et al*. 2016). Pour d'autres la performance est recherchée prioritairement. Ainsi, il est intéressant de faire la distinction entre la recherche reproductible en général et la reproductibilité dans le contexte du calcul à hautes performances.

À l'avant-garde de l'exploration scientifique, le calcul à hautes performances incarne l'innovation technologique, mais se débat plus particulièrement depuis une dizaine d'années avec le principe fondamental de la reproductibilité des résultats des expériences computationelles. Cet article s'est lancé dans un voyage à travers le paysage complexe de la reproductibilité dans le monde du calcul informatique et plus précisément celui du calcul à hautes performances. Nous avons examiné la crise de la reproductibilité qui sévit non seulement

dans ce domaine, mais aussi dans de nombreux autres domaines scientifiques. Nous avons souligné le rôle essentiel de l'ingénierie logicielle, de la gestion des différentes versions des codes, de la gestion des workflows et de la culture scientifique qui influence la reproductibilité, et avons proposé des solutions robustes telles que la programmation lettrée, la gestion avancée des workflows et des technologies de conteneurisation telles que Guix et Apptainer.

En ce qui concerne la recherche reproductible dans son ensemble, à notre avis, un problème concerne la culture scientifique des chercheurs informaticiens. Les agences de financement devraient mettre l'accent sur la promotion et l'incitation à la recherche reproductible, encourageant les chercheurs à utiliser les outils existants. Avec des efforts raisonnables, il pourrait devenir courant d'accompagner les publications scientifiques de documents connexes (les artéfacts). Comme l'informatique est un domaine de recherche relativement jeune, de nombreux praticiens manquent de formation en épistémologie. Cela se manifeste souvent lorsque les informaticiens utilisent à tort le terme de « méthodologie » au lieu de « méthode » par exemple (ils ne sont pas les seuls…). Cependant, avec la poursuite de la promotion de la recherche reproductible, la pratique bien établie de tenir des cahiers de laboratoire, essentiels en biologie et dans d'autres disciplines, commence à trouver son équivalent via les « notebooks » de type Jupiter qui commencent à se répandre en informatique.

Dans le cas de la reproductibilité en calcul haute performance, la situation est plus compliquée. Le calcul exaflopique est désormais une réalité. Les super calculateurs deviennent beaucoup plus denses et, même s'ils affichent les meilleures performances jamais obtenues, ils ont un temps moyen de fonctionnement sans panne très limité (en dessous de 24h), souvent en raison d'erreurs silencieuses mais pas uniquement. Nous avons constamment besoin de puissance de calcul accrue pour mener des explorations plus poussées avec des programmes complexes. Par exemple, un plan complet d'expériences avec toutes les combinaisons des facteurs possibles (paramètres d'entrée) reste bien souvent inabordable. Par conséquent, une augmentation de la capacité d'échantillonnage de l'espace de toutes les expériences possibles est toujours la bienvenue. Cette approche nécessite toujours plus de puissance de calcul. Même si l'objectif principal est d'améliorer la vitesse de calcul, cela nécessite le déploiement d'architectures parallèles étendues et d'optimisations, et nous avons découvert au cours de la dernière décennie que cela se fait trop souvent au détriment de la reproductibilité. D'autre part, les calculs couteux sont des calculs que nous ne voulons pas avoir besoin de reproduire (Hinsen 2021). Cependant, dans ce cas, nous devons pouvoir avoir confiance dans les résultats obtenus, cela peut supposer l'archivage de données conséquentes comme dans (Boyer *et al.* 2022a;

Boyer *et al.* 2022b). Atteindre la reproductibilité en calcul haute performance passe souvent par une perte de performances lorsque l'on accepte de désactiver certaines optimisations qui font perdre la répétabilité dans un contexte de calcul avec des nombres réels flottants. Nous pensons qu'il restera toujours des problèmes en matière de reproductibilité de simulations parallèles massivement optimisées. Pour des calculs parallèles plus modestes, des outils tels que Guix ou Apptainer (Singularity) se sont révélés efficaces avec des performances proches des performances natives. Pour l'instant, que vous travailliez sur une plateforme à mémoire partagée ou à mémoire distribuée, si vous optimisez fortement des calculs parallèles massifs, vous avez de grandes chances de rencontrer un manque de reproductibilité numérique.

Des défis émergents proviennent également des domaines de l'intelligence artificielle et du Big Data. Une part importante des ressources informatiques mondiales est actuellement consacrée à la formation de modèles d'IA et au stockage de vastes ensembles de données. Obtenir une reproductibilité en termes de résultats numériques, mais aussi en termes de performances, peut être difficile mais essentiel pour garantir l'explicabilité en intelligence artificielle. De plus, nous devons également répondre à l'impératif de réduire notre consommation énergétique, et l'on peut aussi travailler sur des algorithmes optimisés pour être plus économes en énergie.

Enfin, une nouvelle forme de calcul haute performance est disponible grâce à l'informatique quantique. Cette technologie est probablement une de celle qui pose le plus de problèmes ouverts et stimulants. Nous ne disposons pas actuellement d'un ordinateur quantique général avec des qubits parfaits. De nombreuses options techniques sont testées pour fournir d'excellents qubits, mais ils ne sont pas parfaits, s'il est déjà possible de travailler sérieusement, il reste difficile de concevoir des circuits fiables au moment de l'écriture de cette thèse (été 2024). Bien que les simulateurs quantiques fonctionnent correctement, l'utilisation des machines quantiques rélles se heurte encore à de nombreux défis pour obtenir une reproductibilité statistique, par essence la répétabilité numérique exacte est exclue.

Au final, nous avons souhaité mettre en avant le fait que les ordinateurs et les algorithmes servent d'outils fondamentaux, non seulement pour les informaticiens, mais aussi pour de nombreuses disciplines scientifiques et que de ce fait, toutes les disciplines sont concernées. Tout comme l'approche précise et systématique de la métrologie appliquée aux instruments des biologistes et des physiciens, les scientifiques doivent connaître et reconnaître les imperfections et les incertitudes inhérentes aux outils informatiques. Pour continuer cette prise de conscience, et développer des aptitudes pratiques, nous recommandons fortement le MOOC sur la

Recherche reproductible qui aborde la méthodologie (branche de l'épistémologie qui étudie les méthodes qui produisent de la connaissance !). Ce MOOC montre l'utilisation de plusieurs outils pour une science transparente. Enfin, il nous semble impératif que les principes de la méthode scientifique, y compris le principe de la reproductibilité, soient intégrés dans les programmes d'enseignement en informatique. Dans le chapitre qui suit, nous ferons quelques propositions face à divers cas d'études concrêts principalemet en lien avec le calcul à haute performance.

III. Constats pratiques, propositions et mises en œuvre sur des « études de cas »

III.1. Introduction

Dans ce chapitre, nous proposons une liste de recommandations utilisant les outils existants afin de favoriser une recherche reproductible. Nous allons ensuite présenter des sujets d'études spécifiques qui nous ont été suggérés ou que nous avons élaboré. Pour chaque étude, nous vérifions si l'on peut mener une recherche reproductible et au besoin nous apportons une contribution dans ce sens. Nous proposons tout d'abord le développement d'un modèle épidémiologique stochastique a base de multi-agents capable de fournir des résultats répétables d'une exécution à l'autre. Ensuite, nous étudions la reproductibilité d'une étude de machine learning dans le contexte médical impactée par l'utilisation des générateurs de nombres pseudo-aléatoires. Puis, nous proposons d'étudier l'impact du matériel dans le cas d'applications utilisant le simultaneous multi-threading (utilisation des cœurs logiques des processeurs modernes) puis dans le cas des machines quantiques IBM en se focalisant sur l'algorithme de Grover. L'ensemble de ces études pose des questions et entraîne des propositions qui seront mise en œuvre dans le chapitre IV.

III.2. Recommandations et bonnes pratiques pour améliorer la recherche reproductible

Nos recommandations pour faciliter la recherche reproductible dans le domaine du calcul haute performance incluent les directives suivantes : Choisissez d'utiliser des technologies open source et courantes. Étant donné que ces technologies ne seront pas maintenues indéfiniment, des initiatives telles que l'archive « Debian snapshot » ou Software Heritage s'avèrent très importante. Utilisez des plateformes telles que GitHub ou GitLab pour favoriser le processus de développement par le contrôle de version. Utilisez des notebooks computationnels, tels que Org-mode ou Jupyter Notebook, pour rationaliser la gestion des données et des workflows de résultats. Il est impératif de maintenir une documentation approfondie de toutes les procédures. Une fois terminé, la plateforme Zenodo devrait être utilisée pour archiver tout le code et les données. Pour permettre à d'autres scientifiques de reproduire vos calculs avec précision, nous conseillons vivement l'utilisation de conteneurs légers, tels que Apptainer ou Guix ; ces outils

facilitent la reproduction sans compromettre les performances ; notamment, Guix qui s'intègre également avec Software Heritage et qui aide à créer un environnement spécifique à un moment fixé. Si votre code est hautement spécialisé pour une architecture matérielle spécifique, cette spécificité peut entraver la reproductibilité, surtout si des optimisations agressives et une parallélisation sont employées. Malgré ces défis, de nombreux outils offrent désormais des capacités substantielles pour mener une recherche reproductible. L'obstacle principal reste dans des contextes très spécifiques de calcul haute performance, où un matériel particulier est nécessaire et le code est hautement optimisé. De plus, il existe une réticence générale à reproduire des tâches coûteuses en calcul. Un calcul informatique prenant des semaines voir des mois n'est pas prévu pour être refait, de fait, l'utilisation des outils mentionnés ci-dessus renforce la confiance dans le code et les résultats, réduisant ainsi le besoin de reproduire de tels calculs couteux en énergie et en temps. En fin de compte, il ne faut pas oublier qu'une partie importante des problèmes de reproductibilité survient initialement lorsque les auteurs échouent à partager les artefacts.

Dans les sections suivantes, nous détaillerons les sujets précis que nous avons étudiés afin d'apporter une contribution au monde de la science. Nous nous sommes majoritairement concentrés sur l'utilisation des générateurs de nombres pseudo-aléatoires, de leur qualité et de leur reproductibilité selon les technologies dans lesquels ils sont employés. Nous avons également développé un modèle épidémiologique Covid, afin de mettre en place les bonnes pratiques de la recherche reproductible et d'apporté une contribution. Nous avons également étudié l'influence du matériel sur la reproductiblité des temps de calcul et certaines bibliothèques scientifiques du HPC, en termes de performance et de reproductibilité. Enfin, nous avons réalisé une petite étude pratique sur les machines quantiques, afin de voir où nous en sommes avec cette technologie qui a le potentiel de révolutionner le monde du calcul intensif.

III.3. Proposition pour une modélisation reproductible dans le contexte d'une pandémie – L'importance d'avoir un modèle reproductible

Pour améliorer les processus décisionnels, les scientifiques utilisent fréquemment des simulations. Ces outils sont largement utilisés dans diverses industries, y compris l'automobile et l'aérospatiale, entre autres. La pandémie de COVID-19 a mis en évidence le rôle crucial des simulations dans la prise de décisions concernant les interventions pharmaceutiques, l'utilisation de masque et les stratégies de confinement. La réalisation d'expériences basées sur

la simulation nécessite l'utilisation du calcul intensif. Pour garantir l'intégrité scientifique, en particulier dans la gestion des épidémies, les simulations doivent être reproductibles.

En matière de modélisation épidémiologique, quelle que soit l'approche retenue, les prédictions sont souvent illusoires, mais l'apport des modèles reste indéniable pour l'aide à la décision. Il est important d'améliorer notre compréhension des phénomènes épidémiologiques, et dans ce cadre, la simulation est un des rares outils permettant d'appréhender les systèmes complexes. En effet, nous ne sommes pas dans un contexte de modèles physiques déterministes et maîtrisés. De plus, il existe également des modèles déterministes complètement élémentaires qui montrent de façon surprenante et flagrante les limites de nos connaissances (Zwirn 2000; Zwirn 2006) et l'importance des simulations pour explorer le fonctionnement de ces systèmes que l'on nomme complexes (Wolfram 2018). De fait, Christophe Langton propose avec sa fourmi un automate cellulaire bidimensionnel avec un jeu de règles excessivement simples dont le comportement surprenant en 3 phases ne peut s'observer qu'à travers une simulation (symétrique, chaotique, dérive à l'infini) (Gajardo *et al.* 2002). Il est important de connaître ce type de système avant de se lancer comme des apprentis sorciers dans la programmation du vivant, de loin bien plus obscure que la programmation de la fourmi de Langton qui déjà nous dépasse. Les schémas utilisés pour l'évolution dans le temps, la gestion de l'évolution simultanée d'agents, engendrent eux aussi des variations parfois surprenantes comme l'ont montré (Fatès et Chevrier 2010).

III.3.1. Modélisation épidémiologique

III.3.1.1. SIR

Avant de disposer de modèles informatiques, ce sont les modèles mathématiques qui ont été utilisés pour étudier les phénomènes épidémiologiques. Les modèles à compartiments ont été introduits au début du XXe siècle par les Ecossais Kermack et McKendrick (Kermack et McKendrick 1932). Ces travaux ont été revisités de nombreuses fois (Brauer 2005) et sont connus sous différents acronymes tels que celui utilisé pour les modèles dit SIR. Cet acronyme désigne les différents compartiments d'individus : (S) pour ceux qui sont susceptibles d'être infectés par le virus, (I) pour ceux sont infectés et (R) pour ceux qui sont guéris (Recovered) et ont acquis une immunité. Pour ce type de modèle, un algorithme simple à base de règles permet de régir l'évolution d'une épidémie. Un paramètre important est le nombre de reproduction de base nommé R0. Il donne le ratio du nombre moyen d'individus contaminé par une personne infectieuse. Une estimation de ce nombre n'est pas toujours aisée, les essais par plusieurs

équipes au début de la pandémie de Covid montraient des écarts allant de 1.5 à 6.8 (Bauch 2021). Une fois ce ratio estimé, on peut aussi faire évaluer les effectifs des différents compartiments avec des équations différentielles discrétisées. Ces modèles, s'ils ont leurs limites, restent toujours intéressants, dans de nombreux cas concrets tels que ceux présentés par (Grais *et al.* 2003). Ce type de modèle ne permet pas de prendre en compte l'environnement local des individus, leurs caractéristiques propres (super contaminateur asymptomatique, facteur de risque, …), leurs caractéristiques sociales et les réseaux de relation qui régissent la propagation du virus.

III.3.1.2. SMA

On peut aller plus loin en différenciant les individus. Ils sont en effet bien différents dans leurs nombres de contacts et leurs caractéristiques personnels. La prise en compte des réseaux de connexions entre individus est un paramètre pertinent à ne pas négliger (Newman 2003). Ces aspects étaient bien étudiés par (Bansal *et al.* 2007) qui mettent plus d'accent sur l'impact des comportements différents des individus, certains étant bien plus sociaux et d'autres plus isolés. Les modèles dits individus centrés (IBM ou Individual Based Models) sont adaptés à cette modélisation (Koopman et Lynch 1999). Lorsque l'on prend en compte les interactions sociales, on tombe dans le cadre des systèmes multi-agents (SMA). Ces systèmes peuvent prendre en compte de façon très détaillée le fonctionnement d'une épidémie à petite échelle comme dans (Lee *et al.* 2021) pour étudier l'interaction entre les humains et les lieux de cluster, dans le contexte du COVID, ou évaluer la pertinence d'une application de suivi de cas contact (Omae *et al.* 2021).

III.3.1.3. Des modèles déjà développés

Plusieurs modèles ont été proposés récemment dans le cadre de la pandémie de COVID (Castro *et al.* 2021) et un modèle individu centré est proposé en Open Source (Aylett-Bullock *et al.* 2021). Dans ce contexte, nous souhaitons prendre en compte les densités variables des populations sur de larges territoires (pays entier) et sur de longues périodes, ce qui suppose le recours à des capacités de calcul conséquentes et à une capacité d'analyse de plans d'expériences sérieux tout comme ce qui a été réalisé dans l'étude (Vyklyuk *et al.* 2021). Nous avons vu récemment que la prise en compte de réseaux sociaux (en termes de relation entre les individus) peut-être intégrée dans un modèle basé sur la percolation, c'est ce qu'une une équipe Clermontoise a réalisé (Mathiot *et al.* 2021). Ce modèle se nomme PERCOVID et représente des ménages interconnectés avec leurs réseaux de relations sociales. Ce modèle a été testé pour comprendre la dynamique d'évolution de la pandémie COVID-19 en France entre décembre

2019 et décembre 2021. Pour la France, la référence en termes de reproductibilité est donnée par l'Institut Pasteur. L'article correspondant a été publié dans Science (Salje *et al.* 2020).

III.3.2. Les problèmes de reproductibilité des simulations Covid et constats de non-reproductibilité

La crise du COVID-19 a considérablement accru la sensibilisation du public à l'importance critique de la recherche reproductible, y compris chez les non-scientifiques. Des incidents notables pendant la pandémie, tels que le retrait de deux études influentes du Lancet et du New England Journal of Medicine (Piller et Servick 2020). Ces deux études, qui avaient influencé les politiques internationales concernant l'utilisation de certains médicaments, ont dû être rapidement retirées. De plus, le modèle COVID de Neil Ferguson (Ferguson *et al.* 2020), critiqué avec humour par Pouzat (Pouzat 2022) pour sa non-divulgation initiale du code, a eu une influence considérable sur les politiques de confinement, notamment au Royaume-Uni. Sous une pression significative, Ferguson a finalement publié une version révisée du code, qui a ensuite été critiquée pour de graves défauts, provoquant une pétition sur GitHub pour empêcher son utilisation comme base pour d'autres modèles épidémiologiques.

Notre recherche se concentre sur la reproductibilité des modèles COVID. Nous visons à tester l'exécution du modèle en mode console et la reproduction des résultats annoncés. Parmi divers modèles, des systèmes multi-agents comme Comokit (Gaudou *et al.* 2020) se distinguent par leur complexité et leurs fonctionnalités complètes. Cependant, nous avons notés des défis lors de l'utilisation de ces modèles pour des conceptions expérimentales parallèles, ainsi que d'autres problèmes mineurs signalés aux auteurs originaux. Nous avons sélectionné vingt modèles épidémiologiques (Covid ou autres) dans notre étude. Certains sont basés sur des modèles multi-agents, d'autres sur des modèles mathématiques. D'autre part, nous constatons que très peu de ces modèles fournissent le code source. Il est très préjudiciable à l'avancement de la science de ne pas fournir le code source qui permettrait de reproduire les résultats des articles, les calculs étant impossibles à reproduire sans le code.

De plus, même lorsque le code est fourni, des problèmes tels que des liens morts, la compatibilité des bibliothèques et une documentation incomplète peuvent entraver la reproductibilité. L'exécution du code présente souvent des défis supplémentaires, y compris des erreurs ou des exceptions, malgré le respect des instructions fournies. Les efforts en faveur de la reproductibilité ne sont pas toujours couronnés de succès, et vérifier la documentation et le fonctionnement du code sans connaissances préalables s'avère être une tâche exigeante.

	Code disponible	Langage	Mode console pour le calcul intensif	Succès au lancement du programme
Percovid (Mathiot *et al.* 2021)	-	?	?	-
Comokit (Gaudou *et al.* 2020)	Oui	Gama	Oui	Oui
SimPest (Laperrière 2006)	-	NetLogo	?	-
MicMac (Gampe *et al.* 2009)	-	R + Java	?	-
EpiSim (Mniszewski *et al.* 2008)	-	C++	Oui	-
MATSim-Covid (Axhausen 2021)	Oui	Java	Oui	-
OpenABM-Covid19 (Hinch *et al.* 2021)	Oui	C	Oui	Oui
Unamed (Hufnagel *et al.* 2004)	-	?	?	-
SABCoM (Davids *et al.* 2020)	Oui	Python	Oui	-
Unamed (Tadić & Melnik 2020)	-	?	?	-
Unamed (Aleta *et al.* 2020)	-	?	?	-
Unamed (Wang *et al.* 2021)	Oui	Python	?	-
JUNE (Aylett-Bullock *et al.* 2021)	Oui	Python	Oui	-
FluTe (Chao *et al.* 2010)	-	C/C++	?	-
COCOA SMA (Omae *et al.* 2021)	-	?	?	-
MOMA (Maneerat et Daudé 2016)	Oui	Gama	?	-
REINA (Tuomisto *et al.* 2020)	Oui	Python	Oui	Oui
INFEKTA (Gomez *et al.* 2020)	Oui	Python	Oui	-
Unamed (Chang *et al.* 2020)	-	?	?	-
Unamed (Klôh *et al.* 2020)	-	?	?	-

Table 2: Résumé des tests de reproductibilité sur différent modèles épidémiologiques

Une approche intéressante pourrait consister à faire en sorte que des individus non affiliés au projet tentent d'utiliser la documentation pour évaluer sa clarté et son applicabilité, avant sa publication. Notre examen est cependant limité par la contrainte pratique de ne consacrer qu'une heure à chaque modèle en mode console. Des modèles complexes comme Comokit (Gaudou *et al.* 2020) et MatSim-Covid (Axhausen 2021) nécessiteraient vraisemblablement plus de temps pour une évaluation approfondie, ce que nous reconnaissons pourrait considérablement améliorer leur utilité. Les résultats de notre étude sont résumés dans la table 2 ci-dessus, soulignant le besoin d'améliorations dans la reproductibilité scientifique à tous les niveaux.

L'influence des intérêts financiers, particulièrement dans le domaine médical à l'échelle mondiale (Abbasi 2020), peut gravement compromettre la qualité scientifique. L'absence de reproductibilité rapproche les efforts scientifiques de la pseudoscience. Le financement et les conflits d'intérêts gangrènent ces situations, malgré le besoin d'une collaboration publique internationale rapide et mondiale. Iqbal *et al.* (Iqbal *et al.* 2016) ont déclaré que « les articles publiés dans des revues de la catégorie médecine clinique par rapport à d'autres domaines étaient presque deux fois plus susceptibles de ne pas inclure d'informations sur le financement et d'avoir un financement privé ». De plus, (Collberg *et al.* 2015) ont observé que la recherche financée par l'industrie ou avec des auteurs issus de l'industrie est moins susceptible de partager des codes et des données, soulevant des questions sur la fiabilité de la recherche soutenue par l'industrie si les artefacts sont inaccessibles.

Pour répondre aux besoins évoqués précédemment, nous proposons de développer un modèle multi-agent, implémentant une utilisation correcte et maîtrisée de la parallélisation des générateurs de nombres pseudo-aléatoires. Nous utiliserons des outils de génie logiciel pour la recherche reproductible. L'application sera présentée dans le chapitre suivant.

III.4. Pour des expériences répétables en machine learning avec une bonne utilisation de l'aléatoire dans les framework - Reproductibilité des PRNGs dans les technologies du machine learning

III.4.1. Le contexte des cadriciels en Machine Learning

Les chercheurs en machine learning (ML) contemporains utilisent principalement des langages de programmation et des frameworks de haut niveau pour mener leurs études. Python est le principal langage de programmation utilisé en ML, ce qui a conduit à l'adoption

généralisée de frameworks tels que PyTorch et TensorFlow, souvent associés à NumPy. Nous souhaitons étudier la qualité statistique, la reproductibilité, la consommation d'énergie et de temps de la génération de nombres pseudo-aléatoires dans ces technologies. La littérature sur la qualité des générateurs de nombres pseudo-aléatoires (PRNG) dans les technologies du machine learning reste rare ; nous proposons d'aborder cette lacune.

En Python, l'algorithme de génération de nombres pseudo-aléatoires (PRNG) par défaut utilisé est Mersenne Twister (MT) (Matsumoto et Nishimura 1998). Dans TensorFlow, le PRNG par défaut est Philox (Threefry, de la même famille de générateurs crypto-secure est également disponible) (Salmon *et al.* 2011), de même que dans PyTorch. NumPy offre une variété de PRNG, offrant ainsi plus de flexibilité. Le PRNG par défaut proposé par NumPy est PCG (O'Neill 2014). Nous proposons de vérifier et de comparer la reproductibilité, la performance, la qualité statistique et la consommation d'énergie des PRNGs suivants : MT, Philox, PCG et Mrg32k3a (L'Ecuyer 1999) comme référence. Nous utilisons les implémentations originales en C fournies par les auteurs des PRNG.

L'essor de l'apprentissage profond et des modèles complexes en ML nécessite des ressources informatiques efficaces pour traiter de vastes quantités de données. Les fabricants d'accélérateurs matériels se précipitent pour proposer de meilleures performances à un rythme impressionnant. Les performances, souvent quantifiées par le temps de calcul ou la vitesse des opérations, impactent directement la faisabilité de l'entrainement de modèles plus grands. Bien qu'un algorithme optimisé ou un matériel efficace puisse améliorer l'efficacité temporelle, l'énergie consommée pendant les calculs devient également une préoccupation importante, surtout avec l'accent actuel sur la problématique environnementale (Goralski et Tan 2020). Une consommation énergétique élevée entraîne non seulement des coûts opérationnels plus élevés, mais contribue également à augmenter les empreintes carbones dans les centres de calcul. Par conséquent, comprendre et optimiser la performance et l'efficacité énergétique des calculs, y compris ceux des PRNGs, est impératif. Des PRNGs efficaces peuvent être utilisé pour des méthodes stochastiques dans les workflows de machine learning, réduisant ainsi à la fois le temps et la consommation d'énergie.

Un autre aspect de l'avancement scientifique doit être abordé : la reproductibilité en tant que pierre angulaire de l'intégrité scientifique (Hill 2015). Elle permet aux chercheurs de valider, d'étendre ou de contester les résultats antérieurs. Dans le domaine du machine learning, la reproductibilité garantit que les résultats obtenus lors d'une exécution peuvent être reproduits de manière cohérente lors des exécutions ultérieures, avec les mêmes configurations. Ce résultat

constant est crucial pour le débogage, la comparaison des modèles, la validation et l'assurance de la fiabilité de la technologie dans les applications du monde réel. Les PRNGs jouent un rôle essentiel dans ce contexte. Étant donné que de nombreux processus en ML, de la sépération des données à l'initialisation des poids, reposent sur des séquences pseudo-aléatoires, la reproductibilité des sorties des PRNGs est vitale. Sans sorties PRNGs répétables et cohérentes, des différences subtiles peuvent s'amplifier au cours du processus d'entrainement, conduisant à des résultats sensiblement différents. Au-delà des expériences individuelles, la reproductibilité est également vitale pour la communauté scientifique au sens large (Nature Editorial 2024). Lorsque les résultats peuvent être reproduits de manière fiable, cela renforce la base sur laquelle se construit la recherche future, assurant une trajectoire scientifique progressive et fiable.

Nous proposons d'étudier les questions suivantes, auxquelles nous essayerons de répondre dans le chapitre suivant :

- Les PRNGs implémentés dans les frameworks du machine learning donnent-ils les mêmes résultats que leurs codes C proposés par les implémentations originales lorsqu'ils sont initialisés de manière identique ?
- Les nombres pseudo-aléatoires générés avec les bibliothèques et les frameworks du ML ont-ils la même qualité statistique que ceux produits par le code original fourni par les auteurs des PRNGs ?
- Le processus de génération de nombres aléatoires dans les frameworks du ML est-il plus chronophage par rapport aux codes C originaux ?
- La génération de nombres aléatoires dans les frameworks du ML nécessite-t-elle plus d'énergie que ses homologues en code C ?
- En prenant en compte les points précédents, y a-t-il une équivalence sur les nombres entiers 32 bits et les nombres à double précision 64 bits générés ?

III.4.2. L'importance des PRNGs dans les technologies du machine learning

Pour souligner l'importance de la qualité statistique des PRNGs dans l'entraînement des réseaux neuronaux, un travail récent de Huk (Huk *et al.* 2021) a tenté de quantifier les différences potentielles de performance en classification des CNN et des MLP en faisant varier les PRNGs utilisé lors de l'entrainement. Ils ont mesuré l'intervalle de confiance à 95 % pour chaque mesure de qualité, pour différents PRNGs. Les résultats ont indiqué des variations mineures de qualité associées à différents PRNGs, comme en témoignent les intervalles de

confiance non superposés. Cette étude montre que le PRNG utilisé pourrait avoir une incidence (nécessitant de doubler les intervalles de confiance des métriques d'évaluation) sur la qualité de l'entraînement du réseau neuronal. (Koivu *et al.* 2022) montrent également une corrélation entre la qualité statistique d'un PRNG et la qualité résultante de la méthode de dropout appliquée au réseau neuronal. Des recherches supplémentaires sont nécessaires pour explorer diverses architectures de réseaux neuronaux, évaluer l'impact de la qualité des PRNGs sur la performance des réseaux neuronaux, et reproduire ces résultats, étant donné la rareté de la littérature sur ce sujet. La qualité des PRNGs utilisés en machine learning est peu étudiée, et il serait intéressant de l'investiguer. En effet, les processus stochastiques sont devenus de plus en plus importants en ML au fil des ans en raison de leur efficacité dans certaines méthodes. En conséquence, les PRNGs sont devenus indispensables dans les technologies du ML.

Pour illustrer l'importance des PRNGs en apprentissage automatique, nous considérons plusieurs méthodes stochastiques telles que le Stochastic Gradient Descent (SGD). C'est un algorithme d'optimisation fondamental pour l'entraînement des modèles en machine learning. Il fonctionne en utilisant un seul ou un petit lot d'échantillons d'entraînement pour calculer le gradient et mettre à jour les paramètres, plutôt que d'utiliser l'ensemble complet des données d'entraînement. Bottou a démontré la polyvalence du SGD à travers diverses méthodes de ML, y compris les perceptrons et les réseaux multicouches (Bottou 2003). Bottou (Bottou 2010) a mis en évidence le rôle du SGD dans la gestion des contraintes des volumes de données massifs et de la puissance de calcul limitée, tandis que (Nguyen *et al.* 2017) ont introduit « SARAH » (StochAstic Recursive grAdient algoritHm) pour affiner le SGD initial.

Au-delà de l'algorithme SGD couramment utilisé, connu pour son efficacité, il convient de noter le rôle significatif des techniques de dropout qui ont démontré une utilité considérable et nécessitent également une dimension stochastique. Le dropout est une stratégie de régularisation adaptée aux réseaux neuronaux pour atténuer le sur-apprentissage. Le sur-apprentissage se produit lorsqu'un modèle se conforme excessivement aux données d'entraînement, compromettant sa capacité à généraliser, ce qui entraîne une performance médiocre sur de nouvelles données. Le dropout y remédie en omettant aléatoirement une sélection de neurones et leurs connexions tout au long du processus d'entraînement (Srivastava *et al.* 2014).

De plus, le concept de profondeur stochastique, une autre technique de régularisation reposant sur le hasard, a été conçu pour surmonter les obstacles inhérents à l'entraînement des réseaux convolutionnels profonds, tels que les « vanishing gradients » et les durées

d'entraînement longues. Il simplifie le processus d'entraînement en omettant aléatoirement un ensemble de couches dans chaque lot d'entraînement et en connectant les couches restantes en utilisant la fonction identité, réduisant ainsi le temps d'entraînement et augmentant potentiellement la précision des tests (Huang *et al.* 2016).

Le hasard est également utile dans l'augmentation de données, une méthode visant à élargir l'ensemble de données en incorporant des répliques modifiées des données existantes ou en générant de nouvelles données. Cette approche est particulièrement bénéfique en machine learning, améliorant la performance du modèle grâce à un ensemble de données plus grand. Pour les tâches liées aux images, l'augmentation de données peut impliquer des altérations comme la rotation, le recadrage ou le retournement. Les algorithmes notables qui utilisent l'augmentation de données incluent l'algorithme Expectation-Maximization, et les méthodes de Monte Carlo par chaînes de Markov pour l'échantillonnage a posteriori (Van Dyk et Meng 2001). En apprentissage profond pour les images, les techniques d'augmentation qui incorporent le hasard couvrent un large spectre, des ajustements géométriques et des altérations de l'espace colorimétrique avec matrice de convolution, au mélange d'images, ou à l'effacement aléatoire. De plus, l'augmentation au moment du test introduit de la variabilité lors de l'évaluation du modèle, ce qui est crucial pour enrichir les ensembles de données et renforcer la résilience du modèle (Shorten et Khoshgoftaar 2019).

En outre, le concept de bootstrap complète ces techniques en fournissant une autre couche de hasard et de robustesse. Le bootstrap, impliquant la création de multiples sous-ensembles de l'ensemble de données par échantillonnage avec remplacement, permet la génération de conditions d'entraînement diversifiées. Cette technique est essentielle pour améliorer la précision et la stabilité du modèle, en particulier dans les méthodes d'apprentissage par ensemble où elle contribue à une exploration plus complète de l'espace de données et à une meilleure généralisation du modèle (Tsamardinos *et al.* 2018).

Une enquête récente met en évidence l'application omniprésente du hasard en machine learning comme compromis pour l'efficacité matérielle et la performance computationnelle (Liu *et al.* 2020). L'utilisation des PRNGs en ML est largement répandue. Les exemples incluent les réseaux neuronaux bayésiens (Neal 2012), les autoencodeurs variationnels présentés dans (Kingma et Welling 2013) et l'apprentissage par renforcement (Sutton et Barto 2018). De plus, certaines méthodes proposent l'injection de bruit de gradient comme stratégie pour améliorer l'entraînement des réseaux neuronaux profonds (Neelakantan *et al.* 2015).

Des travaux récents se concentrent davantage sur l'utilisation de la génération pseudo-aléatoire et la consommation d'énergie des réseaux neuronaux. Dans (Kim *et al.* 2016), ils ont utilisé le calcul stochastique sur des réseaux neuronaux profonds et ont obtenu de meilleurs résultats en termes de latence et de consommation d'énergie. Dans ce cas, l'ancienne approche du calcul stochastique (SC), introduite à l'origine par John Von Neumann au début des années soixante, où l'information est représentée et traitée à l'aide de flux de bits aléatoires, sert pour des calculs complexes opérés avec des opérations au niveau des bits. Dans (Liu *et al.* 2018), les auteurs soulignent que le SC peut être coûteux en termes d'efficacité énergétique lorsqu'il est utilisé dans des réseaux neuronaux profonds.

En outre, le paysage en rapide évolution du machine learning a vu l'émergence des architectures Transformer (Vaswani *et al.* 2017), en particulier dans le domaine des modèles de langage (LLM, Large Language Model). Ces architectures, illustrées par des modèles comme GPT (Generative Pre-trained Transformer), s'appuient toujours sur le hasard dans leur phase d'entraînement. Ce hasard se manifeste sous la forme de la descente de gradient stochastique et des techniques de dropout, essentielles pour prévenir le surentraînement et favoriser la généralisation du modèle.

Avec cette brève revue de la littérature, nous pouvons confirmer que le hasard, par le biais des PRNGs, sont des éléments importants des technologies d'intelligence artificielle qui deviendront omniprésentes dans nos vies. Étant donné que la qualité des nombres pseudo-aléatoires dans les frameworks de machine learning reste peu explorée, et nous voulons contribuer à combler cette lacune de connaissance.

III.4.3. Etude de la reproductibilté d'un article utlisant le « Deep Embedded Clustering » avec Tensorflow

Nous proposons également d'explorer l'algorithme de clustering profond intégré (Deep Embedded Clustering - DEC). Après une première observation de non-reproductibilité, nous avons tenté de reconstruire l'algorithme et avons rencontré des problèmes de répétabilité, n'obtenant pas des résultats identiques d'une exécution à l'autre avec le même matériel et exactement le même environnement.

Nous étudions l'algorithme de clustering profond intégré (DEC) décrit dans l'article de (Xie *et al.* 2016). Le DEC est une méthode d'apprentissage non supervisée qui combine un réseau neuronal avec un algorithme de clustering. Cette méthode a pour but d'apprendre une représentation latente des données d'entrée, c'est-à-dire une version condensée et informative

des données. Une fois cette représentation latente obtenue, un algorithme de clustering est appliqué sur cet espace latent pour regrouper les données similaires ensemble. Cela implique de traiter des données médicales provenant d'un groupe de recherche médicale, dans le but de regrouper les patients souffrant de douleurs chroniques en groupes cohérents, selon leurs divers symptômes, afin de permettre un traitement personnalisé pour chaque groupe. Bien que les résultats obtenus initialement paraissent plausibles, ils manquent à la fois de reproductibilité et de répétabilité.

Nous proposons d'étudier plusieurs pistes qui pourraient expliquer de tels problèmes, nous supposons obtenir une répétabilité d'exécution à exécution sur une machine identique, grâce à une compréhension particulière des paramètres stochastiques et à une utilisation appropriée des générateurs de nombres pseudo-aléatoires cachés.

III.5. Une méthode pour une parallélisation des PRNGs efficace

III.5.1. Le contexte pour une répétabilité des résultats

Les simulations stochastiques parallèles sont largement utilisées dans de nombreux domaines : physique de hautes énergies, transports, climat, finance, modélisation épidémiologique, etc. Comme pour les applications séquentielles, des flux de nombres aléatoire statistiquement corrects sont indispensables pour la qualité des résultats (Hellekalek 1998; Maigne *et al.* 2004; Click *et al.* 2011). Une connaissance approfondie de la parallélisation des nombres aléatoires n'est pas très répandue, ce qui conduit parfois à une mauvaise utilisation des flux de nombres pseudo-aléatoires parallèles, impactant non seulement la qualité des résultats mais aussi la reproductibilité des recherches (Hill 2019). Les générateurs de nombres pseudo-aléatoires sont déterministes et répétables à partir d'un état initial (ou d'une graine pour les anciens générateurs); ils produisent le même flux aléatoire pour le même état initial, ce qui est essentiel pour le débogage. En calcul à haute performance (HPC), nous parallélisons souvent les simulations stochastiques et la plupart d'entre elles sont massivement parallèles; elles occupent jusqu'à 80 % de la grille de calcul mondiale du LHC (WLCG) (Boyer 2022). Pour les simulations de Monte Carlo massivement parallèles, nous avons besoin de flux de nombres aléatoires indépendants pour chaque cœur ou fil exécutant des calculs indépendants (sans communication).

Les simulations stochastiques parallèles sont utilisées dans de nombreux domaines scientifiques différents, où tous les scientifiques n'ont pas une connaissance approfondie de

l'informatique. Lorsqu'elles sont bien conçues, et si nous nous rappelons que les PRNGs sont déterministes, il est possible d'obtenir des simulations stochastiques parallèles répétables. Pour gagner en confiance dans l'exécution parallèle, certains domaines de recherche souhaitent pouvoir comparer les résultats obtenus (à petite échelle) par une exécution séquentielle et son équivalent parallèle. Une fois les résultats satisfaisants, nous pouvons exécuter à plus grande échelle. Cette approche doit être soigneusement conçue comme spécifié dans l'article de Hill (2015). La version séquentielle doit être préparée avec une version parallèle en tête, avec autant de flux aléatoires indépendants que nécessaire pour la version parallèle correspondante. Par exemple, au lieu d'utiliser un seul flux parallèle avec une boucle pour les réplications, chaque réplication utilisera un flux différent même dans le programme séquentiel. Le but est de pouvoir tracer chaque partie stochastique indépendante dans l'exécution séquentielle et parallèle. Archiver tous les états utilisés permet des exécutions parallèles répétables. À petite échelle, les mêmes états étant utilisés dans la version séquentielle, nous espérons obtenir les mêmes résultats qu'avec l'exécution parallèle. Différentes techniques sont disponibles pour distribuer des flux aléatoires indépendants, elles sont présentées par (Hill *et al.* 2013). Principalement, il existe deux approches différentes : la partition d'un flux de générateur unique et la génération de nombreux générateurs paramétrés supposés indépendants. Avec une approche de paramétrisation, il est possible de créer différents plus petits générateurs, comme par exemple avec Mersenne Twister (MT) (Matsumoto et Nishimura 2000). Cette approche a été testée par (Reuillon 2008) sur le MT original et par (Passerat-Palmbach *et al.* 2011) pour la version GPU (MTGP). Cela valait la peine d'être testé car cela a montré des limites qui ont été reproduites et rapidement corrigées par Makoto et son équipe très réactive, toujours sensible à la qualité d'un travail scientifique. Les approches qui partitionnent un générateur unique sont également utilisées pour le calcul parallèle. Au vu de l'importance de la qualité et de la parallélisation des PRNGs, nous proposons d'étudier spécifiquement la qualité statistique des nombres aléatoire généré selon la méthode de parallélisation utilisée pour le générateur Mersenne Twister. Voici une liste des techniques courantes, qui seront utilisées dans notre étude :

- Sequence Splitting : Cette approche suppose que vous pouvez facilement connaitre le nombre maximal de nombres aléatoires dont vous allez avoir besoin pour chacune de vos simulations parallèles. Vous pouvez alors décider d'une limite supérieure avec une marge confortable afin de donner des sous-flux à chaque élément de traitement (chaque simulation). Diviser un grand flux en sous-flux espacés en part égale suppose que vous êtes capable d'archiver l'état de votre PRNG

pour démarrer chaque sous-flux à la même position. Lorsque la taille des sous-flux est correctement sélectionnée, ils ne se chevauchent pas. Par exemple, si vos éléments de simulation ont besoin au maximum de 10^{10} nombres aléatoires, vous devez espacer chaque flux d'au moins $10^{10} + 1$, cependant une bonne habitude est d'être prudent en ajoutant au moins un ordre de grandeur pour l'espacement.

- Espacement aléatoire/ Random spacing : Cette approche génère des états aléatoires adaptés au générateur. Le risque pourrait être que le PRNG ne soit pas bien initialisé. Nous savons d'après des travaux antérieurs que l'initialisation peut impacter la qualité des flux aléatoires. (Matsumoto *et al.* 2007) ont démontré que la majorité des générateurs de nombres aléatoires modernes ont des corrélations dans leurs flux en fonction de l'état d'initialisation. Parmi les 58 générateurs de la GNU Scientific Library, 40 d'entre eux ont montré des problèmes avec des schémas d'initialisation négligents. Nous devons également être prudents quant au problème des flux qui se chevauchent. Dans l'article de (Hill *et al.* 2013), les auteurs trouvent qu'avec une période comme 2^{61}, la probabilité de chevauchement commence à être notable (0,43 %), alors qu'elle est proche de 0 avec des périodes énormes comme celle proposée par la famille MT.
- Séquence indexée propre à MT : Lors de l'initialisation avec une simple graine dans Mersenne Twister (avec init_genrand()), le code applique une randomisation à partir de la valeur de type long de la graine, et remplit le tableau mt[] avec ces valeurs. Au final, c'est comme si MT avait déjà implémenté une variante de l'espacement aléatoire et qu'il génère un état MT complet de 2,5 kB à partir d'une simple graine.

Beaucoup plus de détails peuvent être trouvés dans l'article de (Hill *et al.* 2013). Comme la période de MT est énorme, des flux « indépendants » sont facilement obtenus. Nous utiliserons les 3 techniques de parallélisation présentées ci-dessus car elles conviennent aux simulations de haute performance répétables avec MT.

Les propositions que nous allons faire permettent de répondre aux questions suivantes : obtenons-nous des sous-flux de qualité et la technique de parallélisation impacte-t-elle cette qualité ? Actuellement, nous n'avons pas trouvé d'articles de recherche fournissant cette information, ni de base de données d'états MT testés, nous nous proposons donc répondre aux questions précédentes, de manière reproductible.

III.6. Proposition d'étude de l'impact du simultaneous multi-threading (SMT) sur la reproductibilité en temps de calcul et en consommation, selon le type d'application

III.6.1. Le SMT et ses problématiques

Les mesures de temps d'exécution d'un programme peuvent être des données importantes lorsqu'elles sont fournies dans un article de recherche. En général, il faut associer avec cette mesure de temps un environnement logiciel et matériel, qui définissent le contexte dans lequel le processus a pris un certain temps pour s'exécuter. On fournira donc la marque du processeur, ainsi que le nombre de cœurs sur lesquels notre charge de travail s'est exécutée. Néanmoins, l'exécution sur un cœur logique ou sur un cœur physique peut influencer la mesure de performance, et ce n'est pas le seul critère. Nous proposons donc d'étudier cela, dans le cadre de la reproductibilité des temps de calcul dans un 1er temps, puis dans le cadre de la consommation énergétique de ces codes dans un 2nd temps.

Nous étudions de manière reproductible l'impact du multithreading et de l'affinité sur les performances des simulations stochastiques parallèles. Les articles sur le multithreading sont désormais assez anciens et notre objectif est d'actualiser nos connaissances à l'ère des processeurs multicœurs de plus en plus performant, et proposant un plus grand nombre de cœurs s'exécutant en parallèle sur une même puce.

Plus de 80 % des cycles CPU du World Wide Computing Grid exécutent des simulations stochastiques avec multithreading, que l'on appelle « Simultaneous multithreading » (SMT) (Boyer 2022 thèse). Les processeurs multicœurs sont devenus une technologie omniprésente. Au départ, en 2002, Intel a publié un article (Marr *et al.* 2002) qui a introduit le SMT, également connu sous le nom d'hyper-threading (HT) pour l'implémentation brevetée d'Intel, ou SMT pour les autres fabricants. Au cours des deux décennies qui ont suivi l'introduction de ce brevet, les puces multicœurs (ou plus récemment appelées manycore) ont émergé pour réduire la taille de la puce et la consommation énergétique du parallélisme massif impliquant l'utilisation de nombreuses puces. Sur un seul CPU, on peut maintenant trouver 128 cœurs physiques (avec les puces ARM et AMD récentes) et si l'on considère le SMT, on peut multiplier ce nombre par 2 (ou parfois plus selon l'architecture SMT du fabricant) pour obtenir le nombre de cœurs logiques. Avec ce niveau de densité, on pourrait trouver un seul nœud HPC avec deux sockets offrant jusqu'à 512 cœurs logiques à la fin 2022. La distinction entre processeurs multicœurs et

processeurs manycore est quelque peu arbitraire et il n'y a pas de définition stricte qui s'applique à tous les cas. Nous avons fait le choix ici de parler de multicœurs, mais on peut trouver dans la littérature des CPU à 32 cœurs considérés comme des processeurs manycore.

Tout d'abord, expliquons comment fonctionne le SMT à 2 voies (Marr *et al.* 2002). Pour le système d'exploitation, un seul processeur physique apparaîtra comme deux processeurs logiques (le système d'exploitation ne fera pas la différence). Bien que les ressources d'exécution physique soient partagées entre cœurs logiques, l'état de l'architecture complète est dupliqué pour les deux coeurs logiques. Le premier niveau de parallélisme est le parallélisme au niveau des instructions (ILP) ; il se réfère aux techniques pour augmenter le nombre d'instructions exécutées à chaque cycle d'horloge. On peut citer les processeurs superscalaires qui exécutent plusieurs instructions à chaque cycle d'horloge. Cependant, cela implique d'utiliser la technique out-of-order pour être efficace. La technique out-of-order (exécution dynamique) est désormais implémentée dans la plupart des CPU modernes. Une combinaison d'exécution spéculative, de prédiction multiple des branches et d'analyse de flux de données peut conduire à une meilleure utilisation des ressources du cœur (moins de temps d'inactivité). Dans une machine out-of-order, une gestion correcte des exceptions et des prédictions de branchement manquées implique un mécanisme de réparation pour restaurer la machine à un état fiable précédent (Hwu et Patt 1987). La fonctionnalité out-of-order peut également entraîner une perte de répétabilité lorsqu'on traite des opérations en virgule flottante, car elles ne sont pas associatives (Goldberg 1991 ; Lutz et Hinds 2017). En effet, avec les cœurs out-of-order, les instructions ne sont pas toujours exécutées dans l'ordre spécifié par le programme source ou même par le compilateur. L'IBM System 360/Model 91 a été la première machine à proposer cette fonctionnalité en 1967, mais elle ne proposait pas d'instructions en virgule flottante. L'introduction de cette approche aux unités de virgule flottante est assez récente. Un autre niveau de parallélisme est le parallélisme au niveau des threads (TLP). En fait, dans les systèmes actuels, de nombreux processus s'exécutent en même temps (au premier plan ou en arrière-plan). Ainsi, le multithreading est également utilisé sur des machines de bureau, pas seulement pour le calcul haute performance (HPC).

Les puces multicœurs sont un moyen de faciliter la gestion du TLP en fournissant plus de « matériel ». Pour gérer plusieurs threads, il est rare de pouvoir proposer un cœur physique pour chaque tâche ; il faut passer de l'une à l'autre. Différentes méthodes existent : passer d'un thread à l'autre à un moment fixe (ou non), lorsqu'un thread est toujours exécutable, et passer d'un thread à l'autre lorsque des événements de longue latence se produisent, comme les

entrées/sorties ou la synchronisation, ou la gestion des interruptions. La politique de commutation du système d'exploitation peut varier ; cette tâche est effectuée par l'ordonnanceur. Lorsqu'un cœur CPU est programmé pour passer à un autre thread, nous appelons ce processus un « changement de contexte » (context switch en anglais). Les changements de contexte sont coûteux et ralentissent le processus de calcul car l'état et le flux entiers du CPU doivent être arrêtés, sauvegardés et échangés contre un autre état (ainsi que d'autres caches, registres, etc.). Ce qui peut coûter du temps, ce sont aussi les « cache miss ». Un cache miss apparaît lorsque les données ne sont pas stockées dans la hiérarchie locale du cache mémoire pour un accès rapide. Le CPU devra obtenir les données dans la RAM, ce qui entraîne une perte de temps significative. Le but du SMT est de tirer parti du parallélisme au niveau des instructions et du parallélisme au niveau des threads en émettant des instructions de différents threads dans le même cycle (Eggers *et al.* 1997), sans passer de l'un à l'autre (sans avoir besoin de changer tout le contexte). Pour ce faire, il y a un état complet pour chaque cœur logique, comme mentionné précédemment (donc duplication de ce matériel dans le cœur physique). Le processeur peut conserver l'état des deux threads simultanément, et peut facilement passer de l'un à l'autre sans perte de temps. Les cœurs logiques partagent la plupart des ressources physiques du CPU, mais la duplication de l'architecture permet l'autonomie des deux cœurs logiques avec seulement 5 % de puissance et de taille de puce supplémentaires (selon le document d'Intel). Ce dernier montre que la technologie HT offre environ 20 % d'amélioration des performances sur l'exploitation des bases de données et les services web. Cependant, ces informations sont désormais assez anciennes.

Plusieurs articles traitent du SMT et de l'affinité. Si le concept de SMT est décrit dans (Marr *et al.* 2002), l'article (Leng *et al.* 2002) étudie l'impact du SMT dans le cas spécifique des applications MPI. Il nous donne quelques conclusions : le SMT permet une meilleure utilisation des ressources du processeur, mais, d'un autre côté, elle peut également entraîner une surcharge supplémentaire. Certaines études se sont concentrées sur le premier processeur SMT (plus spécifiquement l'hyper-threading), le Pentium 4. L'article (Tuck et Tullsen 2003) montre une amélioration de 20 %, comme prévu dans l'article d'Intel (Marr *et al.* 2002), et (Bulpin et Pratt 2004) ont obtenu des résultats similaires. Une bonne étude de 2011 a testé l'impact du SMT sur l'utilisation du processeur avec quatre applications réelles de la NASA (Saini *et al.* 2011). Elle montre que le SMT a conduit à une meilleure utilisation des processeurs, ce qui a entraîné une augmentation des performances globales lorsque des ressources mémoire suffisantes sont disponibles (cache et bande passante mémoire).

En ce qui concerne l'affinité, il existe également des articles sur ce sujet ; dans (Foong *et al.* 2004), les auteurs travaillent sur les connexions TCP sur un serveur SMP. Cette étude a obtenu un bon impact sur les performances en utilisant l'affinité, environ 20 %. Ce cas est spécifique pour les applications de communication avec le protocole TCP. Dans (Kazempour *et al.* 2008), l'étude est proche de nos considérations. Ils ont essayé de déterminer les performances de l'affinité sur les puces multicœurs. Ils l'ont mesurée sur des uniprocesseurs multicœurs et des multiprocesseurs multicœurs selon leur terminologie. La différence réside dans l'architecture physique : dans le premier cas, il n'y a qu'un seul socket, et dans le second cas, il peut y avoir deux sockets ou plus. Cela conduit au fait que les caches L2 et L3 peuvent être entièrement partagés entre tous les cœurs ou, dans le second cas, plusieurs caches L2 et L3 peuvent être partagés entre un sous-ensemble de cœurs, donc nous pouvons également utiliser l'affinité au niveau du cache L2 ou L3. Leurs résultats montrent que l'affinité n'apporte aucune amélioration sur les uniprocesseurs multicœurs (un seul processeur avec plusieurs cœurs et un seul L2 généralement utilisé sur un nœud à socket unique). L'affinité apporte un gain de 11 % en moyenne pour les multiprocesseurs multicœurs (nœuds avec 2 sockets ou plus, chacun d'eux fournissant leur propre cache L2 pour leurs multicœurs). Une amélioration a été observée avec l'affinité avant l'existence des multicœurs (assignation de tâches aux processeurs physiques sur un système multiprocesseur). Cet article a été publié en 2008, et les puces multicœurs ont évolué depuis. Bien que (Foong *et al.* 2004) et (Kazempour *et al.* 2008) parlent des résultats obtenus avec l'affinité sur la même carte mère (gestion de l'affinité pour mieux travailler avec la mémoire cache), nous pouvons également penser à l'affinité à une plus grande échelle pour les processus communicants. Dans (Bordage et Jeannot 2018), ils travaillent à une plus grande échelle sur des clusters de calcul. Cela nécessite de connaître exactement l'architecture du cluster de calcul que nous utilisons, où exactement nos programmes vont être exécutés et où se trouvent les communications intenses entre les processus. Sans une interconnexion rapide et équilibrée du cluster, plus la distance entre deux points de communication est grande, plus le temps nécessaire sera long. Cet article (Bordage et Jeannot 2018) se place le contexte d'un cluster de calcul qui peut avoir la taille d'une pièce par exemple (un petit superordinateur par exemple). Connaître l'architecture physique exacte du cluster peut être utile pour comprendre le comportement de certains programmes parallèles, où se trouvent physiquement les racks et les nœuds dans la pièce, sur quels racks sont placés les nœuds X et Y par exemple. Lorsque les scientifiques soumettent leurs travaux sur des clusters de calcul, ils ont rarement cette information et ils ne devraient pas s'en préoccuper. Comme dit précédemment, une interconnexion rapide et équilibrée résout la plupart des questions soulevées par cet article.

Dans (Bononi *et al.* 2006), les auteurs ont étudié l'impact des performances du SMT pour les simulations parallèles. Ils ont montré que l'impact du SMT sur les performances était influencé par « les caractéristiques de l'architecture logicielle d'exécution (c'est-à-dire l'implémentation mono-thread/multi-thread) » et que le SMT augmentait légèrement les performances. Dans (Tikir *et al.* 2007), ils ont utilisé des benchmarks pour prédire les performances des applications limitées par la mémoire sur les clusters. Dans (Gilbert *et al.* 2005), ils ont étudié l'impact de plusieurs technologies, y compris l'hyper-threading, sur les simulations de physique des hautes énergies (HEP) sur la grille de calcul pour le CERN. Ils ont travaillé sur les simulations de détecteurs GEANT4. Leurs résultats ont montré que l'hyper-threading causait des surcharges inutiles et ne devrait pas être utilisé en raison d'une perte de performances. Enfin, pour (Celebioglu *et al.* 2004), ils étudient l'effet du SMT sur les applications basées sur MPI tout en testant différentes bibliothèques mathématiques. Ils concluent que le SMT peut améliorer ou dégrader les performances selon la configuration matérielle et les applications que nous exécutons. Pour une bibliothèque mathématique, qui utilise de nombreuses ressources CPU, le SMT peut alors diminuer les performances, car l'activation du SMT partagera le cache d'un cœur physique pour deux cœurs logiques, et entraînera plus de cache miss. C'est pourquoi nous pensons qu'étudier l'efficacité du SMT est toujours pertinent, car nous ne devrions pas supposer que cela est bénéfique dans tous les cas.

La plupart des travaux connexes que nous avons présentés proviennent d'articles publiés entre 2000 et 2010. Comme les technologies évoluent rapidement, des études plus récentes sont également importantes, sans minimiser l'importance des articles plus anciens. Dans les articles récemment publiés sur le SMT, nous ne trouvons pas d'articles qui considèrent l'évaluation des performances du SMT par rapport aux machines avec SMT désactivé. En fonction du type d'applications exécutées sur les clusters, le bénéfice du SMT n'est pas toujours présent. Osborne et ses collègues ont publié plusieurs articles sur le SMT pour les applications en temps réel à partir de 2019. Nous pouvons citer (Osborne *et al.* 2019), (Bakita *et al.* 2021), (Osborne *et al.* 2020), où ils supposent que le SMT offre un avantage global en termes de performances. Partant de ce postulat, ils visent à obtenir une meilleure utilisation des ressources SMT en optimisant la planification, dans le cas particulier des applications en temps réel. Ces articles plus récents ne sont pas pertinents pour notre étude, car nous nous interrogeons principalement sur les avantages de l'activation ou de la désactivation du SMT sur un cluster de calcul, où nous devrions considérer les types d'applications exécutées, ou la configuration matérielle (comme la mémoire ou l'interconnexion), pour savoir si le SMT est efficace. Avec les mêmes

considérations que les travaux récents précédents, (Chen *et al.* 2018) visent également à optimiser l'utilisation des ressources offertes par le SMT ; ils proposent une solution affectant la phase de compilation. De plus, en 2020, (Ide et Yamasaki 2020) voulaient optimiser les politiques de récupération (de la mémoire RAM vers le cache) avec l'apprentissage automatique. Tous ces articles récents supposent la valeur du SMT pour améliorer les performances. Un petit article récent est plus connecté à nos questions de recherche. Dans (Hoo *et al.* 2022), les auteurs étudient si les utilisateurs doivent activer ou désactiver le SMT sur le fournisseur de services cloud. Ils concluent « parfois, désactiver le SMT est une bonne option pour améliorer les performances de certaines charges de travail. Par conséquent, nous devons toujours tester nos charges de travail avec différents réglages SMT et déployer avec le réglage qui apporte de meilleures performances » ; cela est plus similaire à ce que (Gilbert *et al.* 2005) a obtenu, et à ce que nous étudions. Nous proposons de remettre en question le fait que le SMT est toujours bénéfique pour les performances. En effet, comme nous le voyons, les articles évaluant les performances du SMT sont assez anciens maintenant, et ne sont pas non plus reproductibles (car à l'époque, le partage de code et les bonnes pratiques n'étaient pas considérés comme importants comme ils le sont maintenant). De plus, les puces multicœurs ont évolué depuis, et nous pouvons même les appeler des puces « manycore ». Nous proposons d'actualiser nos connaissances, par une expérience pratique, pour aider à la prise de décision des gestionnaires de clusters.

Dans les propositions que nous mettrons en œuvre au chapitre suivant, nous répondrons aux questions suivantes : le SMT est-il vraiment utile aujourd'hui, à l'ère des multicœurs (ou manycore) ? Est-il adapté au HPC et plus particulièrement aux grandes simulations stochastiques ? Elles présentent parfois des besoins de mémoire énormes et parfois des besoins de calcul massifs. La gestion de l'affinité CPU par l'utilisateur augmente-t-elle les performances ? Quels conseils pouvons-nous donner aux utilisateurs non experts des clusters de calcul ? Pouvons-nous aider les administrateurs système dans leurs décisions d'activer ou de désactiver le SMT en fonction du profil de l'application ? Dans notre contexte de simulations stochastiques parallèles, nous avons besoin d'applications reproductibles et répétables pour répondre aux questions précédentes.

III.7. Pour des expériences performantes et reproductible avec l'utilisation de bibliothèque scientifique de transformation de fourrier

Les applications scientifiques, en particulier en physique, utilisent fréquemment Fortran, un langage de programmation ancien mais efficace. Intel a récemment rendu son compilateur propriétaire, ifort, disponible gratuitement et a introduit un nouveau compilateur, ifx, conçu pour succéder à ifort. Nos objectifs sont doubles : premièrement, illustrer un exemple concret des défis de la non-répétabilité et de la non-reproductibilité que rencontrent de nombreux chercheurs en calcul haute performance ; et deuxièmement, évaluer les performances en termes de temps et de consommation d'énergie de trois compilateurs : gfortran, ifort et ifx. Nous comparons également ces variables avec les implémentations largement reconnues de l'algorithme de transformée de Fourier discrète, FFTW et MKL. Nous proposons de fournir des informations techniques de performance et de reproductibilité pour les spécialistes des simulations en physique, afin de faciliter la prise de décision concernant l'implémentation de leurs simulations. Dans cette étude, nous nous concentrons sur une application de physique connue sous le nom de QDD (Quantum Dissipative Dynamics) (Dinh *et al.* 2022).

QDD est un logiciel conçu pour simuler le comportement des électrons et des ions dans des systèmes finis tels que les atomes, les molécules et les clusters sous l'influence de champs électromagnétiques externes. Un aspect clé de QDD est son accent sur la dynamique dissipative, qui concerne les processus de perte d'énergie dus aux interactions entre les électrons, en particulier les collisions entre électrons. Ces interactions conduisent à des corrélations dynamiques qui peuvent maintenant être modélisées dès le début du processus d'excitation en utilisant la mécanique quantique, spécifiquement à travers l'approximation du temps de relaxation (RTA). Le concept physique sous-jacent à QDD repose sur les principes de la théorie de la fonctionnelle de la densité dépendante du temps (TDDFT), en utilisant en particulier l'approximation de la densité locale dépendante du temps (TDLDA). Cette approche est complétée par une correction d'auto-interaction (SIC) qui est indispensable pour prédire avec précision l'émission d'électrons, un phénomène où les électrons sont éjectés du système en raison de l'énergie apportée par des champs électromagnétiques externes tels que les lasers. La méthode de calcul intègre des traitements quantiques de la dynamique des électrons avec la dynamique moléculaire classique pour les ions. Cela implique l'utilisation d'une grille 3D pour cartographier les états électroniques et d'employer des transformées de Fourier rapides (FFT)

pour passer de l'espace réel à l'espace des moments. L'inclusion de conditions aux limites absorbantes aide à modéliser avec précision l'émission d'électrons.

La transformée de Fourier rapide (FFT) est un algorithme qui calcule la transformée de Fourier discrète (DFT) d'une séquence. L'analyse de Fourier convertit un signal de son domaine d'origine (souvent le temps ou l'espace) à une représentation dans le domaine fréquentiel et vice versa. Cette opération est utile dans de nombreux domaines, mais la calculer directement à partir de la définition est souvent trop lent en pratique. C'est pourquoi nous utilisons des implémentations connues de la FFT. FFTW (Frigo et Johnson 2005) est l'une des plus anciennes et des plus utilisées bibliothèques open source pour réaliser la DFT (Donoho et Stodden 2015 ; Nikolić *et al.* 2014). D'autre part, Intel offre une solution propriétaire avec la Math Kernel Library (MKL) qui est gratuite et peut également réaliser des transformées de Fourier rapides, avec son interface de programmation d'application (API) adaptée pour les codes utilisant FFTW (vous pouvez facilement passer de FFTW à MKL).

Pour accélérer le calcul, nous pouvons également le paralléliser. QDD utilise OpenMP. OpenMP, abréviation d'Open Multi-Processing, est une API qui prend en charge la programmation parallèle partagée multiplateforme en C, C++ et Fortran sur de nombreux types d'architectures de processeurs. Elle est conçue pour permettre aux programmeurs de développer des applications parallèles plus facilement en fournissant une interface pour définir des régions parallèles, des boucles et des sections de code. OpenMP est particulièrement apprécié dans la communauté du calcul scientifique pour sa facilité d'utilisation.

L'utilisation de Fortran en conjonction avec OpenMP et un algorithme de transformée de Fourier rapide est courante dans le calcul haute performance pour la physique. Cependant, nous avons observé des problèmes de répétabilité et de reproductibilité dans de tels programmes. De plus, il y a une pression croissante pour améliorer la vitesse de calcul tout en réduisant la consommation d'énergie.

Plusieurs études ont abordé les performances, la consommation d'énergie et la reproductibilité des outils de calcul dans le domaine du calcul haute performance. (Memeti *et al.* 2017) soulignent que la programmation avec OpenMP nécessite moins d'efforts comparée à OpenCL et CUDA, avec des performances et des efficacités énergétiques similaires observées sur ces plateformes. Ils ont également établi une corrélation entre le temps de calcul et la consommation d'énergie : les programmes qui prennent plus de temps à calculer consomment généralement plus d'énergie.

(Pereira *et al.* 2017) ont examiné le temps de calcul et la consommation d'énergie sur plusieurs benchmarks, concluant que les langages qui s'exécutent plus rapidement tendent également à être plus écoénergétiques. Leurs résultats montrent des similitudes fortes dans les classements du temps et de la consommation d'énergie. Cette étude positionne également Fortran comme modéré en termes d'efficacité énergétique et temporelle, se classant respectivement 10e et 14e sur 27. Malgré cela, Fortran reste compétitif en HPC en raison de l'optimisation et de la disponibilité de bibliothèques scientifiques comme OpenMP et FFTW, plus que beaucoup d'autres langages, sauf C et C++.

Les recherches sur les algorithmes de DFT révèlent que l'implémentation open-source de la transformée de Fourier rapide, FFTW, fréquemment utilisée dans les applications scientifiques et d'ingénierie pour ses hautes performances, est comparable aux performances des bibliothèques fournies par les vendeurs comme MKL, qui est récemment devenue gratuite (Frigo et Johnson 2005). Dans l'étude de (Gambron et Thorne 2020), il est suggéré que bien que FFTW et MKL montrent des résultats comparables dans des exécutions unidimensionnelles, et MKL surpasse souvent dans des scénarios multidimensionnels et parallèles. Dans (Nikolić *et al.* 2014), les auteurs écrivent en parlant de FFTW : « L'un des algorithmes les plus fréquemment utilisés dans les applications scientifiques et d'ingénierie est la transformée de Fourier rapide (FFT). Son implémentation open source (Fastest Fourier Transform of the West, FFTW) est largement utilisée, principalement en raison de ses excellentes performances, comparables aux bibliothèques fournies par les vendeurs. »

En ce qui concerne les différents compilateurs Fortran, les benchmarks Polyhedron (Polyhedron benchmarks, n. d.) indiquent que le compilateur Intel performe généralement mieux que gfortran, bien que ces résultats doivent être considérés avec prudence en raison du biais potentiel puisqu'ils proviennent d'un vendeur. Dans (Young-S *et al.* 2017), les auteurs comparent ifort et gfortran, en concluant qu'ifort est globalement plus performant, sur un seul cœur et sur plusieurs cœurs CPU.

Concernant la reproductibilité, tout en essayant d'atteindre des performances élevées, (Charpilloz *et al.* 2017) décrivent les difficultés pour obtenir la reproductibilité avec les programmes Fortran, en soulignant des problèmes tels que le non-déterminisme introduit par certaines optimisations du compilateur et la variabilité des résultats de calcul due aux différentes architectures matérielles et comportements des compilateurs. Des recherches supplémentaires par (Li *et al.* 2016) présentent des options de compilateur pour améliorer la

répétabilité entre les compilateurs GNU et Intel, que nous avons appliquées dans notre travail pour tester la reproductibilité parmi différents compilateurs.

Nous proposons de déterminer si FFTW ou MKL est supérieur en termes de reproductibilité numérique, d'efficacité temporelle et de consommation d'énergie. Des questions similaires s'appliquent aux compilateurs ifort, ifx et gfortran. Nous explorons également dans le chapitre suivant comment atteindre la reproductibilité numérique.

III.8. Les nouveaux paradigmes de calcul : L'informatique quantique

III.8.1. L'informatique quantique

L'informatique quantique exploite les propriétés uniques de la mécanique quantique, telles que la superposition et l'intrication, pour effectuer des calculs. Ce changement de paradigme par rapport à l'informatique classique ouvre de nouvelles frontières dans divers domaines. Si la cryptographie a des enjeux considérables, l'optimisation et la recherche opérationnelle en générale n'est pas en reste.

Dans le domaine de l'informatique quantique, des algorithmes tels que celui de Grover offrent un aperçu du potentiel des systèmes quantiques pour traiter les informations de manière fondamentalement différente des ordinateurs classiques. L'algorithme de Grover, en particulier, est connu pour sa capacité à rechercher dans des bases de données non triées de manière quadratiquement plus rapide que ses homologues classiques, ce qui représente un bond significatif en efficacité de calcul.

L'informatique quantique fait face à de très nombreux défis pratiques. Entre autres, obtenir les mêmes conclusions scientifiques est un pilier épistémologique pour l'avancement de la Science. Les machines quantiques sont stochastiques par essence, chaque exécution produira potentiellement un résultat différent, mais la reproductibilité (et non la répétabilité) reste le principal critère pour vérifier la qualité des machines. Différentes expériences quantiques identiques doivent donner des résultats statistiquement similaires (alors que l'on peut attendre une répétabilité bit à bit pour les calculateurs classique déterministe, cette dernière étant d'ailleurs essentielle pour déboguer les programmes).

III.8.2. Les machines quantiques actuelles

Les principales technologies de qubits comprennent les qubits supraconducteurs, les ions piégés, les qubits semi-conducteurs, les atomes neutres et les qubits topologiques. Les qubits supraconducteurs, largement utilisés, reposent sur le comportement quantique des circuits électriques supraconducteurs fabriqués sur des puces en silicium, offrant de longs temps de cohérence et un bon passage à l'échelle, en faisant des candidats prometteurs pour les ordinateurs quantiques à grande échelle. Les ions piégés, encodés dans les états électroniques internes d'ions individuels piégés dans des champs électromagnétiques, sont manipulés par des faisceaux laser et offrent également de longs temps de cohérence et des opérations de fiables, bien que leur extension à un grand nombre de qubits reste un défi. Les qubits semi-conducteurs utilisent les propriétés des électrons confinés dans des matériaux semi-conducteurs, comme le silicium ou l'arséniure de gallium, et bénéficient de la compatibilité avec les techniques de fabrication de semi-conducteurs existantes, avec un potentiel d'intégration avec l'électronique classique. Les atomes neutres, encodés dans les états électroniques internes d'atomes neutres individuels maintenus dans des réseaux ou pièges optiques, sont également manipulés par des faisceaux laser et présentent des défis similaires en termes d'évolutivité et de connectivité entre les qubits. Enfin, les qubits topologiques, qui reposent sur les propriétés topologiques de matériaux exotiques comme les isolants topologiques ou les fermions de Majorana, en sont encore aux premiers stades de recherche et développement.

III.8.3. Etude de reproductibilité des machines quantiques actuelles

Nous avons retenu l'algorithme de Grover comme point de repère pour évaluer les possibilités des machines quantiques mises gracieusement à la disposition de la recherche par IBM. L'informatique quantique fait face à de très nombreux défis pratiques. Entre autres, obtenir les mêmes conclusions scientifiques est un pilier épistémologique pour l'avancement de la Science. Différentes expériences quantiques identiques doivent donner des résultats statistiquement similaires (alors que l'on peut attendre une répétabilité bit à bit pour les calculateurs digitaux, cette dernière étant d'ailleurs essentielle pour déboguer les programmes). Il est possible de tester cette reproductibilité sur différentes plateformes et environnements.

Notre proposition vise à étudier le potentiel de reproductibilité des résultats l'algorithme de Grover sur différentes machines IBM, qui sont les seules librement accessibles, afin d'établir la fiabilité actuelle des machines quantiques.

III.9. Conclusion

Dans ce chapitre, nous avons fait différentes propositions pour renforcer la recherche reproductible sur plusieurs cas d'étude où des questions se présentaient à nous. Nous avons d'abord présenté des recommandations et des bonnes pratiques que nous avons identifiées comme nécessaires pour améliorer la reproductibilité en général. Puis, nous avons proposé de développer une modélisation stochastique reproductible de pandémie face à l'absence de reproductibilité des modèles existants. Nous avons souligné l'importance cruciale d'un modèle fiable pour ces circonstances. Nous avons ensuite proposé d'explorer comment garantir des expériences répétables en machine learning en maîtrisant l'utilisation des générateurs de nombres pseudo-aléatoires qui sont au cœur de cadriciels d'apprentissage. Nous avons également fait un focus particulier sur la reproductibilité des générateurs pseudo-aléatoires (PRNGs) avec les technologies actuelles. Nous avons proposé par la suite d'optimiser la parallélisation des PRNGs, afin d'améliorer l'efficacité des calculs. Par ailleurs, nous avons proposé d'étudier l'impact du simultaneous multi-threading sur la reproductibilité en termes de temps de calcul et de consommation énergétique en fonction de différents profils d'applications (memory bound / compute bound). Ensuite, nous avons discuté des bonnes pratiques pour des expériences performantes et reproductibles lorsque celles-ci utilisent les bibliothèques scientifiques dédiées à la transformation de Fourier. Enfin, nous avons abordé les questions liées à l'état actuel de la reproductibilité d'un nouveau paradigme de calcul avec l'informatique quantique, afin de mettre en lumière les défis et opportunités qu'il présente pour le futur du calcul à haute performance. Pour répondre à ces questions et pour évaluer la pertinance ces propositions faites, la mise en œuvre des cas d'étude est présentée dans le chapitre suivant.

IV. Mise en place des propositions sur plusieurs applications

IV.1. Introduction

Nous allons maintenant détailler la mise en place de chaque expérience, ainsi que les résultats obtenus. Tout d'abord, nous avons développé un modèle épidémiologique reproductible en C++, permettant d'effectuer des simulations en parallèles, à différentes géographiques suivant les capacités matérielles de la machine. Le modèle est largement paramétrable. Ensuite, nous avons étudié la qualité des flux de nombres pseudo-aléatoires du générateur Mersenne Twister selon la méthode de parallélisation utilisée. Afin de répondre aux questions concernant la génération de nombre aléatoire dans les frameworks du machine learning, nous avons comparé la génération des deux principaux frameworks, Pytorch et Tensorflow, avec leurs homologues dans leur version originale, et nous avons pu observer les différences en terme de performance, de consommation énergétique, de qualité statistique, et de reproductibilité numérique. Nous avons également réalisé des expériences sur des processeurs avec SMT et sans SMT, afin d'étudier les temps de calculs suivant le type d'applications. De même pour FFTW et MKL, ainsi que les différents compilateur Fortran, dans le cadre d'une application de physique fondamentale. Enfin, nous avons étudié l'execution d'algorithme quantique sur des machines quantiques réelles, afin de pouvoir évaluer les niveaux de fiabilité de celles-ci.

IV.2. Développement d'un modèle multi-agent pour des simulations épidémiologiques

IV.2.1. Notre modèle

IV.2.1.1. Fonctionnement du modèle

Les approches mathématiques réductionnistes sont très pertinentes pour le travail au niveau national (Mathiot *et al.* 2021 ; Salje *et al.* 2020). Elles sont complémentaires aux modèles individu-centré qui permettent une compréhension détaillée des impacts des politiques de santé. Pour notre étude, nous avons choisi de construire un modèle multi-agents capable d'étudier l'évolution de la Covid-19 avec des comportements spatialisés et des caractéristiques individuelles. Malgré la prolifération de modèles similaires pendant la pandémie de 2020/2021, nous reconnaissons notre contribution modeste mais soulignons notre engagement à fournir un

modèle reproductible. Ce modèle prend en compte les principales mesures préventives et est conçu pour une utilisation simple, permettant aux utilisateurs de valider les résultats attendus et de mener facilement leurs expériences sur diverses interventions de santé. Ce travail a été présenté à la conférence sur la santé Giseh (Antunes et Hill 2022).

Nous avons choisi la division administrative (département) comme échelle spatiale de notre modèle, laquelle peut être ajustée aux échelles régionale ou nationale en fonction des ressources de calcul disponibles, ou réduite à l'échelle de la ville pour une analyse plus fine. Notre espace de simulation est une matrice carrée de manière à ce que chaque case représente un mètre carré. Initialement, certains humains sont assignés à une ville, tandis que d'autres sont placés aléatoirement, reflétant la distribution inégale de la population entre les zones urbaines et rurales, ce qui influence la propagation du virus, principalement dans les villes densément peuplées. Dans notre modèle, les humains citadins sont confinés à leur ville, tandis que les autres peuvent se déplacer librement. Les humains se téléporte de case en case, aléatoirement, selon la restriction de zone et le nombre de déplacement qu'ils doivent faire par jour. Les tailles et populations des villes sont entièrement personnalisables, avec des mouvements humains aléatoire à chaque étape de simulation, soumis à des mesures préventives modifiables. Par exemple, dans le cas d'un département comme Paris d'une superficie de 10 km², nous avons une carte de 10 000 x 10 000 cellules, avec une seule ville occupant toute la carte.

L'échelle temporelle de notre modèle est la journée, permettant des ajustements quotidiens. Le modèle catégorise les agents humains par groupes d'âge reflétant la population française, avec des états allant de sain à asymptomatique, malade (à domicile, à l'hôpital ou en soins intensifs), ou rétabli (ou mort). Cette granularité permet une analyse quotidienne de la progression de la maladie et de l'efficacité de diverses mesures (par exemple, confinements, port du masque, désinfection des mains, couvre-feux, vaccinations). Tous les paramètres sont accessibles et ajustables, permettant des mises à jour en fonction des nouvelles découvertes sur l'efficacité des vaccins, la protection des masques ou les variants du virus. Les utilisateurs peuvent configurer les calendriers des campagnes de vaccination, les changements de variants et les périodes de confinement, y compris l'introduction de « super-contaminateur » analogues aux modèles de propagation du feu ou du cancer (Filippi *et al.* 2010 ; Franceschini *et al.* 2018 ; Innocenti *et al.* 2009).

La simulation peut commencer avec un nombre spécifié d'individus infectés ou à partir d'un hypothétique « patient zéro ». La propagation de l'infection est modélisée en utilisant un voisinage de Moore d'ordre 2, avec des probabilités de contagion ajustables. Les mesures

préventives telles que les masques, les désinfectants et les vaccins sont prises en compte pour réduire le risque de transmission. Après l'infection initiale, un individu reste asymptomatique pendant deux jours, capable de transmettre le virus, suivi d'une phase symptomatique de neuf jours. La période infectieuse totale s'étend sur onze jours, avec des taux de contagion variables adaptables aux nouvelles données sur les variants du virus.

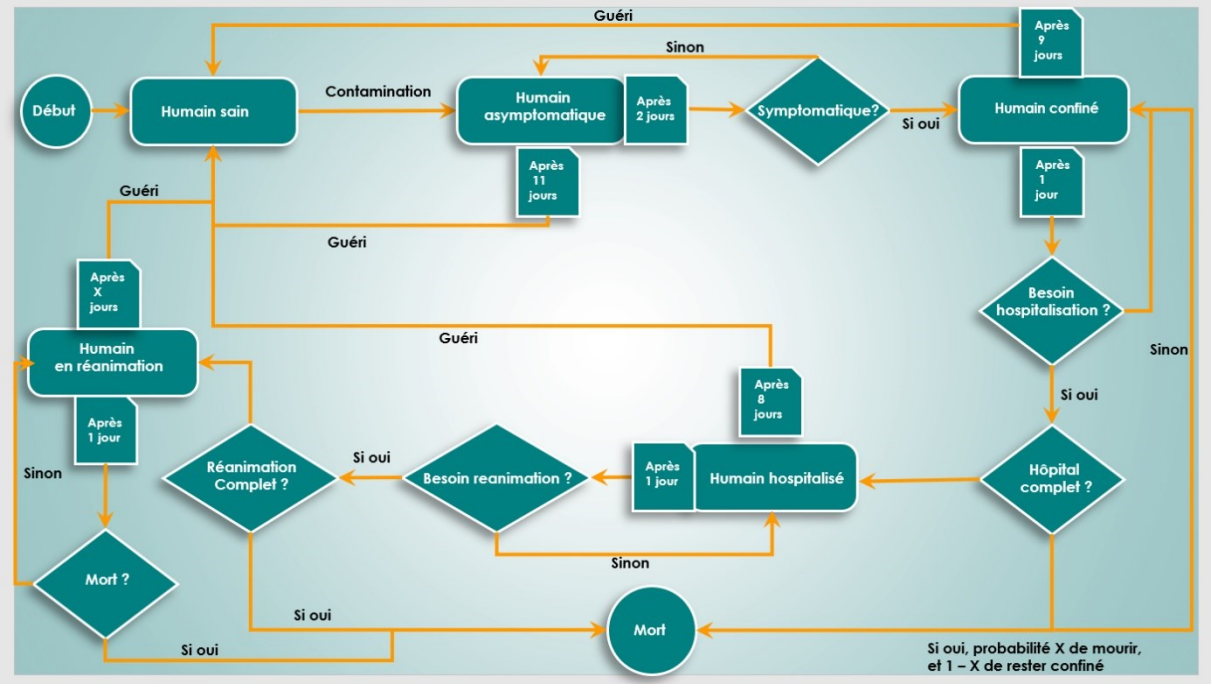


Figure 7: Diagramme du flux

Après une période de deux jours durant laquelle un individu se déplace et peut potentiellement propager le virus, une décision est prise : il peut soit développer des symptômes et s'isoler en raison de la maladie, soit rester asymptomatique et continuer à se déplacer, risquant ainsi de propager davantage le virus. Après la période d'infection de onze jours, un individu est considéré comme guéri et acquiert une résistance accrue au virus, sauf dans les cas graves nécessitant des soins intensifs. Pour les individus symptomatiques en auto-isolement, des évaluations quotidiennes déterminent si une hospitalisation est nécessaire. La saturation hospitalière augmente le risque de mortalité pour ces patients. Ceux nécessitant des soins intensifs sont confrontés à un dilemme similaire ; les unités de soins intensifs surpeuplées augmentent la probabilité de mortalité. Les survivants de cette phase critique sont considérés comme guéris. Ce modèle permet l'analyse de diverses stratégies de santé publique. Tout ceci est illustré dans la figure 7.

Le modèle stochastique utilise pour chaque choix aléatoire le générateur de nombres pseudo-aléatoires Mersenne Twister (Matsumoto et Nishimura 1998). Plusieurs méthodes de

parallélisation sont décrites dans (Hill 2015). L'espacement aléatoire (random spacing), le découpage de séquence (sequence splitting) et l'index de séquence MT sont décrits dans (Antunes *et al.* 2023), montrant une qualité similaire dans les flux stochastiques parallèles. Dans notre cas, nous avons utilisé le sequence splitting pour pouvoir effectuer des simulations parallèles reproductibles. Le sequence splitting consiste en des fichiers d'états préenregistrés du Mersenne Twister, avec lesquels nous initialiserons chaque réplication parallèle.

IV.2.1.2. Les paramètres

Comme décrit dans la présentation du modèle ci-dessus, certains éléments peuvent être paramétrés afin de faciliter la conception expérimentale. Les paramètres sont sauvegardés dans deux fichiers différents : les paramètres du contexte épidémiologique et les paramètres du contexte géographique (actuellement département français). Concernant le contexte géographique, il s'agit de la carte. Voici une liste non exhaustive :

- La taille de la carte, en mètre carrés ;
- Le nombre d'humains au début de la simulation ;
- Le nombre de personnes infectées au début ;
- Le nombre d'itérations de la simulation, chaque itération représentant un jour ;
- Le nombre de places d'hôpital et de soins intensifs ;
- Le nombre de déplacements des personnes par jour ;
- Les tranches d'âge de la population (8 groupes) : par exemple, les jeunes peuvent souffrir plus ou moins d'une maladie que les personnes plus âgées.
- Vous pouvez placer des hôpitaux, des restaurants, des salles de sport, des magasins et des clubs qui pourraient agir comme des clusters de contamination, en augmentant le taux de transmission ;
- Vous pouvez placer des villes, avec leur taille, leurs coordonnées et le nombre d'habitants. La carte sera alors adaptée au monde réel, avec des campagnes et des zones urbaines beaucoup plus denses, où l'épidémie peut croître.

Concernant le contexte épidémiologique, vous pouvez paramétrer :

- Le nombre de variant du virus, pour suivre l'évolution du virus (si vous étudiez une maladie sans mutations, vous pouvez fixer cette valeur à 1) ;
- Pour chaque variant, vous pouvez déterminer le taux de transmission jour par jour pour les personnes infectées, ainsi que la durée de la période de transmission. Vous pouvez également définir la durée de chaque variant ;

- Le pourcentage de personnes asymptomatiques, qui se déplaceront et contamineront les autres ;
- Le taux d'hospitalisation pour chaque variant, en fonction des groupes d'âge ;
- Le taux de mortalité dans les unités de soins intensifs ;
- La résistance immunitaire par groupe d'âge après une infection précédente et/ou après vaccination ;
- Un calendrier pour la vaccination, et le taux de personnes vaccinées à chaque période ;
- Vous pouvez choisir si des médicaments sont disponibles, avec leur pouvoir de protection, et un calendrier associé ;
- Vous pouvez également configurer des mesures non pharmaceutiques comme l'utilisation de masques, avec différents types de masques, confinement, couvre-feu, mouvements restreints et gel hydroalcoolique. Vous pouvez paramétrer le taux de chaque protection, le calendrier, etc. ;
- Le taux d'obésité et de diabète, avec l'augmentation de la mortalité associée, car nous savons que ces deux pathologies sont celles qui causent le plus de complications.

IV.2.2. Expériences

IV.2.2.1. Méthode d'expérimentation (bash, notebook, etc)

Développée en C++ pour des performances optimales sur les clusters de calcul, la simulation utilise un makefile pour la compilation, une pratique reconnue pour sa contribution à la reproductibilité (Schwab *et al.* 2000). Des scripts Bash facilitent l'exécution de multiples expériences, tandis que des Jupyter Notebooks servent d'interface conviviale pour exécuter les simulations et visualiser les résultats. Le projet propose deux versions du code, l'une avec une carte complète et l'autre utilisant une matrice creuse, pour équilibrer l'utilisation de la RAM par rapport à la vitesse de calcul, obtenant des résultats identiques au bit près entre elles. Le code est accessible sur Gitlab : https://gitlab.isima.fr/keodier/projet-epidemiologie-modifications.

Ce code a été exécuté sur plusieurs machines pour garantir la reproductibilité. Nous avons deux nœuds avec des processeurs AMD. Chaque nœud est équipé de deux processeurs AMD EPYC Rome 7452, chacun avec 32 cœurs, 64 threads, 2 Mo de cache L1, 16 Mo de cache L2, 128 Mo de cache L3 et 512 Go de RAM (DDR4). La bande passante mémoire maximale est de 190,7 Go/s. Chaque cœur peut fonctionner à 3,35 GHz. Le système d'exploitation est Ubuntu

20.04, avec la version de noyau 5.4.0.153-generic, et nous travaillons avec le compilateur gcc/g++ version 9.4.0. Pour Intel, nous avons également deux nœuds. Chaque nœud est équipé de deux processeurs Intel Xeon E5-2650 v2, chacun avec 8 cœurs, 16 threads, 32K de cache L1, 128K de cache L2 et 10 Mo de cache L3, ainsi que 128 Go de RAM. Le système d'exploitation est CentOS Linux 7.9.2009, avec la version de noyau 7.9.2009, et nous travaillons avec le compilateur gcc/g++ version 4.8.4. Le Turbo boost est activé, permettant de passer de 1,2 GHz à 3,4 GHz. Enfin, nous avons exécuté l'analyse de sensibilité sur une machine avec 96 cœurs physiques hyperthreadés (c'est-à-dire un total de 192 cœurs logiques), équipés de processeurs Intel(R) Xeon(R) Platinum 8160CPU @ 2.10GHz, et de 768 Go de RAM. Le système d'exploitation est une version Debian 4.19 (Linux).

IV.2.2.2. Impact des différents générateurs de nombrse pseudo-aléatoires

Pour évaluer l'impact de différents générateurs de nombres pseudo-aléatoires sur les résultats expérimentaux, nous avons mené la même expérience en utilisant plusieurs des meilleurs PRNG : Philox (Salmon *et al.* 2011), MRG32k3a (L'ecuyer 1999), PCG (O'neill 2014) et MT. Chaque PRNG a été répliqué 30 fois. Après la collecte des données, les résultats ont été analysés à l'aide d'une ANOVA pour tester l'égalité des variances. Plus précisément, nous avons comparé les amplitudes du premier pic d'infection à travers les simulations de différents PRNGs, en sélectionnant la valeur maximale du premier pic d'infection de chaque simulation pour l'analyse ANOVA.

L'ANOVA est utilisée ici pour tester l'hypothèse nulle « les moyennes des différents échantillons sont égales » contre l'hypothèse alternative « au moins un des échantillons diffère significativement de la moyenne globale ». Cette analyse se concentre sur l'égalité des moyennes des valeurs maximales des premiers pics d'infection parmi les échantillons, en supposant la normalité dans la distribution des résidus et l'homoscédasticité (variances égales entre les groupes).

L'ANOVA à un facteur teste si deux ou plusieurs groupes ont la même moyenne de population, appliquée aux échantillons de ces groupes, qui peuvent varier en taille.

Les résultats de l'ANOVA indiquent qu'il n'y a pas de différence statistiquement significative dans les moyennes des premiers pics d'infection à travers les différentes expériences utilisant différents PRNGs. La valeur F de 0.606 suggère que la variance entre les moyennes des groupes est similaire à la variance au sein des groupes, renforçant l'absence de différences significatives. De plus, la valeur p de 0.750, qui est bien au-dessus du seuil de

signification de 0.05, soutient cette conclusion. Ainsi, nous concluons que le choix du PRNG—que ce soit Philox, MRG32k3a, PCG ou MT—n'affecte pas significativement les résultats des expériences en termes des amplitudes des premiers pics d'infection. Cela conduit à la rétention de l'hypothèse nulle, confirmant que les moyennes des pics d'infection sont statistiquement indiscernables entre les différents PRNGs utilisés dans l'étude.

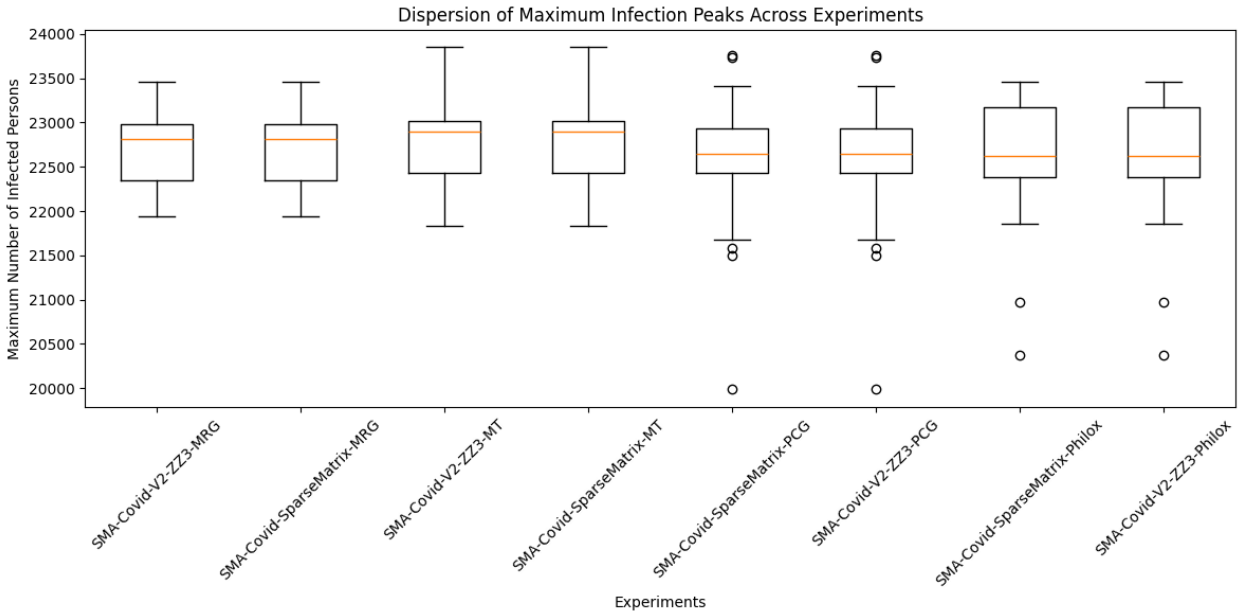


Figure 8: Aucune différence statistiquement significative observée sur le nombre de contamination lors du pic d'infection sur les différents PRNGs

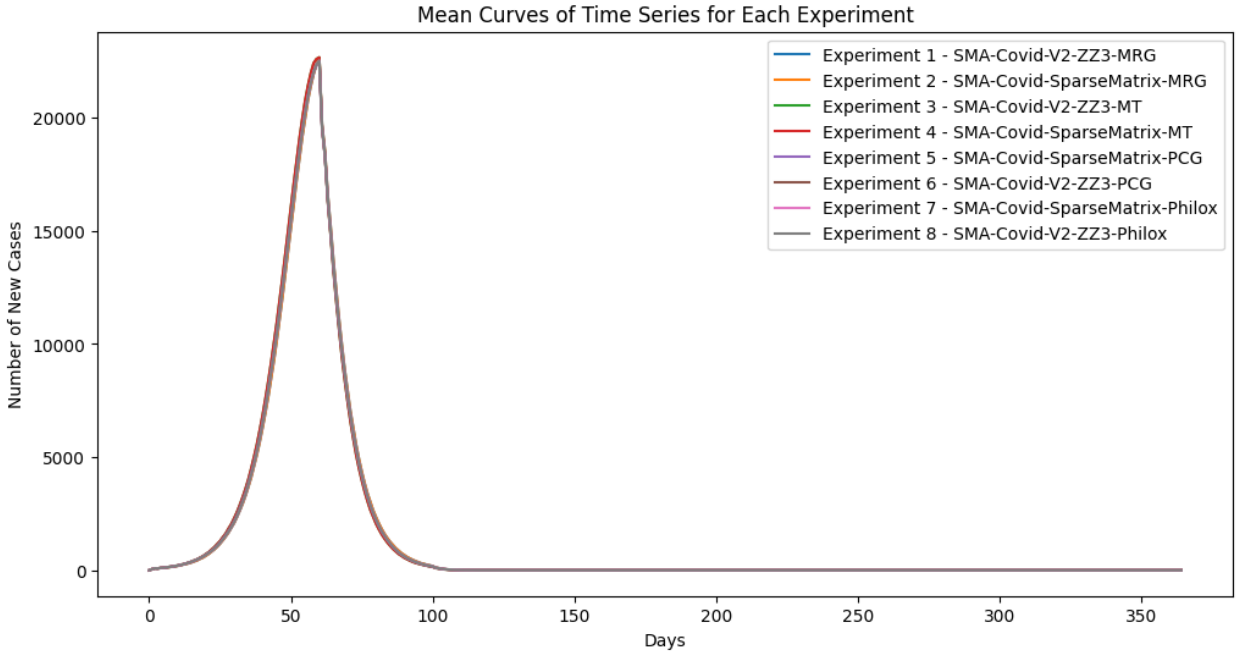


Figure 9: Toutes les courbes moyennes de nouveaux cas journaliers superposées

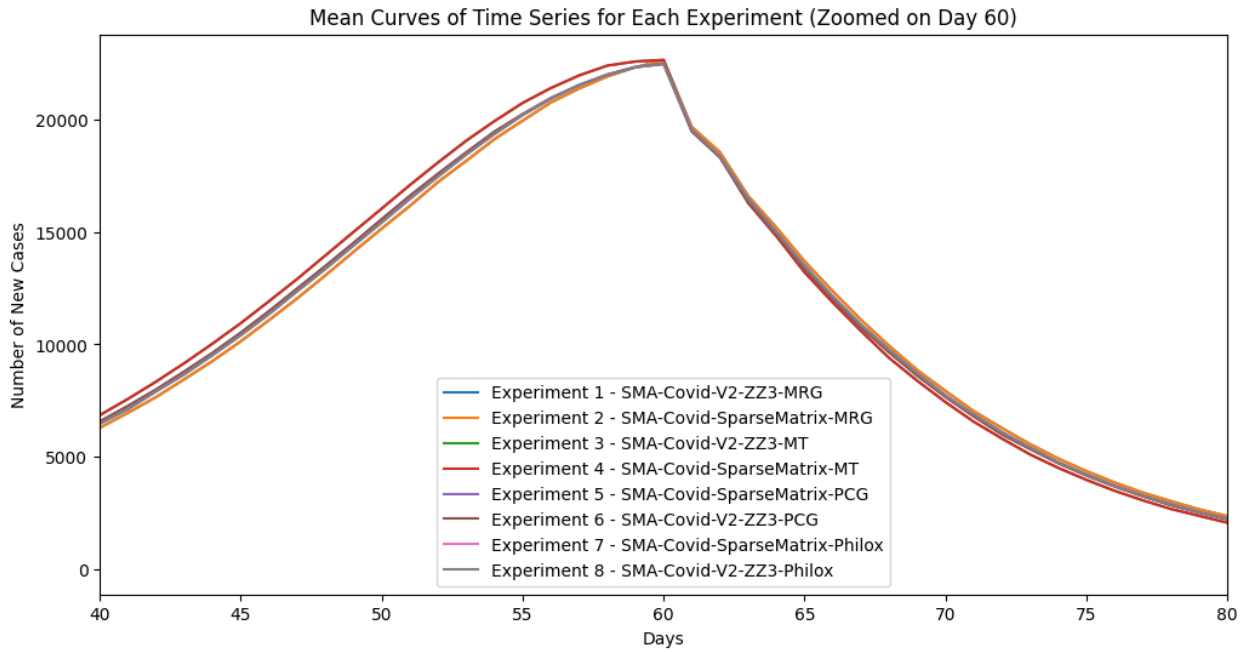


Figure 10: Zoom sur le pic

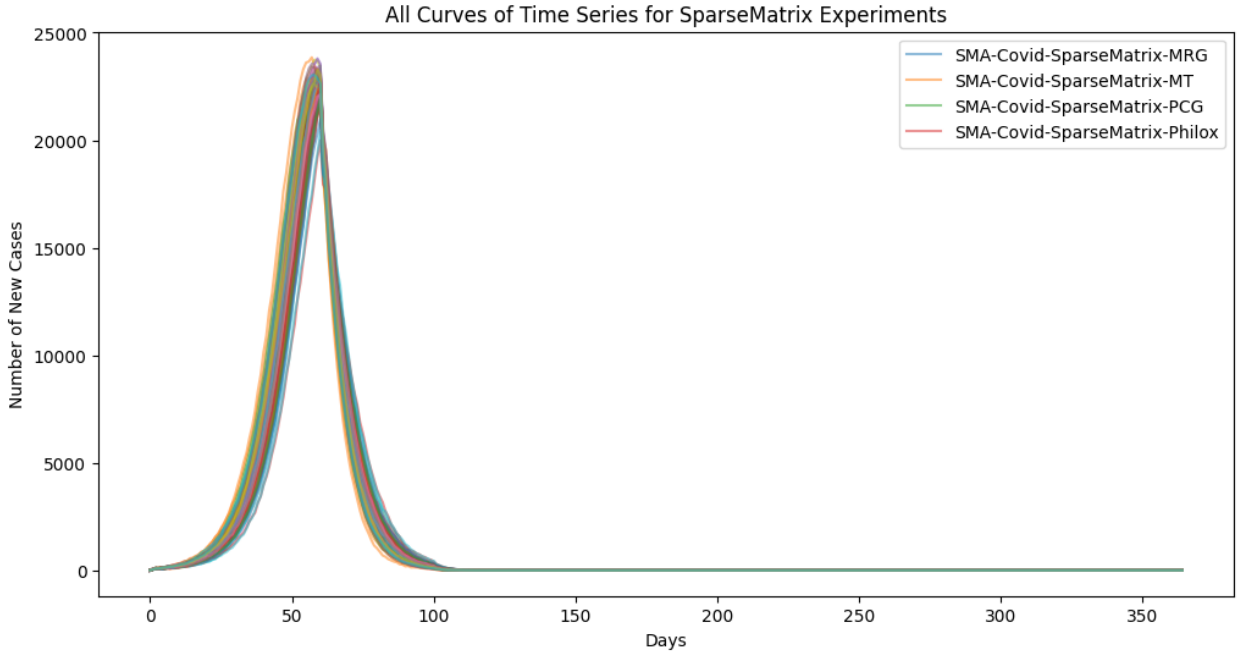


Figure 11: Toutes les courbes (chaque réplication)

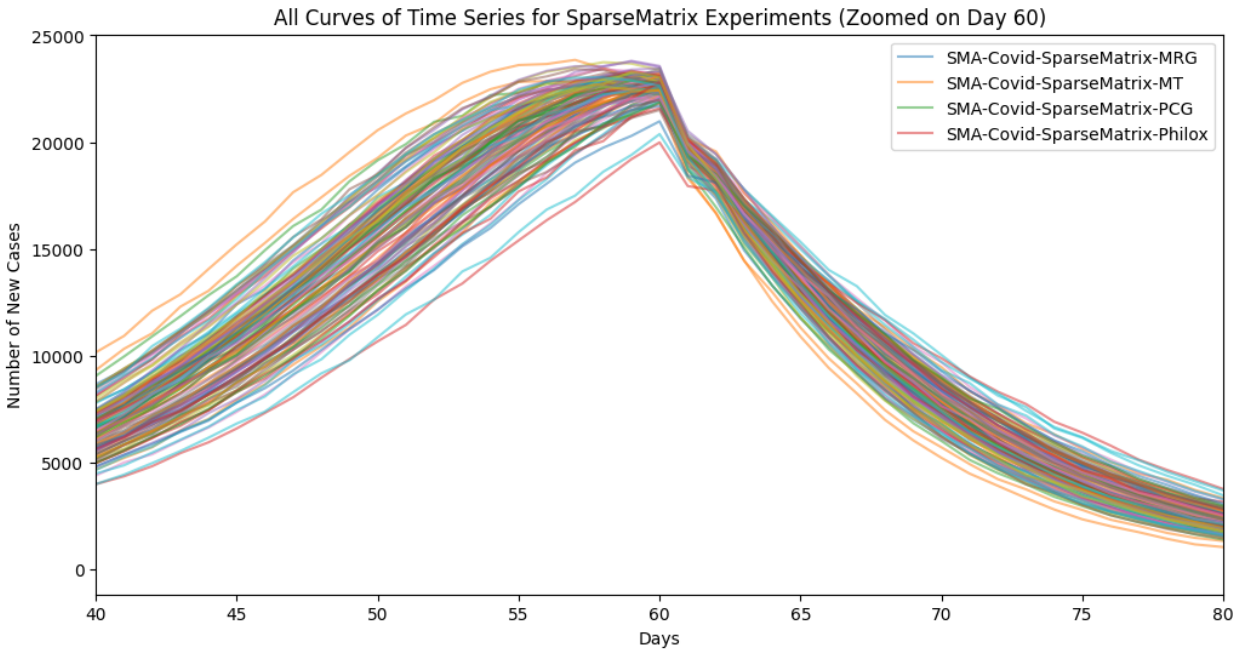


Figure 12: Zoom sur le pic de toutes les courbes

Pour obtenir une compréhension plus nuancée de l'évolution de l'épidémie au fil du temps, il ne suffit pas de comparer les amplitudes et les moments des pics. Nous utilisons le clustering pour analyser l'ensemble des données de séries temporelles de nos expériences, afin d'explorer leurs similitudes en termes de forme et de simultanéité. Le clustering nous permet de découvrir des motifs structurels dans des ensembles de données non étiquetés en organisant les données en groupes homogènes, en veillant à maximiser les similarités au sein de ces groupes et à maximiser les dissimilarités entre eux.

Dans cette étude, nous utilisons le clustering hiérarchique, une méthode qui ne nécessite pas de pré-spécification du nombre de clusters, contrairement au k-means. Le clustering hiérarchique construit un modèle des données sous forme d'un arbre de clusters et ne requiert pas la sélection de centroïdes initiaux. Cette approche convient à nos besoins car elle regroupe progressivement les points de données en fonction de leur distance euclidienne, qui mesure la distance en ligne droite entre les points dans un espace multidimensionnel. Cette approche est la même que celle mise en place par Hanae Havion et Claude Mazel lors de notre travail sur la reproductibilité des SMA selon les technologies utilisées.

Le dendrogramme de la figure 13 (montre que toutes les expériences utilisant différents PRNG sont mélangées, donc nous ne pouvons pas différencier le PRNG utilisé à partir des

résultats (un zoom est nécessaire pour exploiter ce dendrogramme). Cela confirme les conclusions de l'ANOVA.

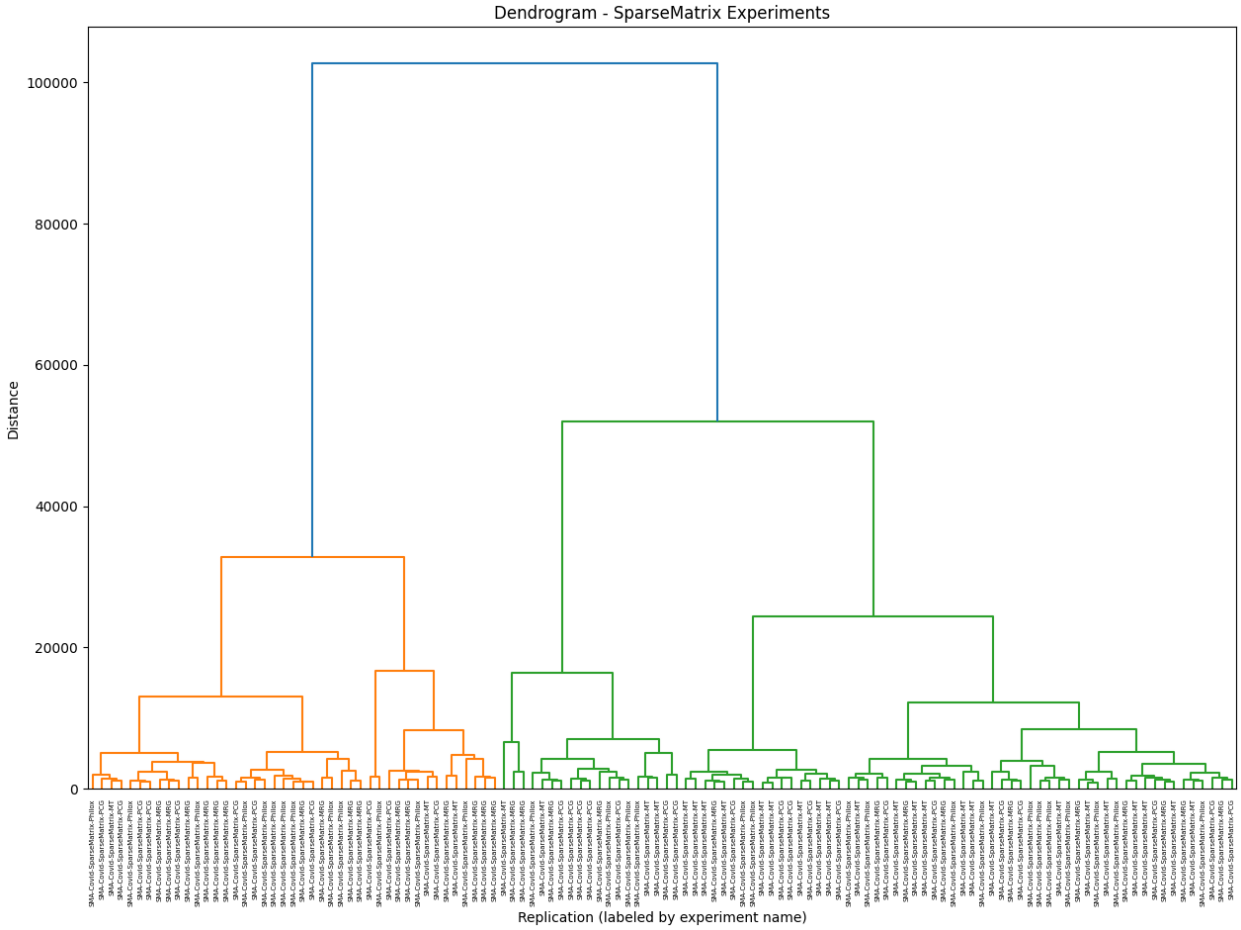


Figure 13: Dendrogramme des clusters générés pour les résulats de chaque simulation

IV.2.2.3. Test de performance matrice creuse et carte complète

Pour évaluer les performances temporelles et la consommation de RAM de notre simulation, nous avons analysé son exécution dans deux départements français distincts : Paris, connu pour sa forte densité de population, et la Gironde, reconnue comme le plus grand département de France métropolitaine. Selon les tables 3 et 4, l'utilisation de matrices creuse nécessite 3 à 4 fois plus de temps de traitement mais réduit considérablement l'utilisation de la RAM, d'environ 200 fois. Notre approche est quelque peu plus rapide que la moyenne des ABM utilisés dans la modélisation épidémiologique, qui incluent fréquemment des interfaces graphiques construites sur des technologies moins optimisées en termes de performance que le C++. Néanmoins, notre simulation est bien adaptée à une exécution sur des clusters de calcul, équilibrant la modélisation détaillée et l'efficacité des ressources.

	Carte complète	Matrice Creuse
Paris	55min	200min
Gironde	18min	69min

Table 3: Temps pris pour simuler 365 jours d'épidémie

	Carte complète	Matrice Creuse
Paris	0.9GB	0.003GB
Gironde	84GB	0.3GB

Table 4: Mémoire RAM utilisée (pic) pour simuler 365 jours d'épidémie

Pour mesurer la consommation d'énergie, nous utilisons PowerJoular (Noureddine 2022). La consommation d'énergie pour chaque simulation est d'environ 6000 Joules par minute pour les deux implémentations. Étant donné que l'utilisation de matrices creuses prend 3 à 4 fois plus de temps, elle consomme également 3 à 4 fois plus d'énergie au total. PowerJoular est basé sur RAPL (David *et al.* 2010). RAPL (Running Average Power Limit) fournit des mécanismes pour surveiller et limiter l'utilisation de l'énergie de divers composants au sein des processeurs Intel, y compris le CPU et la DRAM (pour la mémoire). Cependant, la conception de RAPL est davantage axée sur le contrôle et la mesure de la consommation d'énergie des composants matériels eux-mêmes plutôt que sur l'attribution directe de l'utilisation de l'énergie à des processus logiciels spécifiques. Mesurer la consommation d'énergie d'un processus spécifique est plus difficile. Étant donné que les processus partagent les ressources CPU et RAM de manière dynamique, l'attribution de la consommation d'énergie à des processus individuels nécessite de faire des approximations basées sur leur utilisation observée des ressources comme le temps CPU ou l'utilisation de la mémoire. PowerJoular utilise /proc et le PID du processus pour obtenir des informations sur la consommation d'énergie. Cependant, PowerJoular peut simplement mesurer la consommation d'énergie au niveau du CPU et non au niveau de la

mémoire. Comme les matrices creuses utilisent moins de mémoire, la différence entre les consommations d'énergie est discutable. Nous n'avons pas d'informations sur l'énergie consommée par la RAM pour nos processus.

Nous avons obtenu une reproductibilité bit à bit entre les implémentations de la carte complète et de la matrice creuse, permettant l'utilisation de l'une ou l'autre approche tout en obtenant exactement les mêmes résultats. Une telle répétabilité est cruciale pour une prise de décision éclairée lors de la gestion des épidémies.

Dans l'objectif de validation et pour présenter la portée de notre modèle, nous avons effectué diverses simulations dont nous avons extrait les résultats. Dans les résultats suivants, nous avons calibré le modèle pour qu'il fonctionne sur la métropole de Lyon : 533 km² et 1 411 571 habitants. L'objectif est de pouvoir observer différents comportements, en fonction des paramètres, afin de montrer comment le modèle peut être utilisé. Toutes les simulations ont été réalisées avec 30 réplications.

Nous avons opté pour simuler la pandémie de COVID-19 à travers cinq scénarios distincts. Le scénario initial reflète une situation où aucune mesure de protection n'est mise en œuvre. Le deuxième scénario introduit un confinement au début de l'épidémie. Le troisième scénario applique des masques et des désinfectants pour les mains pour atténuer la transmission. Dans le quatrième scénario, nous supposons que 60 % de la population reçoit un vaccin offrant 95 % de protection contre l'infection, reflétant les attentes initiales de la vaccination. Le scénario final sur base sur un taux de vaccination de 60 % avec un vaccin offrant 30 % de protection contre l'infection, réduisant également les admissions à l'hôpital et en soins intensifs. Chaque scénario a été répété 30 fois, avec des intervalles de confiance de 95 % calculés pour les résultats. La figure 14 présente les courbes de résultats d'une seule réplication, soulignant le potentiel des épidémies, reflétant les possibilités réelles. Ces résultats préliminaires servent à démontrer et à valider les capacités du modèle. Ce modèle suit les nouveaux cas quotidiens, les hospitalisations, les admissions en soins intensifs et les décès dans divers scénarios. L'axe des x indique le temps en jours, tandis que l'axe des y représente les quantités examinées. Notre modèle basé sur les individus fournit des informations cruciales pour l'analyse épidémiologique. Certaines observations clés de nos simulations, qui confirment l'exactitude du modèle, incluent :

- Sans aucune mesure de protection, l'épidémie entraîne une augmentation rapide des décès, avec une durée beaucoup plus courte par rapport aux scénarios avec interventions.
- Le confinement prévient efficacement la surcharge du système hospitalier en étendant la propagation de l'épidémie dans le temps, réduisant ainsi les décès causés par les dépassements de capacité des soins de santé. Cependant, les impacts secondaires négatifs, tels que les chirurgies annulées, les déficiences de soins dues à l'isolement et les effets psychologiques, doivent également être pris en compte.
- Un vaccin efficace à 95 % réduirait considérablement l'épidémie de COVID-19. En revanche, un vaccin avec une efficacité de 30 % reflète la progression réelle observée dans la pandémie en cours.

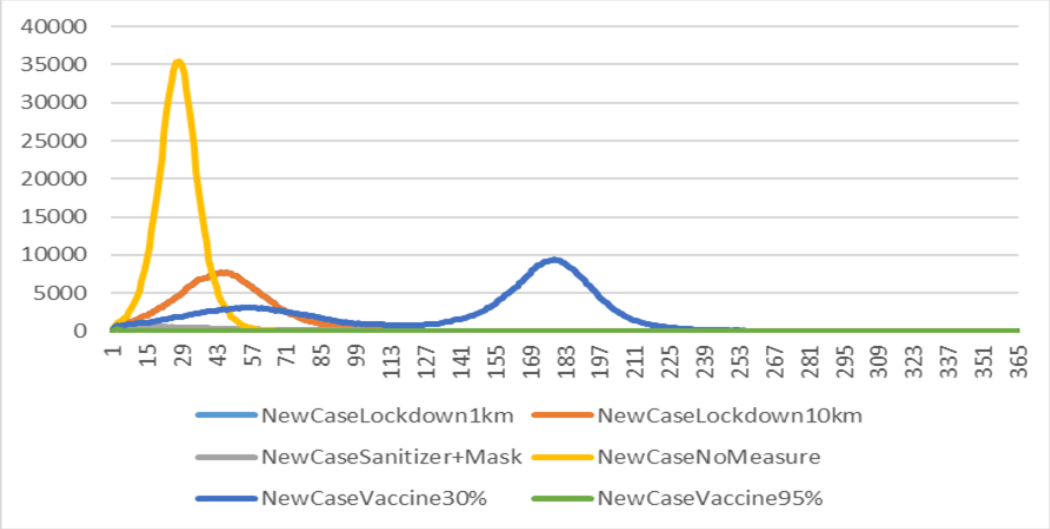


Figure 14: Exemple de résultat que le modèle peut générer, ici les nouveaux cas journaliers, suivant les mesures sanitaires en place pour contrôler l'épidémie

En tant qu'informaticiens, notre objectif n'est pas de disséquer ou de comprendre pleinement chaque mécanisme sous-jacent, mais de confirmer la fonctionnalité du modèle. Notre contribution est d'offrir un outil accessible aux experts pour simuler des scénarios détaillés, aidant les épidémiologistes et autres spécialistes à affiner leur compréhension. Notre modèle est conçu pour être convivial et peut être facilement utilisé sur des serveurs informatiques et des plateformes de calcul haute performance pour réaliser des conceptions expérimentales. Compte tenu de notre revue des normes actuelles, cela positionne notre logiciel comme une contribution significative, en particulier en termes de reproductibilité des expériences. Les résultats sont prometteurs ; nous avons réussi à évaluer les effets des interventions de santé, produisant des intervalles de confiance numériquement reproductibles.

IV.2.2.4. Analyse de sensibilité et plan d'expérience

Comprendre l'influence de chaque paramètre d'entrée sur les résultats est crucial pour évaluer l'utilisation et la fiabilité d'un modèle. Ce processus, connu sous le nom d'analyse de sensibilité, aide à identifier quels paramètres affectent de manière significative le comportement du modèle. Étant donné la complexité des modèles impliquant de nombreux paramètres et leurs combinaisons (combinaisons possiblement infinies), nous avons sélectionné six paramètres clés pour cette analyse : le taux de contamination, le taux d'asymptomatiques, la résistance à la réinfection, la résistance conférée par le vaccin, le taux de vaccination de la population, et le nombre de déplacements quotidiens. Ces paramètres ont été choisis pour représenter un large éventail d'influences sur la dynamique et les résultats des maladies.

Le taux de contamination est un facteur clé, déterminant le degré de contagiosité d'une maladie. Ensuite, pendant la pandémie de COVID-19, le rôle des individus asymptomatiques dans la propagation de la maladie a été souligné, faisant du taux d'asymptomatiques un paramètre critique. La résistance à la réinfection varie considérablement parmi les maladies ; certaines confèrent une immunité à long terme après l'infection, tandis que d'autres, comme la COVID-19, ne le font pas, en particulier en raison des mutations du virus. De même, la protection vaccinale, un sujet controversé pendant la crise de la COVID-19, reflète la capacité du vaccin à conférer une immunité contre la maladie. Le taux de vaccination est également un sujet important. Enfin, le nombre de déplacements quotidiens capture l'impact de la mobilité humaine sur la propagation des maladies, ce qui est un composant vital dans la modélisation épidémiologique.

Nous avons fait un choix sélectif des valeurs des paramètres, puis nous avons exécuté toutes les combinaisons possibles, menant à 1215 expériences (3*3*3*3*3*5). Les cinq premiers paramètres peuvent avoir 3 valeurs chacun, et le dernier paramètre peut en avoir 5. Le taux de contamination est un histogramme dont les valeurs varient en fonction du jour de l'infection, commençant bas, atteignant un pic en milieu d'infection, puis diminuant. Avec les 3 valeurs possibles de ces histogrammes, nous considérons des taux de contamination élevés, moyens et faibles. Les valeurs du taux d'asymptomatiques sont 0,2, 0,5 et 0,8. La résistance à la réinfection est un histogramme spécifique à l'âge. Les personnes plus jeunes ont une meilleure réponse immunitaire. Les 3 valeurs possibles de cet histogramme considèrent une réponse immunitaire élevée, moyenne et faible, allant de 0,999 à 0,1. Les valeurs de protection vaccinale sont également des histogrammes basés sur les groupes d'âge. Nous avons sélectionné 3 valeurs possibles pour une protection vaccinale forte, moyenne et faible. Le taux de

vaccination a été fixé dès le début de l'épidémie, avec comme valeur 0.1, 0.5 et 0.9 pour représenter des situations où le taux de vaccination est très faible, en comparaison avec un taux élevé. Le nombre de déplacements quotidiens a été fixé à 1, 5, 10, 20 et 30 pour modéliser différents niveaux de mobilité et d'interaction humaines.

Les résultats spécifiques mesurés incluent le nombre total de cas connus de COVID-19, capturant le nombre de cas symptomatiques ; le nombre total de décès pendant la période de simulation ; le taux de mortalité parmi la population infectée ; le taux de mortalité en considérant l'ensemble de la population initiale ; et la durée de l'épidémie, qui est le temps jusqu'à l'arrêt de l'épidémie.

Nous avons employé une méthode pour l'analyse de sensibilité : la méthode un-élément-à-la-fois (OAT, one at a time). La méthode OAT consiste à faire varier un paramètre à la fois tout en maintenant les autres fixés à leurs valeurs « moyennes », ce qui simplifie l'analyse et met en évidence l'effet individuel de chaque paramètre.

Pour la méthode OAT, nous avons fixé le taux de contamination à environ 0,6. C'est arbitraire car nous n'avons pas suffisamment de connaissances en sciences de la santé, mais c'est une simplification équitable. Nous avons fixé le taux d'asymptomatiques à 20 %, comme certaines données publiées pendant la COVID-19, en particulier lorsque certaines personnes vaccinées étaient supposées être plus asymptomatiques. Nous avons fixé la valeur par défaut de la résistance à la réinfection à environ 90 %, selon la catégorie d'âge de la personne. Nous supposons que lorsqu'une personne est infectée, il y a une très faible probabilité de réinfection. Cependant, l'immunité commence à diminuer après un certain temps. Nous avons fixé la valeur moyenne de la protection vaccinale à environ 60 %. Cette information n'était pas vraiment claire pendant la COVID, supposée initialement protéger à 95 %, et finalement seulement contre les cas graves. Nous avons fixé la valeur par défaut du taux de vaccination à 50%, certains pays européens pouvant monter à 90%, tandis que certains pays africains étaient plutôt autour de 10%. Le nombre de déplacements quotidiens, qui ne reflète pas vraiment une valeur réelle du monde, concerne le nombre de fois où les gens se déplacent, donc se rencontrent. Nous avons fixé cette valeur à 10. Dans la table 5, nous montrons les résultats de l'OAT.

Les résultats de l'analyse de sensibilité fournissent un aperçu de la manière dont chaque paramètre influence les résultats du modèle, mettant en évidence la complexité (et probablement l'interdépendance, qui ne sont pas montrées dans une méthode telle que OAT) des facteurs dans la dynamique des épidémies. Les variations du taux de contamination ont

significativement affecté le nombre total de cas de COVID-19 et la durée de l'épidémie. Par exemple, un taux de contamination plus élevé de 0,8 a conduit à plus de cas (2 066 597) mais à une durée d'épidémie plus courte (23 jours) comparé à un taux plus faible de 0,4, qui a entraîné beaucoup moins de cas (1108). En ce qui concerne le taux asymptomatique, nous pouvons voir qu'il n'affecte principalement que deux résultats, avec des taux plus élevés réduisant le taux de mortalité parmi la population infectée et prolongeant la durée de l'épidémie. N'oubliez pas que lorsque nous considérons le nombre total de cas de COVID-19, nous ne considérons que les cas symptomatiques, supposément parce que seules les personnes symptomatiques sont testées, tandis que les personnes asymptomatiques se déplacent simplement. Il existe plusieurs infections qui infectent effectivement plus de 50 % de la population totale, sans avoir d'impact sur leur vie, comme l'herpès ou le Toxoplasma Gondii.

Taux		**Nombre total de cas connus**	**Nombre de morts**	**Taux de mortalité cas connus**	**Taux de mortalité sur la population globale**	**Durée de l'épidémie en jours**
Contamination	0.4	1108	13	1.14	0.00	49
	0.6	1635098	22961	1.40	1.06	50
	0.8	2066597	11914	0.58	0.55	23
Asymptomatique	0.2	1635098	22961	1.40	1.06	50
	0.5	1692505	23431	0.45	1.08	49
	0.8	1910461	24192	0.37	1.18	61
Résistance Réinfection	0.3	23116128	84555	0.37	3.90	364
	0.6	13852336	23431	0.45	2.90	364
	0.9	1635098	22961	1.40	1.06	50
Résistance vaccin	0.2	1973630	22900	1.16	1.06	34
	0.6	1635098	22961	1.40	1.06	50
	0.8	1019398	16844	1.65	0.78	69
Taux vaccination	0.1	1961361	32469	1.66	1.50	34
	0.5	1635098	22961	1.40	1.06	50
	0.9	1147331	9770	0.85	0.45	91
Nombre de déplacement par jour	1	150	1	0.69	0.00	18
	5	818713	4636	0.57	0.21	362
	10	1635098	22961	1.40	1.06	50
	20	2046153	11863	0.58	0.55	29
	30	2125817	11421	0.54	0.53	17

Table 5: Résultat de l'analyse de sensibilité avec la méthode « One at a time »

La résistance à la réinfection a démontré un impact profond sur le nombre de cas et le taux de mortalité global. Des niveaux de résistance plus faibles (0,3) ont entraîné une augmentation spectaculaire des cas (23 116 128) et des décès (84 555), soulignant l'importance de l'immunité dans le contrôle des épidémies. Bien sûr, la plupart du temps, si vous avez un système immunitaire compétent, il y a une probabilité beaucoup plus faible d'être infecté à nouveau,

encore moins de mourir de la maladie après une réinfection. Fait intéressant, les variations du vaccin montrent un impact très faible, cependant, tandis qu'un vaccin plus fort réduit le nombre de personnes infectées connues, il augmente également le taux de mortalité des personnes infectées (tout en diminuant le taux de mortalité de la population globale). En ce qui concerne le pourcentage de personnes se faisant vacciner, nous pouvons voir que plus de personnes se faisant vacciner conduit à une épidémie moins intense, avec moins de contaminations et de décès, cependant, cela augmente la durée de l'épidémie. Le nombre de déplacements par jour a une influence marquée sur la dynamique de l'épidémie ; une mobilité plus élevée augmente le nombre de cas mais raccourcit la durée de l'épidémie, indiquant une propagation plus rapide et un pic plus rapide de l'épidémie.

Dans l'ensemble, ces résultats soulignent l'importance d'une estimation précise des paramètres dans la modélisation des épidémies et la nécessité d'interventions de santé publique ciblées qui prennent en compte ces facteurs critiques pour gérer et atténuer efficacement l'impact des maladies infectieuses. Ces expériences ne servent qu'à tester le modèle et ne fournissent aucune hypothèse sur la manière de gérer une épidémie. Dans les figures suivantes des courbes de nouveaux cas, nous constatons que nous pouvons obtenir des résultats intéressants.

Des résultats classiques d'une épidémie très faible sont montrés dans la figure 15, avec un très faible nombre de personnes contaminées, tandis que l'épidémie (si l’on peut toujours parler d’épidémie avec si peu de cas) dure un peu plus longtemps. Un autre cas classique est lorsque l'épidémie est très forte, comme montré dans la figure 16. Il y a rapidement un énorme pic, et l'épidémie commence à diminuer rapidement, comme un incendie de forêt qui a brûlé toutes les ressources disponibles.

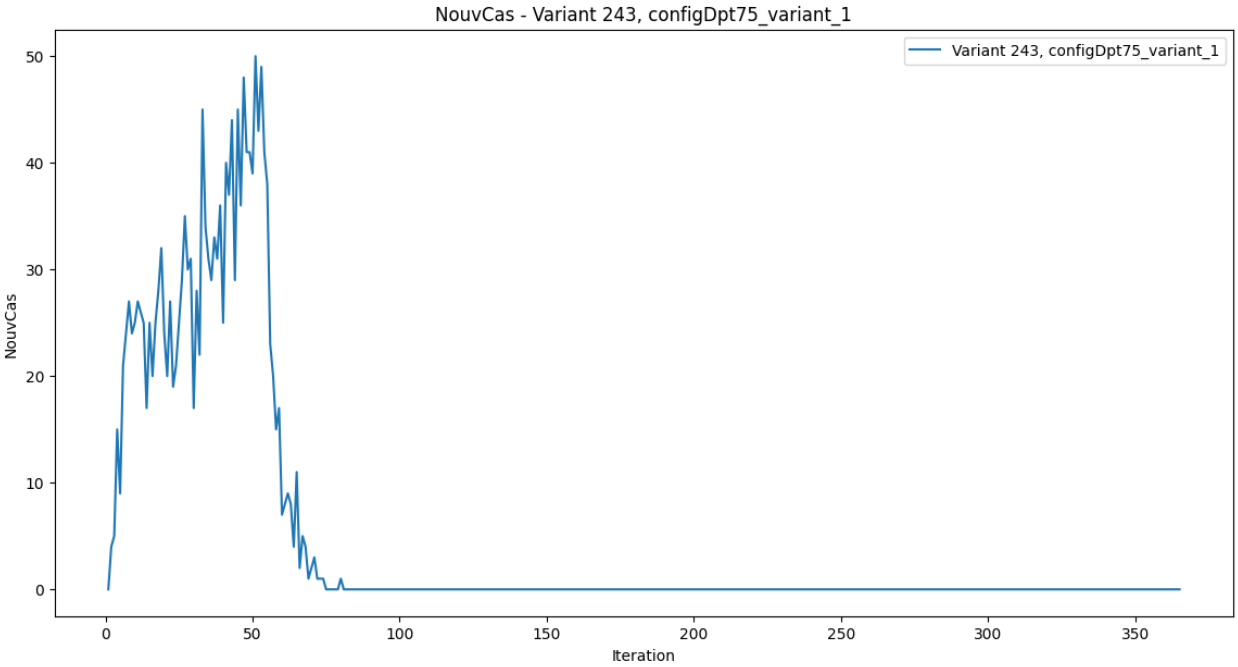


Figure 15 : Exemple de courbe avec une très faible contamination, mais une durée d'environ 80 jours

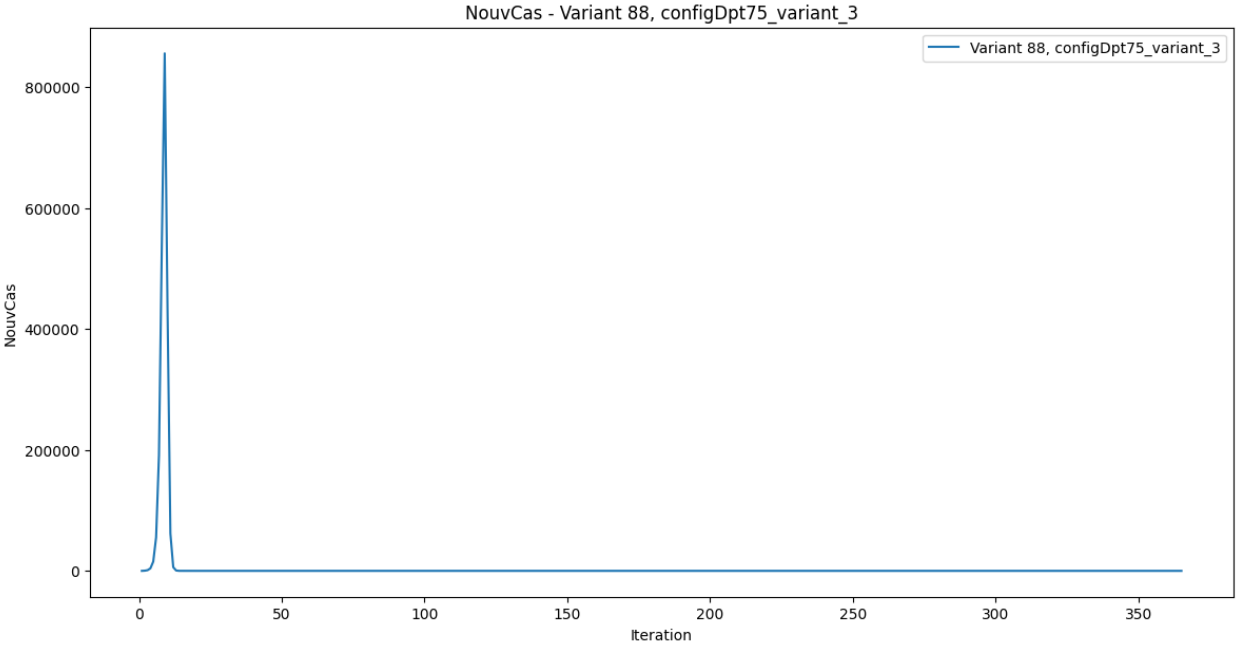


Figure 16 : Exemple d'une épidémie extrémement virulante, avec contamination totale en seulement 15 jours

Nous pouvons également voir des résultats assez intéressants dans les figures 17 et 18, où nous pouvons voir des courbes plus « réalistes ». Parfois, comme dans la figure 17, la courbe ressemble exactement à ce que nous aurions pu obtenir avec une modélisation par équations différentielles ordinaires (ODE).

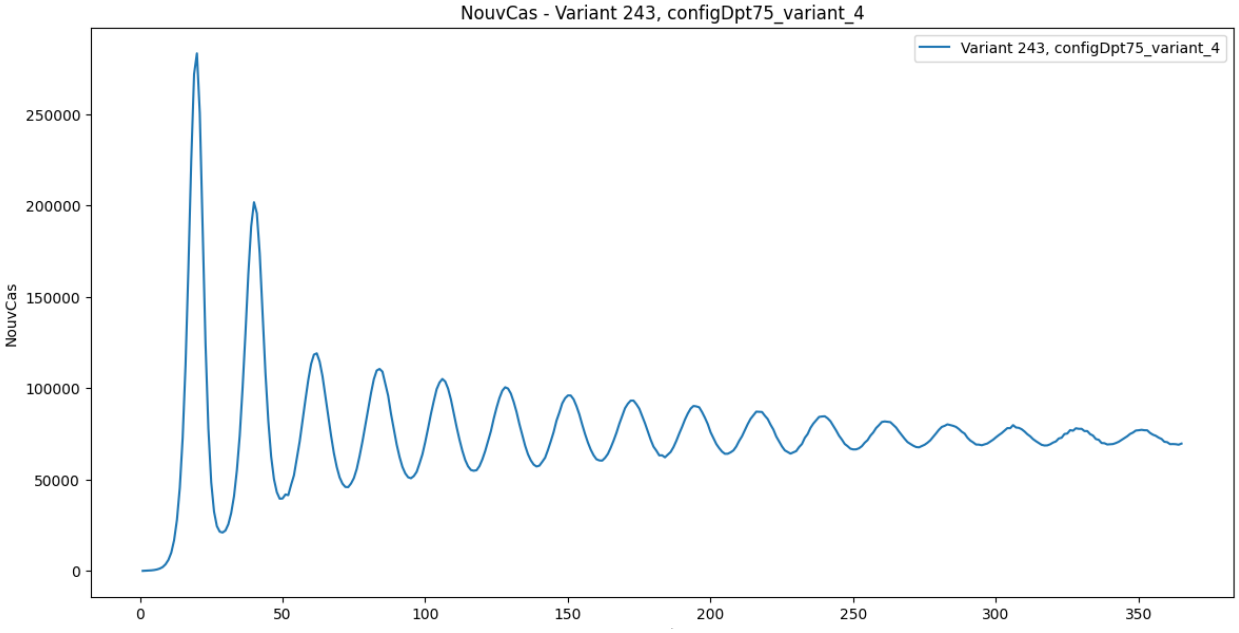


Figure 17 : Exemple d'une épidémie avec un comportement étonnant, proche d'une modélisation avec ODE

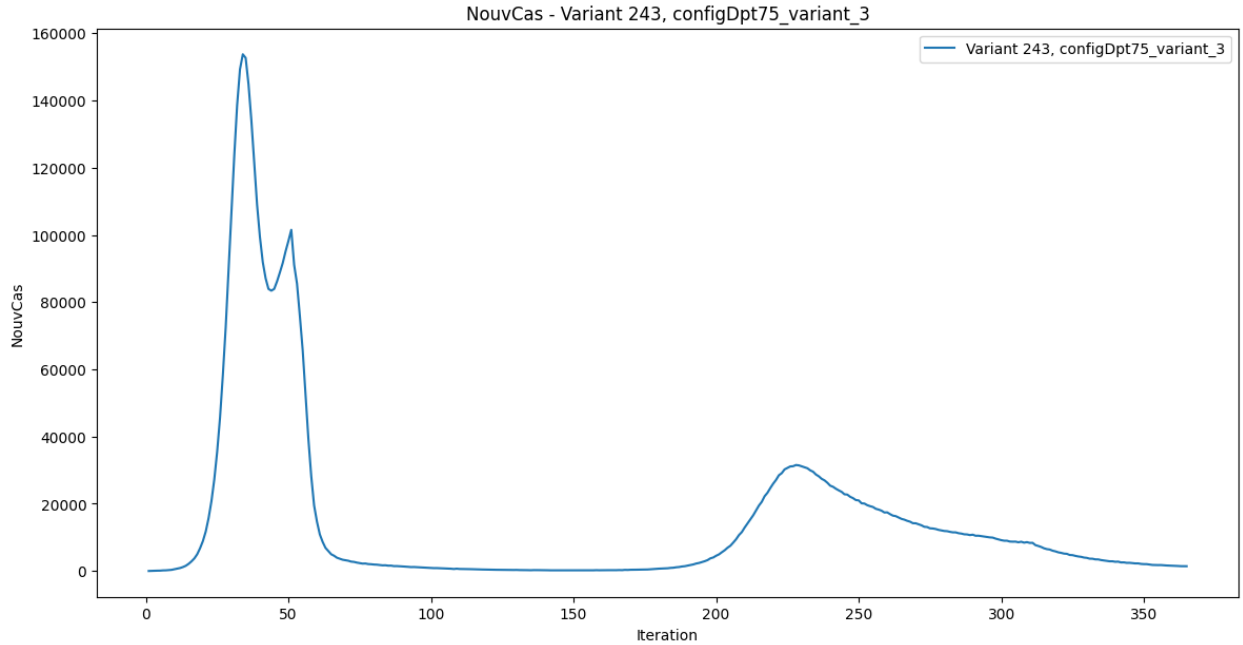


Figure 18: Exemple plus réaliste d'une épidémie moyenne, avec rebond

IV.2.3. Discussion

Les sociétés ne restent pas sans réaction face aux épidémies, et encore moins face aux pandémies. Pour interpréter avec précision les dynamiques que nous observons, il est impératif que les modèles permettent d'examiner diverses interventions et politiques de santé publique. Ceux-ci incluent les obligations de port du masque, l'utilisation de désinfectants pour les mains, la distanciation sociale, les protocoles de dépistage, les quarantaines individuelles, les couvre-feux, les confinements, les restrictions de mouvement, les traitements potentiels et le déploiement de vaccins expérimentaux ou d'améliorateurs d'immunité dans des scénarios où les traitements ne sont pas disponibles. Nous avons développé un modèle de base qui utilise des agents réactifs et spatialisés, doté de capacités multi-échelles. Ce modèle est non seulement capable de simuler les dynamiques de transmission locale entre individus, mais peut également évoluer pour représenter les interactions au niveau de la ville, du département ou du pays. Nos évaluations préliminaires indiquent que cet outil peut être utile pour améliorer notre compréhension de la progression nuancée des pandémies, comme celle que nous avons affrontée en 2020-2021. Il permet notamment d'évaluer les diverses stratégies de santé publique et leurs impacts. Le modèle est conçu pour le calcul parallèle, facilitant des calculs rapides de réplication, d'intervalles de confiance et de conceptions expérimentales reproductibles, conformément aux meilleures pratiques de parallélisation de l'algorithme Mersenne Twister. Dans notre engagement pour la reproductibilité, nous fournissons le code source, un exemple démonstratif et des instructions d'utilisation simples, en soulignant l'importance de reproduire les résultats numériques. Étant donné la nature stochastique de ce modèle, sa répétabilité est assurée grâce à l'utilisation d'un générateur de nombres pseudo-aléatoires de haute qualité, ici Mersenne Twister. Nous avons réussi à convertir notre modèle pour utiliser des matrices creuses, ce qui nous permet d'utiliser beaucoup moins de RAM tout en conservant les mêmes résultats numériques. En ce qui concerne la consommation d'énergie, elle dépend principalement du temps de calcul, donc les matrices creuses consomment plus. Diverses expériences ont été testées sur différentes configurations matérielles et logicielles, et nous confirmons la répétabilité des résultats sur ces machines. La dernière décennie a vu un intérêt croissant pour la reproductibilité au sein de la communauté scientifique, et notre travail contribue au discours en cours sur le maintien de la rigueur dans la recherche basée sur la simulation.

IV.3. Parallélisation des PRNGs : Quelles méthodes choisir ?

IV.3.1. Expérience mise en place

Afin de déterminer la qualité des états (à partir desquels on génère une séquence déterministe de nombres pseudo-aléatoires) de Mersenne Twister en fonction de la technique de parallélisation, et pour mettre à l'épreuve la reproductibilité de TestU01, nous avons sélectionné un ensemble de configurations matérielles et un ensemble de méthodes de parallélisation. Comme dit précédemment, nous savons depuis longtemps que la qualité des flux de nombres pseudo-aléatoires est un facteur déterminant pour la recherche utilisant des simulations stochastiques parallèles (Lazaro *et al.* 2005; Click *et al.* 2011). Voici le matériel dont nous disposons pour les tests : nous avons 3 configurations différentes, A, B et C. Dans la configuration A, nous avons un cluster via Slurm avec 6 nœuds fonctionnant sous Linux (Ubuntu) avec 2 AMD EPYC 7452 avec 32 cœurs physiques et 512 Go de RAM. Chaque nœud a donc 64 cœurs physiques. Pour la configuration B, nous avons utilisé avec accès direct une machine avec 2 AMD EPYC 7763 avec 64 cœurs physiques et 512 Go de RAM. Cette machine a 128 cœurs physiques. Enfin, pour la configuration C, nous avons utilisé une machine avec 16 processeurs Intel(R) Xeon(R) CPU E7-8890 v4 2,20 GHz (Broadwell), soit 384 cœurs physiques (et 12 To de RAM). Le compilateur utilisé sur les trois machines est gcc. La version de gcc est différente sur chaque machine que nous avons utilisée, respectivement 9.4.0, 11.3.0 et 4.8.4. Cela a entraîné de petites différences dans le binaire exécutable vérifié avec la commande « diff » de Linux. Nous étudions le PRNG Mersenne Twister original (sa version 32 bits) car TestU01 est conçu en 32 bits. En plus du fait que MT est classé comme l'un des meilleurs générateurs actuels, il est de loin le générateur moderne le plus connu et le plus répandu. Même si certains défauts sont connus, il a été utilisé pendant une longue période sans qu'aucune défaillance observée n'ait été signalée dans des applications réelles. De plus, contrairement à ce qui est dit sur Wikipédia, le MT original est déjà très rapide, en particulier lorsqu'il est compilé avec des options d'optimisation (le SFMT est deux fois plus rapide mais moins connu et utilisé). Le fait qu'il soit très répandu parmi la communauté scientifique et les bibliothèques standard est peut-être l'argument principal de notre choix. Même si nous avons des éléments montrant que MT n'est pas dépendant de la « graine », l'expérience en médecine nucléaire a montré que nous devons vérifier la qualité des états d'initialisation. En effet, l'état réel de MT n'est pas une petite graine mais un tableau de 624 valeurs plus un index. Nous avons généré 4096 états MT avec 3 méthodes différentes : le sequence splitting, le random spacing et la méthode spécifique à MT qui produit un état complet de 2,5 Ko à partir d'un entier non signé

long (fonction init_genrand()). La taille de 10^{12} d'espacement a été retenue pour l'approche de sequence splitting ; avec les 4096 états, elle est adaptée aux grands clusters ou aux petites partitions de superordinateurs. Ces états et le code associé pour les produire sont disponibles sur notre Gitlab. Pour le sequence splitting, nous avons démarré MT avec l'initialisation par défaut et nous avons séquentiellement enregistré les états après 10^{12} tirages. Cette méthode prend d'abord beaucoup de temps car nous devons générer espaceEntreEtats * nombreDEtats nombres. Dans notre cas, nous avons dû générer 10^{12} * 4096 nombres (~4 * 10^{15} nombres). Pour le random spacing, nous utilisons la fonction genrand_int32() de MT pour remplir les éléments d'un état de MT (remplir le tableau unsigned long mt[] et donner un index entier mti approprié). Nous pouvons générer autant d'états que nous le souhaitons, et cette méthode est très rapide pour produire des états. La probabilité de chevauchement est proche de 0 avec la période énorme de MT (2^{19937}). Pour la séquence indexée de MT, nous utilisons simplement la fonction init_genrand() qui accepte un entier non signé long et génère un état. Réaliser cela dans une boucle de 0 à 4095 produit l'ensemble des 4096 états. Matsumoto et son équipe ont soigneusement implémenté cette fonction pour éviter les mauvaises initialisations (Matsumoto *et al.* 2007). Cette méthode est également très rapide comparée à la première. Le seul avantage de l'approche de sequence splitting est que les flux ne se chevauchent pas et que lorsque les états sont déjà calculés, nous pouvons les réutiliser directement (comme pour les autres méthodes d'ailleurs). Sur les trois types de machines différents, nous avons mis les 4096 états MT correspondant à chaque méthode. Une commande « diff –r » de Linux nous assure que tous les états sont les mêmes pour chaque méthode sur chaque machine. Cette vérification des fichiers d'état MT élimine les problèmes de reproductibilité pouvant provenir de cette partie de l'expérience. Avant d'utiliser les états générés en parallèle, nous devons tester leur qualité. Le code TestU01 provient du site web de L'Ecuyer, et nous utilisons la version 1.2.3. La batterie de tests Big Crush se compose de 106 tests et chaque test est considéré comme réussi si sa p-value se situe entre 10^{-10} et 1 - 10^{-10}. Tous les tests ont été exécutés en parallèle avec les machines dont nous disposons. La batterie de tests Big Crush est relativement intensive en calcul. Elle nécessite environ 4 heures de temps sur des processeurs modernes pour le test des états. Nous avons 4096 tests parallèles avec 3 techniques différentes de génération d'états sur 3 machines différentes, avec un test statistique pour les nombres aléatoires entiers et réels. Cela correspond à 73 728 exécutions de 4 heures. Au total, 294 912 heures, soit plus de 33 années CPU de calcul ont été effectuées en deux mois de calcul sur nos serveurs et clusters. Le temps nécessaire pour configurer l'expérience est variable en fonction de la technique utilisée. La génération d'états MT avec des techniques aléatoires et des techniques d'index MT est

négligeable (700 millisecondes pour 4096 états, y compris l'écriture des fichiers d'état), tandis que la génération avec des techniques de sequence splitting peut prendre plusieurs jours en fonction de la quantité d'états ; 1 heure pour chaque état espacé de 10^{12} nombres sur des processeurs modernes.

IV.3.2. Résultats

Lors de la parallélisation d'une application stochastique, chaque flux de nombres pseudo-aléatoires donné à un élément de traitement doit être de bonne qualité, et ne pas se recouvrir avec les autres flux. La bonne nouvelle est que nous avons une parfaite répétabilité sur les 3 configurations. D'un autre côté, bien que nous soyons conscients que MT présente quelques défauts connus, nous avons été surpris par nos résultats.

Configuration matérielle	**A (AMD)**	**B (AMD)**	**C (Intel)**
GCC version	9.4.0	11.3.0	4.8.4
Random spacing (int)	1185 / 4096 28.93%	1185 / 4096 28.93%	1185 / 4096 28.93%
Random spacing (double)	1185 / 4096 28.93%	1185 / 4096 28.93%	1185 / 4096 28.93%
Sequence Splitting (int)	1156 / 4096 28.22%	1156 / 4096 28.22%	1156 / 4096 28.22%
Sequence Splitting (double)	1156 / 4096 28.22%	1156 / 4096 28.22%	1156 / 4096 28.22%
MT Index Seq. (int)	1139 / 4096 27.8%	1139 / 4096 27.8%	1139 / 4096 27.8%
MT Index Seq. (double)	1128 / 4096 27.53%	1128 / 4096 27.53%	1128 / 4096 27.53%

Table 6: Nombres d'états ayant échoué à plus que les deux tests LinearComp dans les 3 configurations différentes avec 3 techniques de parallélisation différentes

Dans la Table 6, nous observons qu'environ 30 % des états de MT générés avec les différentes techniques produisent des flux aléatoires qui échouent à plus de 2 tests, alors qu'ils étaient censés échouer n'échouer qu'a deux tests (tests pour les applications cryptographiques,

80 et 81 LinearComp). La première colonne donne les noms de la technique de parallélisation (test avec des nombres entiers ou réels), les 3 colonnes suivantes donnent les résultats obtenus dans les 3 configurations que nous avons testées (A, B et C). Une remarque pour la Table 6 : en comparant les flux de nombres entiers et réels, il apparaît que nous avons la même statistique concernant les états ratant plus que les deux tests de LinearComp, mais en réalité les flux n'échouent pas aux mêmes tests, ce qui sera discuté plus tard (même si nous avons le même nombre d'échecs).

Deuxièmement, nous remarquons que la méthode de parallélisation utilisée n'a pas d'impact clair sur la qualité des flux aléatoires. Les petites différences que nous pouvons voir ne sont pas statistiquement significatives, même si nous pouvons maintenant donner un petit classement en faveur de la fonction d'indexation spécifique développée spécialement pour MT pour produire un état complet. Même si nous pouvons obtenir 70 % de flux aléatoires parallèles de très bonne qualité, nous devons approfondir la question pour les 30 % restants d'échecs. Une première suggestion est que les chercheurs travaillant avec MT et ayant besoin d'utiliser la parallélisation des simulations stochastiques devraient éviter la technique de sequence splitting pour générer des états, sauf si les états sont déjà calculés et disponibles. En effet, cette approche nécessite un long pré-calcul séquentiel avec le déroulement de la séquence de MT et la sauvegarde des états à une distance fixe. Le test a été réalisé avec des séquences non chevauchantes espacées de 10^{12} nombres aléatoires. Un autre point intéressant montré par ce tableau est une bonne nouvelle pour la science numérique : TestU01 est bien répétable et reproductible sur différentes plateformes dans notre expérience. En fait, pour chaque technique de parallélisation, nous avons obtenu une reproductibilité numérique sur les trois configurations matérielles et logicielles différentes (avec des processeurs différents et même des fichiers exécutables différents, chaque configuration étant livrée avec une version différente de gcc). Dans la Table 7, nous pouvons voir que chaque flux parallèle qui a échoué à plus de 2 marqueurs (tests Linear Comp), n'échoue pas à plus de 6 tests (sur 106). Cela signifie que même si nous avons un grand nombre de défauts, la qualité des flux reste bonne avec plus de 94 % de réussite sur 106 tests. Le point principal est que la méthode de sequence splitting nécessite un temps de préparation énorme sans obtenir de meilleurs résultats que l'espacement aléatoire ou la séquence d'index MT.

Nom de la technique	Nombre de tests ratés	Fréquence correspondante
MT Index Sequence (entier)	3	964
	4	156
	5	18
	6	1
MT Index Sequence (réel)	3	951
	4	158
	5	19
	6	
Sequence splitting (entier)	3	971
	4	164
	5	21
	6	
Sequence splitting (réel)	3	971
	4	161
	5	24
	6	
Random spacing (entier)	3	990
	4	172
	5	21
	6	2
Random spacing (réel)	3	987
	4	175
	5	21
	6	2

Table 7: Nombre de tests échoués et fréquence des échecs sur les 3 configurations différentes

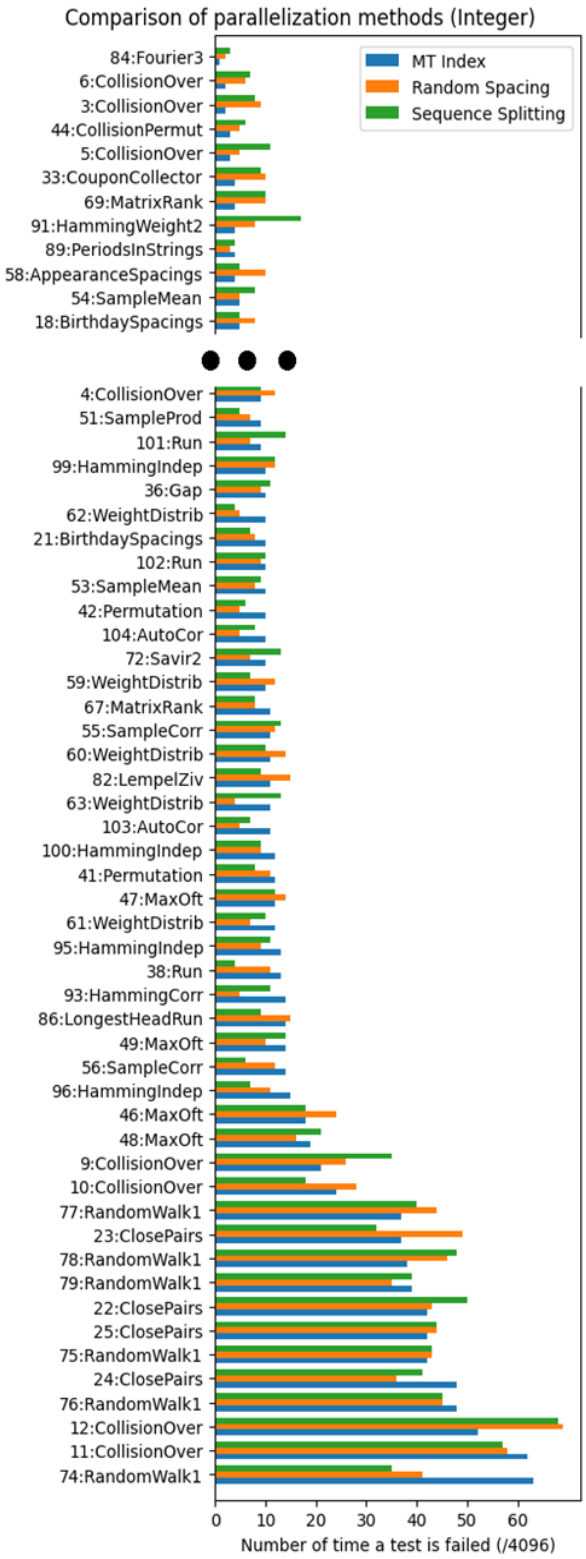


Figure 19: Fréquence des échecs pour les tests de la batterie BigCrush pour les entiers 32 bits

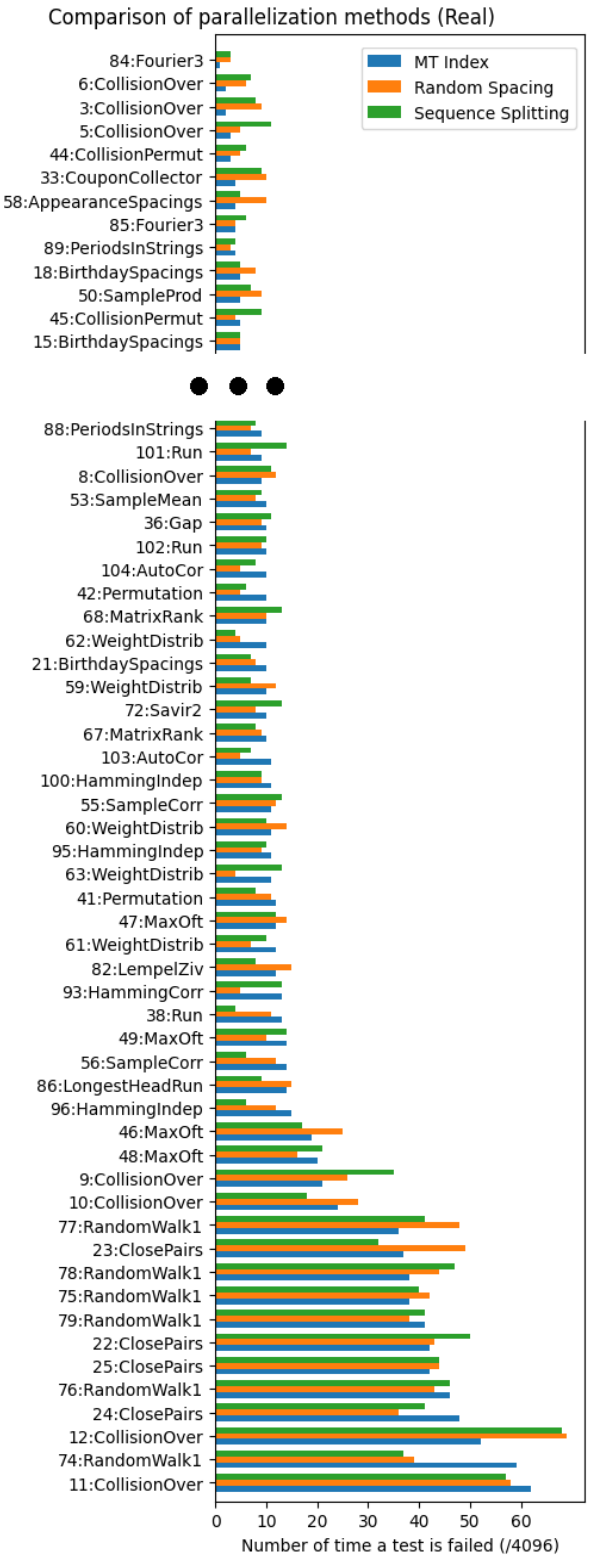


Figure 20: Fréquence des échecs pour les tests de la batterie BigCrush pour les doubles 64 bits

Nous voyons dans la Table 7 que même si les flux aléatoires de nombres entiers et réels ont le même nombre d'échecs, dans le détail, ils n'échouent pas aux mêmes tests. Nous ne pouvons pas dire que nous avons une meilleure performance avec l'un ou l'autre, la qualité des flux de nombres aléatoires entiers et réels reste assez équivalente.

Les figures 19 et 20 donnent les résultats ordonnés par la fréquence des échecs pour chaque test : tous les 106 tests sont concernés ! Dans ces figures, nous tronquons la liste des tests pour qu'elle tienne sur une page ; les données complètes sont disponibles avec les artefacts. Les trois méthodes de parallélisation sont comparées pour chaque test, et nous avons des résultats assez similaires. Nous pouvons également remarquer que certains tests sont ratés plus fréquemment par MT.

Numéro et nom des tests	**MT Index Sequence % tests ratés (sur 4096)**	**Random Spacing % tests ratés (sur 4096)**	**Sequence Splitting % tests ratés (sur 4096)**
11:CollisionOver	1.51%	1.42%	1.39%
74:RandomWalk1	1.44%	0.95%	0.9%
12:CollisionOver	1.27%	1.68%	1.66%
24:ClosePairs	1.17%	0.88%	1.0%
76:RandomWalk1	1.12%	1.05%	1.12%
25:ClosePairs	1.03%	1.07%	1.07%
22:ClosePairs	1.03%	1.05%	1.22%
79:RandomWalk1	1.0%	0.93%	1.0%
75:RandomWalk1	0.93%	1.03%	0.98%
78:RandomWalk1	0.93%	1.07%	1.15%
23:ClosePairs	0.9%	1.2%	0.78%
77:RandomWalk1	0.88%	1.17%	1.0%

Tableau 8: Pourcentage d'échecs de test de la batterie BigCrush avec des nombres réels en fonction de la technique de parallélisation

Nous avons sélectionné certains de ces tests et les listons dans ta Table 8. Ce tableau ne donne que les tests avec le plus grand pourcentage d'échecs (environ 1 %) pour les trois techniques de parallélisation. Lorsque le nom du test est le même et que le numéro du test

diffère, il s'agit en fait du même test statistique appliqué avec des paramètres différents, comme expliqué dans le guide utilisateur de TestU01.

Comme nous pouvons le voir, les résultats sont assez similaires pour chaque méthode de génération. Cependant, le sequence splitting est la seule qui prend beaucoup plus de temps pour générer les états puisqu'elle nécessite une phase de pré-calcul. L'espacement aléatoire et la séquence d'index sont beaucoup plus rapides pour générer ces états. En effet, le sequence splitting doit générer tous les nombres (10^{12} x nombre d'états), tandis que les deux autres ont simplement besoin de générer 624 valeurs multipliées par le nombre d'états. Entre les deux techniques précédentes, nous pouvons voir un résultat légèrement meilleur pour la méthode proposée par Matsumoto et Nishimura. À partir de ces faits, nous recommanderions d'utiliser la technique de séquence d'index pour obtenir des séquences parallèles « indépendantes » de nombres. Cependant, le sequence splitting est la seule méthode garantissant l'absence de chevauchement si nous respectons la taille des séquences générées.

Les limites de cette étude proviennent du fait que nous ne pouvons pas tester une énorme quantité d'états de MT en raison de la puissance de calcul nécessaire pour effectuer tous les tests. Une autre limitation est qu'une version 64 bits de TestU01 n'est pas disponible.

IV.3.3. Discussion

Dans les applications réelles où nous avons besoin de nombres aléatoires parallèles, la qualité de la technique de parallélisation peut avoir un impact significatif sur les résultats (Maigne *et al.* 2004; Lazzaro *et al.* 2005; Hill *et al.* 2013). Nous avons étudié la qualité des flux de Mersenne Twister avec 3 techniques de parallélisation du même générateur : espacement aléatoire, séquence indexée MT et sequence splitting. Nous avons réalisé les tests sur trois plateformes différentes à des fins de reproductibilité et nous avons constaté que nous ne pouvions pas privilégier une technique parmi les autres. Les résultats montrent des différences très légères en faveur de la technique spécifique à MT. Étant donné que le sequence splitting prend beaucoup plus de temps de préparation que les deux autres avec le calcul séquentiel des états initiaux avant de les utiliser en parallèle, nous suggérons d'éviter son utilisation, sauf si vous disposez déjà de bons états initiaux (lorsque le flux correspondant a été testé). La répétabilité de nos expériences sur TestU01 avec différents matériels et logiciels a montré que les résultats produits sont robustes. Nos résultats montrent que 30 % des flux de nombres aléatoires de MT échouent à plus des deux tests de vérification attendus (80 et 81 LinearComp). Étant donné que des flux de haute qualité sont indispensables pour des simulations

stochastiques parallèles fiables, nous fournissons des informations et du code pour générer rapidement des états de qualité pour MT ainsi que tous les états correspondant à des flux de haute qualité.

IV.4. Reproductibilité des PRNGs dans les technologies du machine learning

IV.4.1. L'expérience

Pour répondre aux questions soulevées dans la section propositions concernant l'utilisation des PRNGs en machine learning, nous avons sélectionné des frameworks de machine learning de premier plan, en particulier PyTorch et TensorFlow, ainsi que Python et la bibliothèque NumPy en raison de leur utilisation répandue dans le domaine du ML. À des fins de comparaison, nous avons retenu les implémentations de code C originales de Mersenne Twister, PCG, Philox et Mrg32k3a comme norme de référence (tous les codes sont proposés sur les pages web des auteurs). Cette étude a été publié dans l'article (Antunes et Hill 2024).

Mersenne Twister prend en charge la génération native à la fois d'entiers 32 bits et de doubles 64 bits. En revanche, Mrg32k3a est limité à la génération de doubles 64 bits uniquement. Afin de rester fidèles aux implémentations originales, nous avons limité notre utilisation de Mrg32k3a aux expériences impliquant des doubles 64 bits. Inversement, l'algorithme Philox était uniquement disponible pour la génération d'entiers 32 bits de ses auteurs. PCG offre la possibilité pour les deux, mais l'auteur préfère s'en tenir aux entiers : « Comme les bibliothèques Unix rand et random, cette bibliothèque ne fournit pas de fonctionnalités directes pour générer des nombres aléatoires en virgule flottante. Il s'avère que générer des valeurs en virgule flottante aléatoires est étonnamment difficile. » (https://www.pcg-random.org/using-pcg-c-basic.html). Cependant, comme l'auteur fournit une solution pour générer des doubles, nous avons utilisé PCG dans les deux cas, comme MT. Les frameworks de ML, avec leurs API avancées, permettent de générer facilement des entiers 32 bits ou des doubles 64 bits. La version la plus récente de TensorFlow suggère d'utiliser un objet Generator, que nous avons explicitement appliqué à l'algorithme Philox. Pour PyTorch, bien que l'algorithme sous-jacent soit censé être basé sur Philox selon la documentation, l'utilisateur ne peut pas spécifier son choix de générateur. NumPy se distingue peut-être comme la bibliothèque la plus polyvalente pour la gestion de divers PRNGs, offrant une documentation claire et une gamme d'algorithmes disponibles. Avec NumPy, nous avons utilisé l'objet Generator, en le configurant pour utiliser explicitement Mersenne Twister, Philox et PCG.

Ces technologies diffèrent des pratiques de calcul scientifique traditionnelles en C, C++ ou Fortran, où les nombres aléatoires sont généralement générés individuellement selon les besoins. Les frameworks du ML, au contraire, sont optimisés pour générer des nombres aléatoires en bloc dans le cadre d'objets tensoriels (similaires à des matrices). Par conséquent, nous avons mené des expériences de deux manières : en générant des nombres un par un et en bloc (chunk). Pour Python, l'approche la plus efficace était de générer des nombres individuellement.

Comme PCG propose différentes versions, pour les nombres 64 bits nous avons choisi exactement la même version que celle qu'utilise NumPy (PCG 128/64 XSL-RR) et pour les nombres 32 bits nous avons utilisé PCG 64/32 XSH-RR. Nous avons initialisé tous les PRNGs avec la même valeur de graine. Pour neutraliser les disparités de type de données spécifiques au langage, nous avons utilisé la valeur de graine '0', garantissant un pointeur de mémoire de graine rempli de zéros pour différents types de données. Bien que l'initialisation avec zéro puisse être problématique pour certains PRNGs (Saito et Matsumoto 2006), cela a été fait intentionnellement pour observer le comportement résultant. Il est impératif pour les chercheurs de la communauté scientifique de reconnaître qu'une graine et l'état complet d'un PRNG sont des entités distinctes. L'état du PRNG détermine la valeur de sortie qu'il génère. En revanche, utiliser une graine implique l'application d'une fonction spécifique pour convertir la graine en l'état complet du PRNG. Il est notable que ce processus de transformation peut varier selon les différentes plateformes technologiques. Étant donné que l'ensemble des frameworks de machine learning utilisent une fonction d'initialisation via une graine, notre étude se concentre principalement sur cet aspect.

Notre évaluation a utilisé divers scripts Bash : un pour exécuter les évaluations de temps de calcul et de consommation d'énergie - générant 2^{30} nombres un par un ou en une seule fois et chronométrant le processus avec la commande « time ». La consommation d'énergie a été surveillée sur une période donnée (par exemple, 30 secondes), avec des résultats extrapolés sur toute la durée. Nous avons répété ces mesures 30 fois pour renforcer la validité statistique de nos mesures, ce qui conduit à l'étude d'échantillons de peu moins de 2^{35} nombres, pour chaque technologies et PRNGs étudiés.

Les mesures de consommation d'énergie ont été obtenues à l'aide de PowerJoular (Noureddine 2022). Cet outil offre la possibilité de mesurer la consommation d'énergie d'un process ID (PID) donné, en utilisant la fonctionnalité RAPL d'Intel (David *et al.* 2010), également disponible sur les puces AMD récentes. Nous avons compilé tous les codes C avec

différents niveaux d'optimisation (aucun, -O2 et -O3) pour discerner l'impact des optimisations du compilateur sur le temps et l'efficacité énergétique.

Pour l'évaluation de la qualité statistique des nombres aléatoires générés, nous avons exécuté un autre ensemble de scripts Bash. La batterie de tests BigCrush de TestU01, qui nécessite typiquement un peu plus de 2^{38} nombres selon la documentation de TestU01, nous avons généré 2^{39} nombres (un ordre de grandeur en plus). Étant donné que BigCrush n'est pas conçu pour lire des nombres à partir d'un fichier (avec BigCrush) dans sa forme originale, nous avons créé une interface en code C. Nous avons stocké les nombres générés par les frameworks du machine learning en Python dans un fichier binaire et ensuite, le programme C lit les nombres séquentiellement à partir de ce fichier pour fournir les entrées requises par BigCrush. Cette méthode a également été appliquée aux PRNGs codés en C pour une comparaison équitable. Les tests préliminaires n'ont montré aucune différence entre l'approche modifiée et l'originale, confirmant la validité de notre méthode. Cependant, il est important de noter que le stockage de 2^{39} doubles nécessite 4,4 To de stockage et 2,2 To pour les entiers 32 bits, ce qui est une contrainte non négligeable.

Enfin, pour la reproductibilité numérique, nous avons généré 100 nombres pseudo-aléatoires dans un fichier lisible et utilisé la commande « diff » sur les fichiers, l'algorithme étant le même, avec la même graine, nous attendons des résultats identiques au bit près.

Nous avons utilisé un Jupyter Notebook pour analyser tous les résultats et exécuter tous les scripts Bash afin de reproduire facilement les expériences. Tous les codes et données sont disponibles sur : https://gitlab.isima.fr/beantunes/random-numbers-in-machine-learning/.

Les expériences ont été réalisées sur une machine équipée de deux processeurs AMD 7763 à 64 cœurs, conduisant à 128 cœurs physiques et 256 cœurs logiques. La machine dispose de 512 Go de RAM et de 7,7 To de stockage NVMe. Nous avions un accès root, ce qui nous a permis d'effectuer des mesures de consommation d'énergie (RAPL nécessite un accès root pour être utilisé). La version de Python utilisée est la 3.11.5. La version de GCC utilisée est la 13.2.0. Le système d'exploitation est un Linux, Debian 6.4.13-1.

IV.4.2. La reproductibilité des temps de calcul

Les tables 9 et 10 illustrent le temps nécessaire pour générer 2^{30} nombres dans chaque expérience. Tout d'abord, des différences de performance distinctes entre les entiers 32 bits et les doubles 64 bits sont observées. Notamment, l'algorithme PCG démontre une vitesse

supérieure pour les entiers 32 bits mais nécessite de quadrupler son temps de génération pour les doubles 64 bits.

Générateur	Temps réel (s)	Temps réel 95% IC	Temps User (s)	Temps User 95% IC
pcg32Integer	2,45	[2,27; 2,64]	2,45	[2,27; 2,63]
numpyIntegerTasksetAtOnce	2,60	[2,59; 2,60]	2,20	[2,19; 2,22]
tensorflowIntegerAtOnce	3,22	[3,19; 3,25]	17,89	[17,70; 18,08]
numpyIntegerAtOnce	3,42	[3,23; 3,61]	3,98	[3,80; 4,15]
mt19937arIntegerO3	4,29	[4,17; 4,42]	4,29	[4,17; 4,41]
numpyIntegerMtAtOnce	4,55	[4,42; 4,68]	5,15	[5,02; 5,27]
mt19937arIntegerO2	4,74	[4,68; 4,81]	4,74	[4,67; 4,81]
numpyIntegerPhiloxAtOnce	6,77	[6,63; 6,92]	7,37	[7,23; 7,51]
tensorflowIntegerTasksetAtOnce	7,08	[7,04; 7,13]	6,23	[6,20; 6,27]
mt19937arInteger	7,10	[6,96; 7,24]	7,10	[6,95; 7,24]
pytorchIntegerTasksetAtOnce	8,06	[8,00; 8,13]	7,12	[7,06; 7,18]
pytorchIntegerAtOnce	9,09	[8,99; 9,19]	8,93	[8,85; 9,01]
philoxInteger	90,06	[89,74; 90,39]	90,06	[89,73; 90,38]
pythonIntegerOneByOne	425,92	[424,24; 427,60]	425,91	[424,23; 427,58]
pythonIntegerTasksetAtOnce	486,11	[484,14; 488,09]	452,94	[451,32; 454,55]
pythonIntegerAtOnce	489,02	[487,30; 490,75]	453,29	[451,88; 454,70]
pytorchIntegerOneByOne	2281,79	[2248,98; 2314,61]	2282,33	[2249,51; 2315,16]
numpyIntegerMtOneByOne	6327,76	[6228,26; 6427,27]	6328,50	[6228,97; 6428,02]
numpyIntegerOneByOne	6458,61	[6396,91; 6520,31]	6459,50	[6397,77; 6521,23]
numpyIntegerPhiloxOneByOne	6552,21	[6472,50; 6631,91]	6553,13	[6473,41; 6632,85]

Table 9: Temps réel et temps utilisateur pour chaque experience, pour la génération aléatoire de 2^{30} entiers de 32 bits

Le code Mersenne Twister, dans son implémentation originale, prend le même temps pour les deux. Lorsqu'il est implémenté en utilisant NumPy, l'algorithme MT montre une divergence prononcée dans le temps de génération, prenant environ 4,5 secondes pour les entiers 32 bits contre 13 secondes pour les doubles 64 bits, alors que la version originale maintient une durée constante de 4 secondes pour chacun. Cependant, nous pouvons voir que les implémentations des PRNGs via les frameworks Python ont une bonne efficacité computationnelle, car les temps d'exécution du code Python et du code C se mélangent dans les classements de performance. Cependant, l'algorithme MT est significativement plus lent en pur Python. Pour les algorithmes PCG et Philox, les implémentations utilisant les technologies ML semblent surpasser les versions originales (en code C), malgré l'utilisation d'optimisations de compilation -O2 ou -O3 (lorsque nous avons pu les utiliser, car parfois, l'utilisation de l'optimisation de compilation conduit à un dysfonctionnement du code). La principale distinction entre le code C original et le code basé sur les technologies du machine learning réside dans l'inadaptabilité de ce dernier pour générer des nombres séquentiellement un par un, entraînant des performances significativement médiocres lorsqu'il s'agit de générer les nombres aléatoires séquentiellement et, dans le cas de TensorFlow, une impossibilité due à une surcharge de RAM, malgré la disponibilité de plus de 500 Go de RAM dans notre test. Tandis que lorsque l'on génère tous les nombres en une seule fois, il n'y a pas de soucis (avec les frameworks).

Enfin, en comparant le temps utilisateur et le temps réel, nous pouvons voir que TensorFlow est la seule technologie utilisant une parallélisation implicite. Nous pouvons donc supposer que si moins de cœurs étaient disponibles dans la machine, ou si la machine était surchargée en raison d'autres processus en cours, la génération avec TensorFlow aurait pris plus de temps que NumPy et serait similaire à PyTorch, car faire de la parallélisation sur une machine déjà surchargée n'améliorera pas les performances et peut même les aggraver. L'utilisation de taskset pour définir l'affinité d'un seul processus à un seul cœur montre une légère amélioration pour les frameworks du ML, sauf pour TensorFlow, en raison de sa parallélisation implicite native.

Dans les tables 11 et 12, nos observations se sont également étendues au temps nécessaire pour générer et stocker 2^{39} nombres pseudo-aléatoires (en minutes). Comme prévu, la durée de génération et de stockage de ces nombres est approximativement moitié moins longue pour les valeurs 32 bits par rapport aux valeurs 64 bits. Notamment, le générateur Mrg32k3a présente les performances les plus lentes parmi les générateurs codés en C, bien qu'il réussisse tous les tests statistiques. Nous pouvons noter que le générateur PCG est plus rapide que Mrg32k3a, et

parfois « Crush resistant ». Il est clairement inattendu que la génération d'entiers avec Python soit considérablement plus chronophage ; dans les cas 32 bits et 64 bits, c'est la technologie la moins efficace (en utilisant l'algorithme MT). Pour la création de 2^{39} nombres, nous avons employé une stratégie favorisant les frameworks du ML: générer des blocs de 2^{20} nombres séquentiellement jusqu'à atteindre les 2^{39}. Parmi ces frameworks, TensorFlow exige le plus de temps système et utilisateur, conséquence probable de sa parallélisation sous-jacente qui pourrait être problématique sur des ressources informatiques limitées. Il est intéressant de noter que les frameworks du ML montrent des performances compétitives par rapport aux implémentations en C. Ce résultat était inattendu et souligne le haut degré d'optimisation présent dans ces frameworks de haut niveau.

Générateur	Temps réel (s)	Temps réel 95% IC	Temps User (s)	Temps User 95% IC
tensorflowAtOnce	3,38	[3,28; 3,47]	32,33	[31,90; 32,76]
mt19937arO3	4,20	[4,06; 4,34]	4,20	[4,06; 4,34]
numpyTasksetAtOnce	4,35	[4,32; 4,39]	3,56	[3,52; 3,61]
mt19937arO2	4,50	[4,34; 4,67]	4,50	[4,34; 4,67]
well19937O3	4,96	[4,85; 5,08]	4,96	[4,85; 5,07]
well19937O2	4,97	[4,83; 5,12]	4,97	[4,83; 5,12]
numpyAtOnce	5,77	[4,73; 6,82]	5,75	[4,71; 6,79]
pytorchTasksetAtOnce	6,02	[5,94; 6,11]	5,31	[5,25; 5,36]
pytorchAtOnce	6,90	[6,76; 7,05]	7,01	[6,90; 7,12]
mt19937ar	7,48	[7,31; 7,66]	7,48	[7,31; 7,66]
tensorflowAtOnce	8,18	[8,12; 8,24]	6,67	[6,63; 6,72]
pcg64O3	11,00	[10,83; 11,17]	11,00	[10,83; 11,16]
pcg64O2	11,07	[10,92; 11,23]	11,07	[10,91; 11,23]
numpyMtAtOnce	13,08	[11,56; 14,60]	7,27	[7,16; 7,38]
well19937a	13,08	[13,08; 13,09]	13,08	[13,07; 13,09]
pcg64	13,18	[13,05; 13,31]	13,18	[13,05; 13,31]
numpyPhiloxAtOnce	13,26	[12,59; 13,92]	12,00	[11,89; 12,11]
mrg32k3aO3	19,97	[19,80; 20,14]	19,96	[19,79; 20,13]

mrg32k3aO2	31,47	[31,21; 31,72]	31,46	[31,21; 31,71]
pythonOneByOne	36,87	[36,23; 37,51]	36,86	[36,22; 37,50]
mrg32k3a	43,13	[42,96; 43,29]	43,12	[42,96; 43,29]
pythonTasksetAtOnce	69,52	[69,02; 70,03]	43,89	[43,66; 44,12]
pythonAtOnce	75,46	[72,06; 78,86]	48,48	[47,23; 49,73]
numpyMtOneByOne	319,89	[318,05;321,74]	320,83	[318,99 ; 322,67]
numpyPhiloxOneByOne	323,06	[321,28; 324,85]	323,98	[322,20; 325,76]
numpyOneByOne	330,98	[326,41; 335,54]	331,88	[327,31; 336,44]
pytorchOneByOne	2388,41	[2381,38; 2395,44]	2388,85	[2381,80; 2395,90]

Table 10 : Temps réel et temps utilisateur pour chaque experience, pour la génération aléatoire de 2^{30} doubles de 64 bits

Fichier	Temps réel	Temps User	Temps système
timeIntegerNumpySaving	91,2	27,6	63,4
timeIntegerNumpyMtSaving	97,6	36,4	61,1
timeIntegerPytorchSaving	121	41,6	79,4
timeIntegerNumpyPhiloxSaving	125,6	57,6	67,8
timeIntegerTensorflowSaving	131,4	826,9	149,6
timeIntegerPcgSaving	164,4	98,1	66
timeIntegerMtSaving	180,9	107,2	73,4
timeIntegerPhiloxSaving	229,8	150,8	78,8
timeIntegerPythonSaving	4957,1	4890,9	65,5

Table 11: Temps pris pour sauvegarder 2^{39} entiers de 32 bits pour chaque framework

Fichier	Temps réel	Temps User	Temps système
timeNumpySaving	170,6	33,1	137,3
timePytorchSaving	176,2	43,2	132,5
timeNumpyMtSaving	177,5	48,4	128,6
timeNumpyPhiloxSaving	231,9	96,7	135,0
timeMtSaving	281,8	126,0	154,4
timeWellSaving	283,0	127,4	155,2
timeTensorflowSaving	288,8	1893,2	393,4
timePcgSaving	346,8	185,8	160,6
timeMrgSaving	449,9	307,2	142,2
timePythonSaving	1355,8	1218,5	136,7

Table 12: Temps pris pour sauvegarder 2^{39} doubles de 64 bits pour chaque framework

IV.4.3. La reproductibilité de la consommation énergétique des PRNGs

Dans les tables 13 et 14, la consommation d'énergie est présentée en termes de joules par minute pour chaque expérience, correspondant respectivement aux entiers 32 bits et aux doubles 64 bits. D'après ces résultats, il est évident que les technologies du ML consomment environ 10 % d'énergie en plus que les implémentations traditionnelles en code C. Nous remarquons que le PRNG Philox est identifié comme un algorithme particulièrement énergivore par rapport à ses homologues. Il est supposé être cryptographiquement sécurisé, une caractéristique généralement associée à un temps de calcul accru en raison de la complexité supplémentaire. Cependant, les générateurs de nombres pseudo-aléatoires cryptographiquement sécurisés (CS-PRNG), bien que traditionnellement plus lents, réussissent souvent avec succès les tests statistiques. Selon l'article de M. O'Neill sur PCG, les variantes utilisées dans cette étude (PCG 128/64 XSL-RR et PCG 64/32 XSH-RR) sont également réputées être cryptographiquement sécurisées. Elles sont censées réussir à passer la batterie de tests BigCrush de TestU01, mais nous avons constaté que ce n'était pas forcément le cas. Des investigations supplémentaires seraient nécessaires pour confirmer l'affirmation selon laquelle

le PCG est cryptographiquement sécurisé, par exemple avec la suite de tests statistiques NIST (STS), mais cela dépasse le cadre de notre étude. Nous pouvons également constater que, pour tous les frameworks du ML, la génération par bloc utilise moins d'énergie que la génération un par un, en particulier dans le cas des entiers 32 bits. Dans la section suivante, nous parlerons de la consommation d'énergie, mais basée sur le temps réel nécessaire pour calculer, et non par minute. La figure 21 décrit les différences de consommation d'énergie par minute entre les différentes versions (code C ou framework du ML) d'un algorithme PRNG implémenté. Dans l'ensemble, lorsque nous comparons la consommation d'énergie de toutes les implémentations en C et de toutes les implémentations en Python, nous constatons environ 20 % d'énergie consommée en plus par minute pour les implémentations en Python. Dans cette figure, nous n'avons pas pris en compte la génération de nombres un par un avec NumPy, car elle consomme beaucoup plus d'énergie par minute (voir tables 13 et 14). Cette différence est probablement due à la faible efficacité de NumPy pour générer des nombres pseudo-aléatoires un par un, l'approche par bloc étant beaucoup plus efficace.

Générateur	Consommation énergétique (J/min)	Consommation énergétique 95% IC
pcg32Integer	3209,70	[3185.35; 3234.05]
mt19937arIntegerO2	3323,89	[3226.38; 3421.40]
mt19937arInteger	3419,13	[3285.98; 3552.27]
mt19937arIntegerO3	3607,74	[3476.79; 3738.68]
pythonIntegerTasksetAtOnce	4298,72	[4260.00; 4337.43]
pythonIntegerAtOnce	4375,21	[4110.53; 4639.88]
numpyIntegerAtOnce	4688,14	[4669.67; 4706.61]
numpyIntegerTasksetAtOnce	4766,35	[4747.59; 4785.10]
pytorchIntegerTasksetAtOnce	4766,96	[4748.06; 4785.86]
tensorflowIntegerTasksetAtOnce	4800,25	[4784.14; 4816.36]
pytorchIntegerAtOnce	4812,53	[4775.33; 4849.73]
philoxInteger	4845,48	[4824.15; 4866.81]
numpyIntegerMtAtOnce	4847,67	[4828.75; 4866.58]

numpyIntegerPhiloxAtOnce	4849,22	[4829.80; 4868.63]
tensorflowIntegerAtOnce	4893,72	[4862.41; 4925.04]
pythonIntegerOneByOne	5410,52	[5185.95; 5635.09]
numpyIntegerOneByOne	5925,94	[5579.41; 6272.47]
pytorchIntegerOneByOne	8846,44	[8492.64; 9200.24]
numpyIntegerMtOneByOne	45223,56	[42623.11; 47824.02]
numpyIntegerPhiloxOneByOne	64212,81	[61111.12; 67314.50]

Table 13: Consommation énergétique en Joule par minute pour chaque expérience, sur des entiers de 32 bits

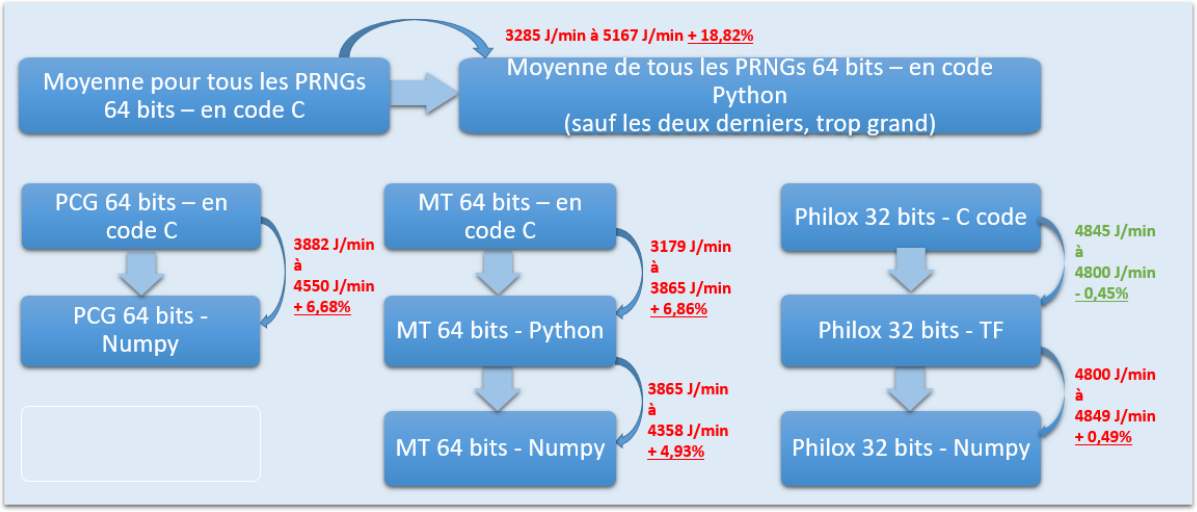


Figure 21: Consommation énergétique en Joule par minute - Différences entre les implémentations en C et en Python, pour chaque PRNG

Générateur	Consommation énergétique (J/min)	Consommation énergétique 95% IC
mrg32k3aO2	2750,80	[2744,56; 2757,03]
mrg32k3a	2783,59	[2744,86; 2822,31]
well19937O2	2979,42	[2970,81; 2988,02]
well19937O3	2991,04	[2981,97; 3000,10]
mrg32k3aO3	3000,87	[2941,58; 3060,15]

well19937a	3040,66	[3008,42; 3072,90]
mt19937arO2	3179,17	[3168,29; 3190,04]
mt19937ar	3186,49	[3172,36; 3200,62]
mt19937arO3	3226,83	[3159,98; 3293,67]
pythonOneByOne	3865,71	[3806,83; 3924,59]
pcg64O3	3882,55	[3812,37; 3952,73]
pcg64O2	3925,18	[3872,56; 3977,79]
pythonTasksetAtOnce	3994,90	[3871,83; 4117,96]
pytorchTasksetAtOnce	4348,02	[4306,74; 4389,30]
numpyMtAtOnce	4358,07	[4168,48; 4547,65]
pcg64	4473,07	[4459,61; 4486,52]
numpyTasksetAtOnce	4550,28	[4530,53; 4570,03]
tensorflowTasksetAtOnce	4762,63	[4750,04; 4775,22]
numpyPhiloxAtOnce	5033,01	[4875,79; 5190,22]
tensorflowAtOnce	5131,12	[5100,87; 5161,37]
pytorchOneByOne	5630,90	[5233,18; 6028,62]
numpyAtOnce	5906,65	[5670,16; 6143,13]
numpyOneByOne	5935,29	[5772,42; 6098,16]
pythonAtOnce	6521,71	[6078,98; 6964,44]
pytorchAtOnce	7141,50	[6891,12; 7391,88]
numpyMtOneByOne	37375,55	[36671,35; 38079,75]
numpyPhiloxOneByOne	37590,44	[36613,97; 38566,92]

Table 14: Consommation énergetique en Joule par minute pour chaque expérience, pour des doubles 64 bits

Les tables 15 et 16 illustrent la consommation d'énergie en joules pendant le temps réel pris par chaque expérience (par exemple, la génération de 2^{30} nombres aléatoires, en fonction de l'algorithme et de la technologie). Les précédentes figures concernaient la consommation d'énergie en joules par minute, mais nous nous concentrons ici sur la consommation d'énergie pendant le temps pris pour chaque expérience. Bien que les implémentations en C soient plus économes en énergie par minute, le temps d'exécution global compétitif de Python lui permet

de rivaliser avec les implémentations en C. Les implémentations sous-jacentes des technologies du machine learning en Python utilisent souvent C, C++ ou CUDA. Pour les entiers 32 bits, l'algorithme PCG démontre une efficacité notable, surpassant les autres implémentations en C et suivi de près par NumPy, tout en maintenant une consommation d'énergie raisonnable compte tenu de son temps d'exécution. L'algorithme Mersenne Twister en C affiche la plus grande cohérence, produisant des résultats similaires pour la génération d'entiers et de doubles. Malheureusement, à part MT et PCG, les autres PRNGs basés sur C sont dédiés à un type de sortie spécifique. Par exemple, Mrg32k3a et Well (Panneton *et al.* 2006) sont dédiés à la génération de valeurs doubles et Philox génère des entiers. Contrairement à Mersenne Twister, PCG présente une différence significative entre ses performances de génération d'entiers (32 bits) et de doubles (64 bits). Bien que l'optimisation O2 ou O3 n'affecte pas la consommation d'énergie par minute, sa réduction du temps de calcul total contribue à diminuer l'énergie totale nécessaire. Il est à noter que pour PCG et Philox (dans la génération de 32 bits), les optimisations O2 et O3 n'ont pas été appliquées car leur utilisation provoque des dysfonctionnements du code, entraînant une terminaison immédiate sans exécuter les opérations prévues. Le niveau d'optimisation O3 est connu pour être agressif et peut souvent produire des résultats non reproductibles ou un comportement étrange, comme cela est documenté. Cependant, c'est la première fois, en plus de 30 ans d'expériences informatiques pour mon encadrement, que nous observons que le niveau O2 produit des résultats inutilisables et donc non reproductibles.

Générateur	Consommation énergétique durant le temps réel (J)	Consommation énergétique durant le temps réel 95% IC
pcg32Integer	131,17	[130.18; 132.17]
numpyIntegerTaskseAtOnce	206,41	[205.60; 207.22]
mt19937arIntegerO3	258,09	[248.73; 267.46]
tensorflowIntegerAtOnce	262,52	[260.84; 264.20]
mt19937arIntegerO2	262,77	[255.07; 270.48]
numpyIntegerAtOnce	267,33	[266.28; 268.38]
numpyIntegerMtAtOnce	367,77	[366.33; 369.20]
mt19937arInteger	404,65	[388.90; 420.41]

numpyIntegerPhiloxAtOnce	547,38	[545.19; 549.58]
tensorflowIntegerTasksetAtOnce	566,63	[564.73; 568.53]
pytorchIntegerTasksetAtOnce	640,54	[638.00; 643.08]
pytorchIntegerAtOnce	728,84	[723.20; 734.47]
philoxInteger	7273,10	[7241.08; 7305.12]
pythonIntegerTasksetAtOnce	34827,75	[34514.05; 35141.44]
pythonIntegerAtOnce	35659,70	[33502.49; 37816.92]
pythonIntegerOneByOne	38407,42	[36813.25; 40001.59]
pytorchIntegerOneByOne	336428,77	[322973.74; 349883.80]
numpyIntegerOneByOne	637889,20	[600587.67; 675190.72]
numpyIntegerMtOneByOne	4769399,46	[4495148.41; 5043650.52]
numpyIntegerPhiloxOneByOne	7012261,80	[6673546.74; 7350976.87]

Table 15: Consommation énergétique en Joule en extrapolant sur la durée réelle d'execution, pour les entiers 32 bits

Générateur	Consommation énergétique durant le temps réel (J)	Consommation énergétique durant le temps réel 95% CI
mt19937arO3	225,74	[221,06; 230,42]
mt19937arO2	238,60	[237,79; 239,42]
well19937O2	247,04	[246,33; 247,75]
well19937O3	247,41	[246,66; 248,16]
tensorflowAtOnce	288,69	[286,99; 290,40]
numpyTasksetAtOnce	330,27	[328,83; 331,70]
mt19937ar	397,49	[395,73; 399,26]
pytorchTasksetAtOnce	436,27	[432,13; 440,41]
numpyAtOnce	568,32	[545,56; 591,07]
tensorflowAtOnce	649,36	[647,65; 651,08]
well19937a	663,01	[655,98; 670,04]

pcg64O3	711,70	[698,83; 724,56]
pcg64O2	724,40	[714,69; 734,11]
pytorchAtOnce	821,43	[792,63; 850,23]
numpyMtAtOnce	950,24	[908,91; 991,58]
pcg64	982,61	[979,65; 985,56]
mrg32k3aO3	998,58	[978,85; 1018,31]
numpyPhiloxAtOnce	1111,94	[1077,21; 1146,68]
mrg32k3aO2	1442,61	[1439,33; 1445,88]
mrg32k3a	2000,77	[1972,93; 2028,61]
pythonOneByOne	2375,43	[2339,25; 2411,61]
pythonTasksetAtOnce	4629,02	[4486,43; 4771,62]
pythonAtOnce	8202,27	[7645,45; 8759,08]
numpyOneByOne	32740,55	[31842,13; 33638,98]
numpyMtOneByOne	199270,14	[195515,66; 203024,62]
numpyPhiloxOneByOne	202400,80	[197143,08; 207658,52]
pytorchOneByOne	224148,23	[208316,06; 239980,39]

Table 16: Consommation énergétique en Joule en extrapolant sur la durée réelle d'execution, pour les doubles 64 bits

IV.4.4. La reproductibilité des résultats numériques

Une découverte importante de cette étude est l'absence de reproductibilité des nombres générés sur différentes plateformes. L'algorithme Mersenne Twister, initialisé avec la même graine, produira des nombres différents avec les différentes implémentations en C, Python et NumPy. Il en va de même pour PCG avec NumPy et le code C, ainsi que pour Philox avec NumPy et TensorFlow. La raison pourrait être que nous utilisons des graines pour initialiser notre PRNG, au lieu de définir l'état complet. Cette fonction transformant une graine en un état complet peut différer entre les technologies ou les frameworks, ce qui peut entraîner une perte de reproductibilité numérique entre les technologies.

IV.4.5. La reproductibilité de la qualité statistique des PRNGs

Nous examinons maintenant la qualité des nombres pseudo-aléatoires générés par chaque technologie. Les résultats pour les entiers sont présentés dans la table 17, et ceux pour les doubles dans la table 18. Tout d'abord, nous remarquons que la qualité de la génération de doubles se comporte davantage comme prévu que celle des entiers. En effet, chaque algorithme PRNG est connu pour échouer à certains tests spécifiques de Big Crush (ou aucun), nous pouvons donc utiliser ces tests comme marqueurs pour reconnaître un PRNG ou un autre. Dans la génération de doubles, toutes les implémentations de l'algorithme Mersenne Twister — y compris MT en C, Python et NumPy — ont échoué aux tests LinearComp 80 et 81, ce qui est conforme aux attentes, car ces tests sont liés à la crypto-sécurité. L'algorithme Well a également montré des échecs similaires, sa structure interne étant similaire à celle de MT (Linear Feedback Shift Registers). À l'inverse, le PCG et sa variante NumPy utilisant l'algorithme PCG 128/64 XSL-RR ont passé tous les tests, corroborant les affirmations de l'auteur de PCG. D'autre part, la documentation de NumPy indique que des faiblesses statistiques ont été identifiées dans l'algorithme PCG64 lorsqu'il est utilisé dans des contextes massivement parallèles. En conséquence, une nouvelle version appelée PCG64DXSM a été introduite. Cependant, toutes les versions de PCG de l'auteur original ont récemment été signalées comme ayant des défauts statistiques, en plus d'être plus lentes que d'autres PRNG plus anciens, par Vigna (mentionné ci-dessous avec Blackman — voir la page d'accueil de Vigna : https://pcg.di.unimi.it/pcg.php). L'algorithme Philox de NumPy a échoué au test BirthdaySpacings, contrairement à son homologue de TensorFlow, qui a passé toutes les évaluations. D'après notre expérience, nous reconnaissons que les PRNG peuvent parfois échouer à des tests auxquels ils ne sont pas censés échouer (Antunes et Hill 2023). Un examen plus approfondi du comportement de chaque PRNG nécessiterait de multiples répliques avec des états initiaux variés pour vérifier la cohérence sur toute la période. Malheureusement, nous avons dû exclure les données de PyTorch de la table 18 en raison de son échec à 62 tests statistiques (sur 106 tests). Ce résultat nécessite une enquête supplémentaire, ce taux d'échec élevé suggère une qualité statistique inférieure (58 % de la batterie Big Crush échouée). Il est intéressant de noter que la génération de nombres pseudo-aléatoires de PyTorch sur des entiers 32 bits était bien meilleure, échouant à seulement 3 tests.

Concernant les entiers 32 bits, les résultats ont été un peu plus surprenants. Tout d'abord, nous pouvons constater que, bien que le code C original de Mersenne Twister échoue aux deux tests 80 et 81, l'implémentation NumPy de MT n'a échoué qu'à un seul test. De manière surprenante, la version Python a réussi tous les tests ; des investigations supplémentaires avec

différents états initiaux pourraient être intéressantes. PCG et Philox, dans leurs différentes implémentations, ont échoué à certains tests, mais restent tout de même de bons générateurs de qualité. Il est intéressant de noter qu'ils échouent effectivement à certains tests, alors que les auteurs supposent qu'ils n'échouent à aucun. De plus, ils n'ont pas échoué aux mêmes tests. Par exemple, la version NumPy de Philox a échoué au test 49 MaxOft, tandis que la version TensorFlow de Philox a échoué au test 9 CollisionOver. Cela nous fait penser que ces PRNGs pourraient échouer à différents tests statistiques si nous tentions de faire des réplications sur les périodes des PRNGs, en utilisant différents états du PRNG. Une observation inattendue a été l'échec de PyTorch à trois tests spécifiques, notamment les tests RandomWalk et deux tests LinearComp. Bien que la documentation de PyTorch suggère Philox comme son PRNG sous-jacent, les échecs observés ressemblent à la « signature » d'un Mersenne Twister, ce qui soulève des questions sur son implémentation.

Générateur	Nombre de tests ratés	Tests ratés
philoxInt32	6	34 Gap, 35 Gap, 36 Gap, 37 Gap, 65 SumCollector, 68 MatrixRank
pytorchInt32	3	77 RandomWalk1, 80 LinearComp, 81 LinearComp
pcgInt32	1	5 CollisionOver
mtInt32	2	80 LinearComp, 81 LinearComp
numpyMtInt32	1	80 LinearComp
numpyPhiloxInt32	1	49 MaxOft
numpyInt32	0	
tensorflowInt32	1	9 CollisionOver
pythonInt32	0	

Table 17: Tests BigCrush ratés pour chaque expérience, sur les entiers 32 bits

Générateur	Nombre de tests ratés	Tests ratés
pcgReal	0	
tensorflowReal	0	
MRG32k3aReal	0	
wellReal	2	80 LinearComp, 81 LinearComp
numpyReal	0	
mtReal	2	80 LinearComp, 81 LinearComp
numpyPhiloxReal	1	21 BirthdaySpacings
pythonReal	2	80 LinearComp, 81 LinearComp
numpyMtReal	2	80 LinearComp, 81 LinearComp

Table 18: Test BigCrush ratés pour chaque expérience, sur les doubles 64 bits. PyTorch est exclu car 62 tests ratés.

D'après ce qui a été découvert, de nouvelles questions se posent : si la perte de reproductibilité ne provient pas des fonctions de seeding, cela nous amène à une question cruciale : comment pouvons-nous vérifier que l'algorithme utilisé est une implémentation correcte du générateur ? Pour nous, cette question reste ouverte. La solution idéale serait que les auteurs originaux fournissent un échantillon de nombres pseudo-aléatoires générés, que nous devrions pouvoir comparer avec les nombres que nous générons, pour garantir une reproductibilité parfaite. À notre connaissance, Mersenne Twister est le seul PRNG à offrir cette fonctionnalité avec le résultat attendu. La reproductibilité numérique est non seulement importante pour l'avancement de la science, mais aussi pour le débogage (Hill 2015). Le changement de matériel ou de pile logicielle affecte-t-il la reproductibilité d'un PRNG ? Ce que nous observons, c'est que la portabilité des PRNGs ne doit pas être considérée comme acquise. Ici, nous utilisons différentes technologies dans le même environnement, et nous obtenons des résultats différents et une qualité statistique différente. Une autre manière d'identifier le PRNG sans se baser sur le résultat numérique serait de réaliser des tests statistiques pour essayer d'identifier l'algorithme sous-jacent, car certains tests statistiques échoués peuvent servir de marqueurs pour certains PRNG. Cependant, comme nous l'avons trouvé dans une étude approfondie (Antunes et Hill 2023), le même algorithme PRNG peut échouer à plusieurs tests statistiques différents. Par exemple, sur 4096 réplications, il apparaît que l'algorithme Mersenne

Twister échoue à tous les 106 tests BigCrush au moins une fois. Nous pourrions nous attendre à un comportement similaire de la part d'autres PRNG. Cela nécessiterait également des investigations supplémentaires. Assurer l'utilisation d'un algorithme spécifique, en l'absence de reproductibilité numérique parfaite, est loin d'être trivial.

Dans les discussions autour du calcul haute performance, il est indéniable que le machine learning est un domaine à forte consommation en termes de temps, d'investissement financier et d'énergie. Avec la marche inexorable vers une plus grande puissance de calcul et malgré l'innovation technologique, ces coûts n'ont fait qu'augmenter en raison de l'inflation des prix du matériel et de l'énergie. Pendant ce temps, le ML est devenu un outil indispensable dans une multitude de domaines tels que les modèles de langage large (LLM), mais aussi dans des domaines plus courants comme les véhicules autonomes, la santé, etc. La sophistication des modèles de ML entraîne ses propres exigences en matière de ressources de calcul et d'énergie. En considérant la génération de nombres pseudo-aléatoires — un composant essentiel pour les processus stochastiques, les simulations et même pour le fonctionnement des algorithmes de ML eux-mêmes — la comparaison entre les générateurs traditionnels codés en C tels que Philox, Mersenne Twister et PCG, et ceux implémentés dans les frameworks de ML (utilisant PyTorch, TensorFlow, Python et NumPy), est intéressante. La consommation d'énergie est un facteur critique ; bien qu'il n'y ait pas de données sur les coûts énergétiques exacts de la génération de nombres aléatoires dans l'entraînement des réseaux neuronaux, il est raisonnable de supposer que la proportion n'est pas négligeable. Evaluer la proportion exacte de temps utilisé dans la génération de nombres pseudo-aléatoires, en fonction de la taille du réseau neuronal, serait précieux. Avec un réseau neuronal comme GPT-4 LLM fourni par OpenAI, qui possède environ 175 milliards de paramètres (arêtes du graphe), on peut facilement imaginer qu'un très grand nombre de nombres pseudo-aléatoires ont été utilisés. Générer des nombres pseudo-aléatoires fait partie intégrante de la phase d'entraînement des réseaux neuronaux, notamment dans des processus tels que l'initialisation des poids, le mélange, et pendant la descente de gradient stochastique où l'aléatoire est utilisé pour garantir la convergence. Les résultats de notre étude suggèrent que les implémentations des PRNGs dans les frameworks de ML peuvent égaler la qualité statistique et la vitesse de leurs homologues en code C. Cependant, la facilité et la rapidité de génération de nombres pseudo-aléatoires à l'aide des frameworks de ML entraînent une légère augmentation du coût de consommation énergétique dédié à cette tâche (environ 10%).

IV.4.6. Discussion

Les frameworks du machine learning utilisent des générateurs de nombres pseudo-aléatoires dans l'entraînement des réseaux neuronaux. L'apport de sources stochastiques s'est avéré intéressant. Cependant, l'investigation de la qualité des nombres pseudo-aléatoires générés, ainsi que de leur temps de génération et de leurs exigences en matière d'énergie, reste incomplète. Nous avons donc étudié l'efficacité de la génération de nombres pseudo-aléatoires au sein des technologies du machine learning par rapport aux implémentations traditionnelles en langage C. Nous avons examiné Python, PyTorch, TensorFlow et NumPy. Nos résultats indiquent que les différentes bibliothèques et frameworks utilisant le langage Python sont bien optimisés. La bibliothèque NumPy se distingue en termes de consommation de temps et de qualité, s'alignant étroitement avec les PRNGs implémentés en C. Néanmoins, deux inconvénients ont été identifiés : premièrement, la génération de nombres pseudo-aléatoires dans les technologies du machine learning a tendance à consommer environ 10 % d'énergie en plus ; deuxièmement, il existe une incohérence dans la répétabilité numérique lorsqu'on utilise des graines identiques avec différentes implémentations de PRNGs, ce qui constitue un problème de portabilité. Cela remet en question la fidélité des implémentations des PRNGs dans les différent framework, par rapport aux algorithmes originaux. L'implémentation des PRNGs dans les outils de machine learning devrait donner des résultats identiques lorsqu'ils sont initialisés de manière similaire à leurs homologues en C. Néanmoins, ils n’offrent que la possibilité d’utiliser une méthode de seeding, tandis que pour garantir l’équivalence, spécifier l’état complet du générateur est nécessaire, si l’implémentation de l’algorithme est identique, mais que la méthode de seeding est différente.

Bien que le site officiel de PCG le qualifie de « très rapide » par rapport à la vitesse « acceptable » de Mersenne Twister, notre analyse suggère que de telles affirmations sont exagérées et même fausses, puisque la génération de valeurs doubles (64 bits), essentielle pour de nombreuses simulations, est 2,5 fois plus rapide avec le MT original (nous avons remarqué certaines différences de performance entre la génération de valeurs pseudo-aléatoires entières 32 bits et doubles 64 bits). L'implémentation en C de PCG prend le même temps que l'implémentation NumPy. De plus, PCG a également échoué à certains tests BigCrush bien qu'il soit censé y résister. PCG devrait être évité pour le calcul massivement parallèle, comme indiqué dans la documentation NumPy qui suggère d'utiliser PCG64DXSM. Des recherches supplémentaires sont nécessaires pour une exploration plus approfondie de chaque PRNG. Bien que l'impact de la qualité des PRNGs sur les résultats de l'entraînement des réseaux neuronaux

n'ait pas été largement étudié, les informations tirées des études récentes suggèrent que la qualité des PRNGs pourrait effectivement influencer la performance du réseau neuronal résultant, sur la base de métriques de qualité (Huk et al. 2021 ; Koivu et al. 2022).

Dans cette section, nous avons répondu aux questions suivantes : les générateurs de nombres pseudo-aléatoires implémentés dans les frameworks de machine learning produisent-ils les mêmes résultats que leurs implémentations originales en code C lorsqu'ils sont initialisés de manière identique ? La qualité statistique des nombres générés par ces frameworks est-elle équivalente à celle des PRNGs dans leur code original ? Le processus de génération de nombres aléatoires dans les frameworks de machine learning est-il plus chronophage par rapport à celui des implémentations C d'origine ? Ce processus est-il également plus énergivore ? Enfin, y a-t-il une équivalence dans la génération de nombres entiers 32 bits et de nombres à double précision 64 bits entre les frameworks de machine learning et leurs homologues en code C, compte tenu des points précédents ?

IV.5. Cas d'étude de reproductibilité sur une application de machine learning

IV.5.1. Discussions sur l'état et la graine d'un générateur

Nous avons également étudié une application non-reproductible utilisant le framework Tensorflow. L'ensemble d'entraînement utilisé dans l'étude et dans l'implémentation logicielle que nous avons vérifiée est le jeu de données MNIST, contenant un ensemble de 70 000 images de chiffres manuscrits. Ce jeu de données MNIST est assez bien connu, étiqueté et facile à regrouper, nous permettant de contrôler les résultats obtenus. L'algorithme a été implémenté en Python en utilisant la bibliothèque NumPy et les frameworks Sci-kit learn et Keras comme interface de programmation d'application (API) de haut niveau et de qualité industrielle de la plateforme TensorFlow. Pour évaluer la pertinence des partitions fournies par les modèles développés, deux métriques ont été utilisées. Tout d'abord, un coefficient de silhouette mesurant la qualité d'une partition d'un ensemble de données basé sur la séparabilité entre les clusters et la compacité de chaque cluster. La deuxième métrique est l'Indice de Rand Ajusté (ARI) qui évalue la similarité entre deux partitions du même ensemble de données en mesurant le taux d'accord entre elles.

Avec le code initial produit au cours du stage de l'année précédente, nous avons d'abord observé que les partitions obtenues en utilisant les différents modèles obtenus avec Deep Embedded Clustering (DEC), qui présentaient la plupart du temps un coefficient de silhouette

très proche de 1, ce qui indiquait une bonne séparation des données. Cependant, lorsque nous avons évalué l'ARI de ces différentes partitions les unes par rapport aux autres, nous avons systématiquement observé des résultats très proches de 0. Les partitions étaient donc très différentes d'un modèle généré à l'autre, sans raison claire. De plus, les résultats numériques étaient différents d'une exécution à l'autre dans les mêmes conditions. Ce manque de répétabilité était problématique et il était impossible de se baser sur une base instable. L'autre aspect que nous avons rencontré est lié à un manque de reproductibilité. Les ressources de l'étude initiale du stage ne nous permettaient pas de générer des modèles proches de ceux obtenus. Certaines des bibliothèques Python utilisées, comme Keras, avaient été mises à jour, rendant certaines lignes de code obsolètes. De plus, la reprise du projet était complexe, notamment en raison d'un manque de documentation. Voici le lien vers notre dépôt Gitlab : https://gitlab.isima.fr/jedomanski/machine-learning-repeatability-quest.

Dans notre tentative de reproduire de manière fiable les résultats de DEC, diverses méthodes ont été appliquées pour identifier si le problème résidait dans les paramètres de configuration. Tout d'abord, conformément à l'article de DEC, nous avons défini : Stochastic Gradient Descent (SGD) comme optimiseur, des dimensions de réseau d-500-500-2000-10, avec « d » la dimension de la base de départ, chaque couche étant pré-entraînée pendant 50 000 itérations, avec un taux de dropout de 20 %. L'auto-encodage est ensuite affiné sur 100 000 itérations sans dropout. La taille des batchs était de 256 et le taux d'apprentissage était fixé à 0,1, puis divisé par 10 toutes les 20 000 itérations, et la méthode de dégradation du poids était réglée sur 0. Nous avons expérimenté différentes tailles de mini-batchs, passant de l'original 256 à seulement 32, puis nous avons essayé de charger des poids de modèles précédemment entraînés, afin d'identifier si l'incohérence entre les partitions résultantes était due à un problème d'apprentissage.

Une autre approche consistait à utiliser la fonction de minimisation SGD. D'une part, nous avons étudié les résultats de cette fonction en utilisant des diagrammes en boîte, et d'autre part, nous avons essayé de faire l'entrainement avec une descente de gradient non stochastique (la descente de gradient simple), dans le but de limiter les sources de hasard introduites par le SGD. Un problème de versionnage identifié plus tôt a également été étudié. En raison de la mise à jour de la bibliothèque Keras, des lignes de code utilisant l'optimiseur étaient devenues obsolètes. Pour maintenir l'algorithme en fonctionnement, des remplacements ont dû être effectués, avec le moins de modifications possible. Un outil d'analyse de modèle de réseau neuronal, « netron.app », a été utilisé pour identifier la version initiale utilisée dans notre projet,

c'était la version v2.11.0, tandis que la dernière version que nous utilisons maintenant est v2.13.1. Toutes les tentatives produisaient encore des problèmes de répétabilité.

Le code n'appelait pas explicitement les sources de hasard, mais nous étions conscients d'au moins trois sources potentiellement cachées. Le générateur utilisé en Python est Mersenne Twister, le générateur par défaut dans NumPy est PCG (Permuted Congruential Generator) et celui utilisé dans TensorFlow est par défaut Philox (un générateur introduit en 2011 à la conférence Supercomputing). Le code original mélangeait les trois générateurs dans sa « boîte noire » et ne spécifiait pas d'états initiaux pour les trois générateurs de nombres pseudo-aléatoires. Nous avons ensuite comparé les modèles générés avec la même graine sur la même machine pour les trois différents générateurs, puis nous avons fait le test sur un autre serveur, en gardant toujours la même graine (bien que les générateurs n'aient pas les mêmes structures internes). Nous avons étudié les huit combinaisons possibles, avec ou sans initialisation de graine pour les trois générateurs, et observé les résultats de répétabilité ci-dessous (Table 19).

Mersenne Twister	**PCG**	**Philox**	**Résultats**
Non initialisé	Non initialisé	Non initialisé	Résultats non répétable
Non initialisé	Non initialisé	Initialisé	Résultats non répétable
Non initialisé	Initialisé	Non initialisé	Résultats non répétable
Non initialisé	Initialisé	Initialisé	Résultats non répétable
Initialisé	Non initialisé	Non initialisé	Résultats non répétable
Initialisé	Non initialisé	Initialisé	**Résultats répétable**
Initialisé	Initialisé	Non initialisé	Résultats non répétable
Initialisé	Initialisé	Initialisé	**Résultats répétable**

Table 19: Exploration des différentes initialisations avec les trois générateurs de nombres pseudo-aléatoires trouvés dans le logiciel

Avec nos comparaisons « run-to-run », nous avons identifié deux combinaisons conduisant à des résultats répétables sur la même machine, nous permettant de commencer une étude de reproductibilité même si nous ne trouvons pas les mêmes résultats sur différentes machines. Ce travail est publié dans l'article (Hill *et al.* 2024).

IV.5.2. Discussion

Les scientifiques comptent sur la reproductibilité pour établir la crédibilité des découvertes et assurer la robustesse des connaissances scientifiques. Dans le domaine des programmes du machine learning, nous avons trouvé des défis complexes qui rendent souvent la reproduction des résultats des articles scientifiques presque impossible. Ce fut notre cas lorsque nous avons

tenté d'étudier l'analyse de réseau neuronal non supervisée et non linéaire connue sous le nom de Deep Embedded Clustering. Lors de nos premières expériences avec le jeu de données MNIST, nous avons identifié un problème de répétabilité lié aux sources de hasard cachées dans le langage, les bibliothèques et le framework utilisés. Nous avons atteint une répétabilité run-to-run grâce à une initialisation minutieuse des générateurs de nombres pseudo-aléatoires par défaut de Python et Keras. Les frameworks de machine learning sont fournis avec une API de seeding, qui cause souvent de la confusion car les graines fournies par les programmeurs ne sont pas les états des générateurs, mais simplement une valeur pour certains d'entre eux. De plus, les appels aux sources de hasard sont parfois cachés dans les fonctions de l'API, et la parallélisation des générateurs par défaut (qui sont parfois statistiquement faibles) est également floue. Même si nous avons pu atteindre le « graal » de la répétabilité sur la même machine, la portabilité d'un serveur à un autre reste un problème, nécessitant une investigation plus approfondie.

IV.6. L'impact du matériel physique sur la reproductibilité des mesures de performances : Etude sur le simultaneous multi-threading

IV.6.1. Notre étude

Nous avons conçu une expérience pour répondre aux trois questions soulevées dans la section proposition. Nous voulons connaître l'impact du SMT (Simultaneous multi-threading) sur les performances, et nous étudions le SMT d'AMD et l'HT d'Intel. La date de sortie du processeur AMD est mi-2019, tandis que celle du processeur Intel est fin 2013. Cela nous permet d'étudier l'évolution à moyen terme ou la cohérence de l'impact des performances du SMT, et de confirmer ou infirmer les résultats que nous obtiendrions sur chaque configuration de clusters différente avec la reproductibilité à l'esprit. Cette étude a été publié dans l'article (Antunes et Hill 2023).

Pour AMD, nous avons deux nœuds. Chaque nœud dispose de deux processeurs AMD EPYC Rome 7452, chacun avec 32 cœurs, 64 threads, 2 Mo de cache L1, 16 Mo de L2, 128 Mo de L3 et 512 Go de RAM (DDR4). La bande passante mémoire maximale est de 190,7 Go/s. Chaque cœur peut fonctionner à 3,35 GHz. Le système d'exploitation est Ubuntu 20.04, avec la version du noyau 5.4.0.153-generic, et nous travaillons avec la version 9.4.0 du compilateur gcc/g++. Les deux nœuds sont identiques, mais sur le premier nœud, nommé ici node-NoSMT, le SMT est désactivé, et sur le second, node-SMT, le SMT est activé. Cela porte

le node-NoSMT à 64 cœurs (physiques), et le node-SMT à 128 cœurs logiques (64 cœurs physiques proposant 128 cœurs logiques). Nos expériences sont facilement équilibrées car nous exécutons des réplications de simulations stochastiques (massivement parallèle). Cela signifie que nous avons des tâches identiques, mais chacune d'elles a un flux aléatoire différent afin d'obtenir des résultats avec des exécutions statistiquement indépendantes. Nous avons sélectionné le générateur pseudo-aléatoire Mersenne Twister et l'avons utilisé avec différents états de flux « indépendants ». La technique de parallélisation utilisée est connue sous le nom de méthode MRIP (Multiple Replications In Parallel) (Passerat-palmbach *et al.* 2011). Nous avons considéré quatre cas ; nous exécutons 4 types d'expériences : (1) 128 tâches parallèles sur node-NoSMT sans définir l'affinité ; (2) 128 tâches parallèles sur node-NoSMT avec définition de l'affinité ; (3) 128 tâches parallèles sur node-SMT sans définir l'affinité et (4) 128 tâches parallèles sur node-SMT avec définition de l'affinité.

Pour voir si nous pouvons avoir une croissance stable de l'accélération (speedup) via l'augmentation du nombre de coeur, nous avons fixé la taille du problème à 128 tâches de simulation, et nous avons exécuté ce problème sur 1, 2, 4, 8, 16, 32, 64 et 128 cœurs. La valeur de l'accélération est calculée en divisant le temps nécessaire pour exécuter la charge de travail (128 processus dans notre cas) par N cœurs par le temps nécessaire pour exécuter la charge de travail sur un seul cœur (Accélération = TempsSurNcoeurs / TempsSurUnCoeur). Évidemment, pour les expériences sur node-SMT avec le SMT activé et 128 cœurs logiques, une tâche peut être assignée à chaque cœur. Sur node-NoSMT avec seulement 64 cœurs physiques, deux tâches devront être traitées sur chaque cœur. Le résultat attendu est que node-SMT devrait obtenir de meilleures performances, d'environ 20 % comme mentionné dans les travaux précédents. Les deux nœuds que nous utilisons pour ce test sont situés dans un cluster accessible avec Slurm. Tous les scripts et le code utilisés sont disponibles sur Gitlab (https://gitlab.isima.fr/beantunes/simultaneousmultithreading-evaluation). Nous avons réservé toute la capacité des deux nœuds (--exclusive avec Slurm), même si nous n'utilisons qu'un seul cœur par exemple, afin de ne pas interférer avec d'autres travaux que d'autres chercheurs de notre laboratoire pourraient avoir lancés. L'affinité est définie avec la commande linux taskset. Pour node-SMT (avec SMT et 128 cœurs logiques), nous avons d'abord défini l'affinité uniquement sur les cœurs physiques autant que possible. Ensuite, pour 128 coeurs, l'affinité assigne un processus à chaque cœur logique (de 0 à 127). Pour node-NoSMT, avec taskset, nous avons fait de même, une tâche par cœur physique jusqu'à 64, puis deux tâches par cœur pour traiter les 128 tâches.

Pour Intel, nous avons également deux nœuds. Chaque nœud dispose de deux Intel Xeon E5-2650 v2, chacun avec 8 cœurs, 16 threads, 32 Ko de cache L1, 128 Ko de cache L2, 10 Mo de cache L3 et 128 Go de RAM. Le système d'exploitation est CentOS Linux 7.9.2009, et nous travaillons avec la version 4.8.4 du compilateur gcc/g++. Le turbo boost est activé, pour passer de 1,2 GHz à 3,4 GHz. Les deux nœuds sont exactement les mêmes, mais sur le premier nœud (nommé node-NoHT), l'HT est désactivé, et sur le second (nommé node-HT), l'HT est activé. En raison des limites techniques du matériel à notre disposition, l'expérience sur l'hyper-threading Intel est plus petite que sur AMD (32 processus contre 128 processus). La configuration logicielle des clusters est assez similaire, mais avec des versions différentes du système d'exploitation, du compilateur, des bibliothèques. Les marques et générations de processeurs sont également différentes, avec des implémentations SMT différentes. En raison de ces différences, cela nous permet d'étudier l'impact du SMT sur les performances en fonction du type d'applications, et de reproduire cela sur deux configurations de clusters réelles, que les utilisateurs pourraient rencontrer au cours de leur carrière, pour confirmer ou infirmer les conclusions obtenues.

Sur les deux clusters, il s'agit d'une politique NUMA locale par défaut. Il y a deux sockets AMD et deux sockets Intel. Chaque puce utilise sa mémoire pour ses propres cœurs (caches L1, L2 et L3). Par défaut, lorsqu'un processus s'exécute sur un cœur, l'ordonnanceur est susceptible de maintenir ce processus sur ce cœur pour éviter une perte de performances. Cela s'appelle aussi l'affinité. Le SMT et l'HT sont désactivés sur les nœuds correspondants au niveau du BIOS.

Pour analyser l'impact du SMT sur les performances dans nos clusters de calcul, nous avons sélectionné quatre types différents d'applications stochastiques. Notre modèle basé sur les agents pour la modélisation épidémiologique du Covid-19 écrit en C++ et avec un besoin important de RAM, un benchmark CPU pour la simulation physique au CERN écrit en Python, une simple simulation Monte-Carlo axée sur les calculs pour estimer Pi écrite en C et la bibliothèque de tests de PRNG TestU01 écrite en C. Chaque application en C ou C++ est compilée en utilisant gcc ou g++ (versions mentionnées ci-dessus), et l'option -O2.

Notre modèle est décrit dans les sections précédentes. Pour rappel, les agents modélisés sont des humains se déplaçant sur une carte. La carte est divisée en cellules et stockée dans une matrice carrée. Si certains humains sont contaminés, lorsqu'ils se déplacent, ils peuvent infecter les autres en utilisant un voisinage de Moore. Pour avoir une charge HPC réaliste pour nos tests, nous avons exécuté les simulations sur Paris. La carte mesure 10 000 m² et nécessite environ 2

Go de RAM par simulation. La charge correspondra à 256 Go de RAM pour 128 simulations parallèles sur des nœuds avec 512 Go de RAM. Ces simulations ne tiennent pas dans la mémoire cache et en raison de certains mouvements aléatoires à longue distance des humains sur la carte, chaque étape de la simulation peut inspecter le contenu des cellules dispersées partout sur la carte. En raison de cette contrainte de modélisation, il y aura de nombreux « cache miss ».

La simulation Monte-Carlo de calcul de Pi est un cas d'exemple bien connu dans le domaine des simulations stochastiques. Ce n'est pas du tout le moyen le plus rapide ou optimal pour obtenir une estimation de la valeur de Pi, mais cela sert de base et de benchmark facile pour l'estimation de surface (convergeant lentement à la racine carrée du nombre de points tirés). De plus, avec cette application simple, nous avons un exemple d'application axée sur les calculs où tout peut tenir dans la mémoire cache. De nombreuses applications réelles nécessitent la récupération des données et les accès mémoire, c'est toujours une préoccupation majeure malgré les améliorations de la DDR5 et l'augmentation de la taille du cache. L'accès à la mémoire peut souvent être la principale limitation pour le calcul intensif (Wulf et McKee 1995).

La troisième application que nous utilisons pour nos expériences est le benchmark DB12 développé au CERN (Boyer *et al.* 2022). Ce benchmark a pour objectif de répliquer le benchmark HEPSPEC06 utilisé pour les applications de physique des hautes énergies (HEP). Ce benchmark est petit et écrit en Python3. Il tente de modéliser la charge des simulations stochastiques exécutées sur la World Large Computing Grid (WLCG) pour le CERN et il est intéressant de vérifier le comportement d'un tel benchmark utilisé en HPC au CERN avec le SMT (le multithreading est presque toujours activé pour le calcul intensif en grille). Les résultats du benchmark lui-même montrent que les performances des cœurs diminuent à mesure que nous augmentons le parallélisme. Une fois que nous atteignons la charge maximale sur les machines avec SMT activé, nous remarquons une différence de performance entre les cœurs physiques et logiques, comme nous le montrerons après.

La dernière application est la bibliothèque de tests de PRNG TestU01. Pour savoir si un PRNG génère des nombres « aléatoires », nous utilisons des tests statistiques qui doivent déterminer s'il existe des corrélations visibles entre les nombres aléatoires générés. TestU01 est actuellement la batterie de tests la plus complète que nous connaissons pour évaluer la qualité des nombres aléatoires, en particulier avec sa célèbre batterie « BigCrush ». Pour notre expérience, nous avons sélectionné la batterie de tests SmallCrush car elle ne prend que 2 minutes, contre 4 heures pour BigCrush sur les processeurs modernes. Cela aurait été une perte de temps de calcul pour nos tests massivement parallèles.

Pour exécuter ces expériences, nous avons utilisé des scripts Bash pour les travaux Slurm. Par exemple, la figure 22 ci-dessous présente le script exécutant la charge de travail de 128 processus sur 4 cœurs, chacun exécutant 32 tâches séquentielles, avec affinité. Comme dit précédemment, nous avons utilisé l'option « --exclusive » pour obtenir le nœud entier réservé pour notre expérience. Nous avons utilisé « --ntasks » pour définir le nombre de coeurs que nous voulons utiliser, et « mem=0 » nous permet de réserver toute la RAM du nœud. Chaque tâche correspond à l'exécution d'une commande. Ici, la tâche est « time taskset –c $(($i%64)) ./exe [args] ». Dans le script donné, nous travaillons sur node-NoSMT, donc nous n'avons que des cœurs physiques (64). Avec taskset (affinité), nous définissons chaque tâche pour travailler sur son numéro de processeur (tâche 0 sur le cœur 0, tâche 1 sur le cœur 1, etc.). Lorsque nous atteignons 128 tâches avec seulement 64 cœurs, le « %64 » assigne deux tâches à chaque cœur. Avec les machines avec SMT activé, nous faisons attention à n'assigner que des cœurs physiques jusqu'à ce que le nombre de tâches dépasse le nombre de cœurs.

```
#!/bin/bash
#SBATCH –exclusive
#SBTACH –ntasks=4
#SBATCH –cpus-per-task=1
#SBATCH –nodelist=node-NoSMT
#SBATCH –mem=0
#SBATCH –output=4processAffinity-SMA-Experiement1-Node34
for I in ‘seq 0 3’;
do
   time taskset –c $(($i%64)) ./exe $((0 + 32*$i)) $((31 + 32*$i))
configParis configNoMesure logConfigParisProc$i &
done
wait
```

Figure 22 : Exemple de script Bash lancer des jobs avec Slurm

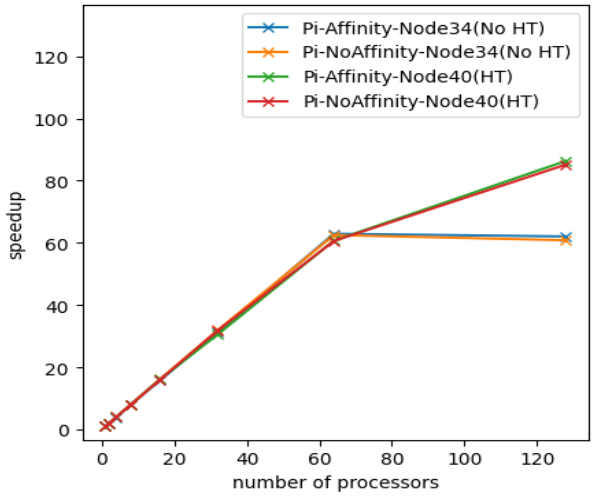
Nous avons adopté une approche de charge de travail massivement parallèle au niveau du système d'exploitation pour utiliser efficacement les nœuds multiprocesseurs. Cela signifie que nous avons une charge de travail qui peut être facilement divisée en tâches indépendantes, chacune pouvant être traitée en parallèle sans besoin de coordination complexe. Ainsi, nous n'avons pas besoin des bibliothèques parallèles classiques : OpenMP, MPI ou pthread avec des threads légers, ce qui nous permet de nous concentrer sur les performances du SMT et d'éviter les limitations potentielles de synchronisation qui pourraient survenir avec des méthodes de traitement parallèle plus sophistiquées. Nous voulons tirer le meilleur parti des nœuds

multiprocesseurs disponibles dans nos systèmes. Plutôt que de diviser chaque simulation en unités plus petites, nous les regroupons pour simplifier la distribution de la charge de travail. Les « bags of work » peuvent alors être facilement assignés à 128 ou 32 tâches parallèles (dans le cas de nos clusters AMD ou Intel, respectivement).

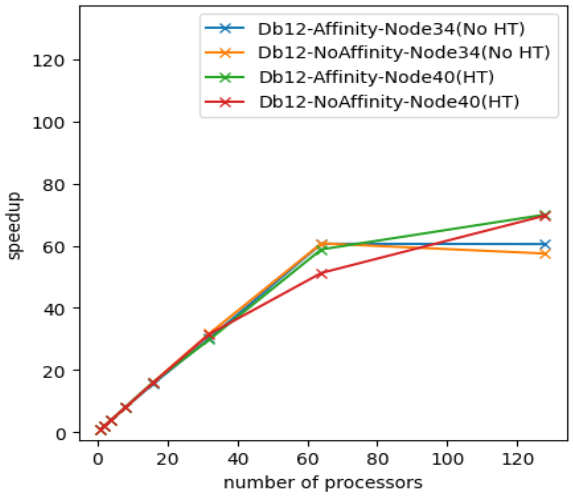
En 1967 (Amdahl 1967), Amdahl a défini l'accélération et ses limites en fonction de la proportion de code obligatoirement séquentiel. Dans notre expérience, la taille du problème à traiter est de 128 tâches et il n'y a pas de partie séquentielle, une accélération linéaire pourrait être attendue. Chaque tâche est un processus lourd suivant le modèle de parallélisation de haut niveau appelé SPMD (Single Program Multiple Data). Cette approche est intensivement utilisée sur les clusters pour les plans d'expériences et l'analyse de sensibilité où des charges parallèles énormes sont nécessaires pour explorer l'espace des résultats en fonction du nombre de valeurs possibles pour chaque facteur d'un plan d'expérience. Pour de telles charges, la meilleure proposition est d'assigner une tâche par cœur, nous pourrions donc espérer « diviser » le temps de calcul par 128 (AMD) ou 32 (Intel). Comme mentionné précédemment, nous n'avons pas de parties séquentielles dans nos tâches, nous sommes dans le cas optimal de la loi d'Amdahl où l'accélération est théoriquement linéaire. Avec un problème massivement parallèle de ce type, nous voulons voir si nous pouvons atteindre cet optimum. Dans cette étude, nous n'avons pas seulement mesuré le temps de calcul, mais nous avons également vérifié les résultats numériques. Des résultats identiques au bit près nous permettent de nous assurer que le même chemin d'exécution est suivi pour chaque processus, de sorte que les écarts de temps de calcul ne résultent pas d'un écart dans l'exécution. Le programme local correspondant est nommé « exe ». Si nous parallélisons la charge sur 4 processeurs par exemple, nous voulons exécuter 32 simulations séquentielles sur chacun des quatre cœurs : 0 – 31, 32 – 63, 64 – 95 et 96 – 127. C'est ce que nous faisons ici avec « $((0 + 32*$i)) $((31 + 32*$i)) » dans le script donné à la figure 22. À l'intérieur du code C++ de la simulation, nous initialisons le générateur Mersenne Twister avec un état dépendant du numéro de la simulation. Ainsi, chaque simulation sera initialisée avec le même état, qu'elle s'exécute séquentiellement ou en parallèle. Cela nous permet d'obtenir des résultats répétables entre l'exécution séquentielle et parallèle. L'évaluation des performances est basée sur deux métriques : le temps total d'exécution et l'accélération obtenue.

IV.6.2. Résultats

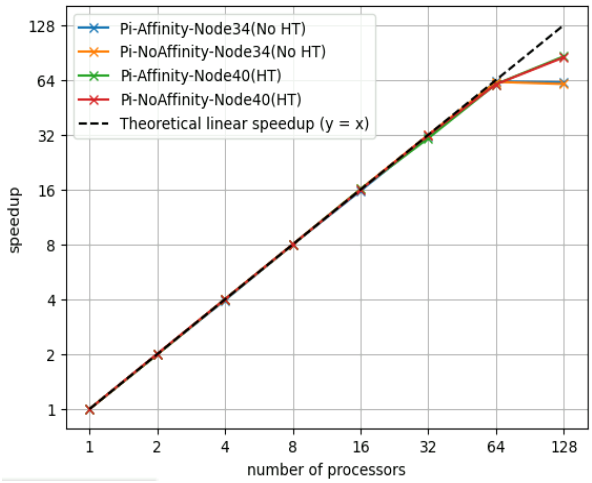
Nous avons effectué 30 réplications avec une parallélisation utilisant 1, 2, 4, 8, 16, 32, 64 et 128 cœurs sur AMD et 1, 2, 4, 8, 16 et 32 cœurs sur Intel. Nous ne comparons pas AMD et Intel; au contraire, nous validons nos résultats sur deux processeurs de marques et de générations différentes que nous avons à notre disposition pour cette étude. La figure 23a présente le résultat du petit programme intensif en calcul qui estime la valeur de PI. Nous pouvons déjà obtenir quelques informations. Premièrement, le SMT est efficace et nous donne une augmentation de performance de 30 % par rapport au programme s'exécutant sans SMT. À la fin (128 tâches entièrement parallélisées sur 128/64 cœurs), nous pouvons observer un léger bénéfice de l'affinité, avec ou sans SMT. Nous n'avons pas montré les barres d'erreur de l'intervalle de confiance à 95 % car l'écart est si faible qu'il n'apparaît pas sur les graphiques; les mesures sont statistiquement solides et la barre d'erreur n'est pas significative pour cette étude. Avec DB12, les conclusions que nous pouvons tirer de la figure 23b sont similaires à celles obtenues avec le programme d'estimation de PI. Ces applications sont similaires, avec une forte intensité de calcul, mais il y a un besoin croissant de mémoire pour DB12, même s'il reste faible par rapport à la RAM disponible par nœud et par cœur (512 Go de RAM pour 128 cœurs logiques).



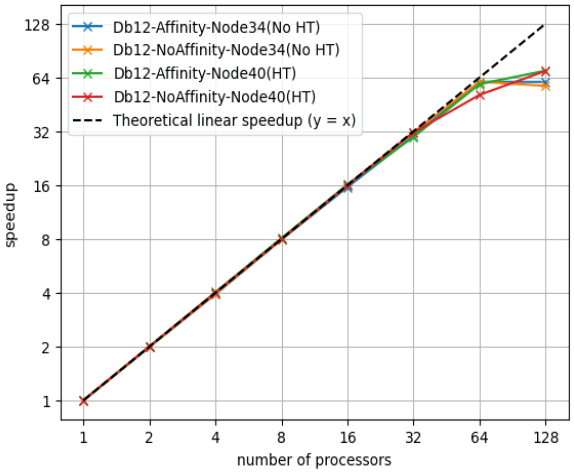
(A1)



(B1)



(A2)



(B2)

Figure 23: (A1 - A2) Accélération de la simulation de Monte Carlo pour estimer PI, en fonction du SMT et de l'affinités ; (B1 - B2) Accélération du benchmark DB12 en fonction du SMT et de l'affinité - sur AMD (128 coeurs)

Pour le SmallCrush de TestU01 (figure 24A), nous avons la même conclusion que pour Db12 et Pi, où le SMT est plus efficace avec 128 processus parallèles. Dans la figure 27B, pour la simulation multi-agent, la situation est différente. Tout d'abord, nous pouvons voir que, bien que l'accélération approchait la linéarité pour PI, DB12 et SmallCrush, ce n'est pas le cas ici. Pire encore, dans la figure 24B, nous voyons que si nous parallélisons avec plus de 32 cœurs, les performances sans affinité commencent à diminuer. Nous pouvons également noter que la meilleure accélération obtenue est sans affinité et avec un faible niveau de parallélisation (32 cœurs). Cela signifie pour nous que laisser le système d'exploitation décider de l'affinité pourrait être une meilleure option, si nous ne définissons pas l'affinité en fonction de l'architecture matérielle exacte (ce qui est souvent impossible pour les utilisateurs non experts). Nous pouvons également observer un phénomène curieux : après 32 cœurs, sans affinité, il y a une diminution de l'accélération jusqu'à 64 cœurs, avant que les performances n'atteignent un plateau constant proche d'une accélération de 16 (loin d'être linéaire). Nous pouvons également remarquer que, bien que l'affinité soit la principale différence de performance, le nœud sans

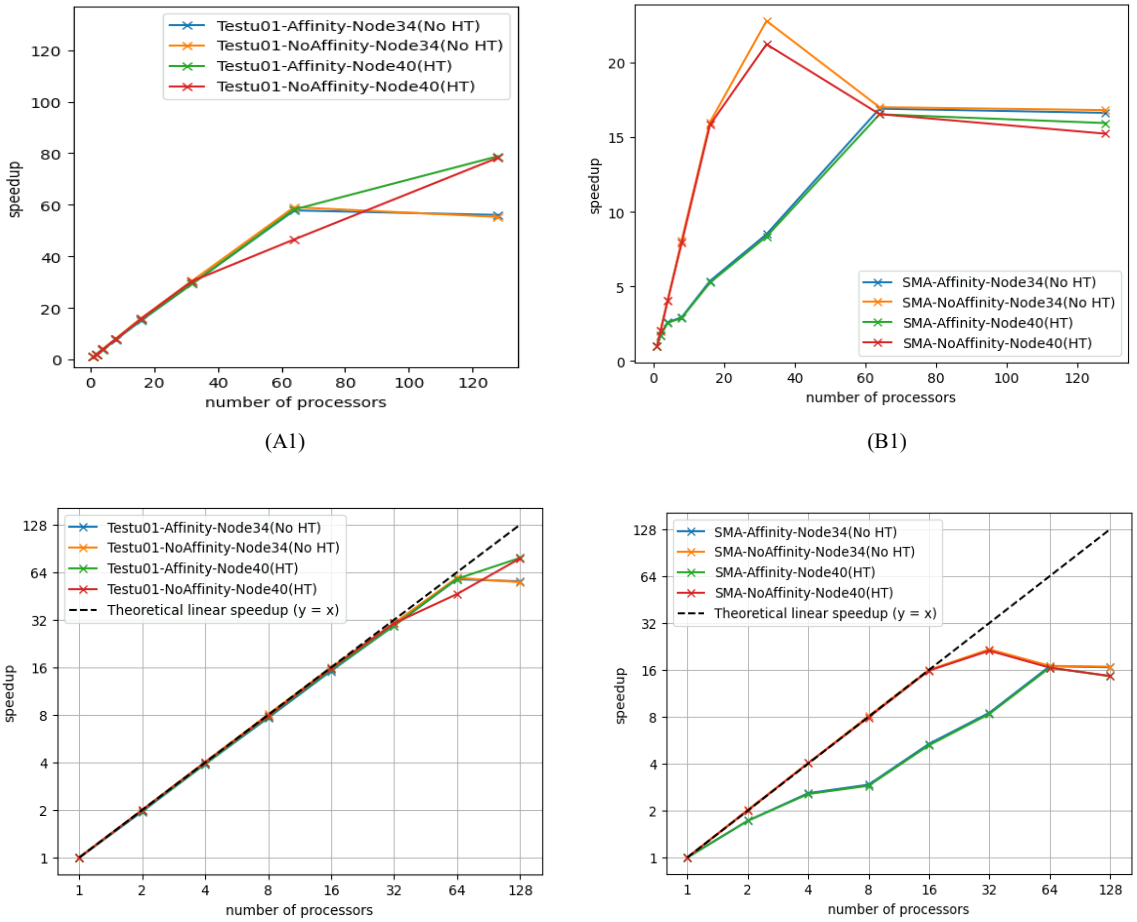
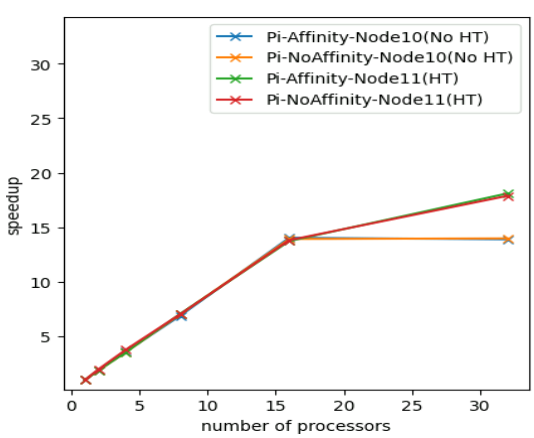


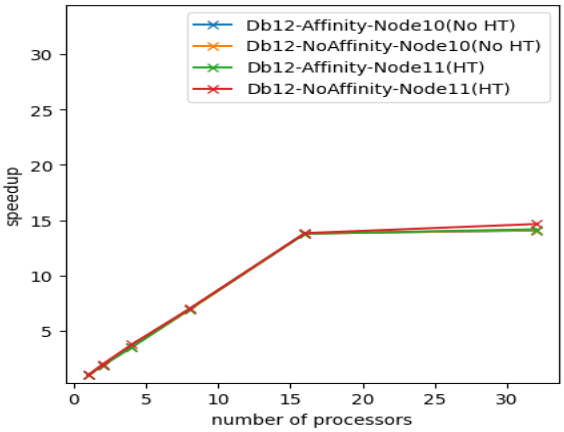
Figure 24: (A1 - A2) Accélération de TestU01 en fonction du SMT et de l'affinité; (B1 - B2) Accélaration du SMA en fonction du SMT et de l'affinité - sur AMD (128 coeurs)

SMT fonctionne légèrement mieux que le nœud avec SMT. Nous discuterons plus tard des hypothèses de ces résultats.

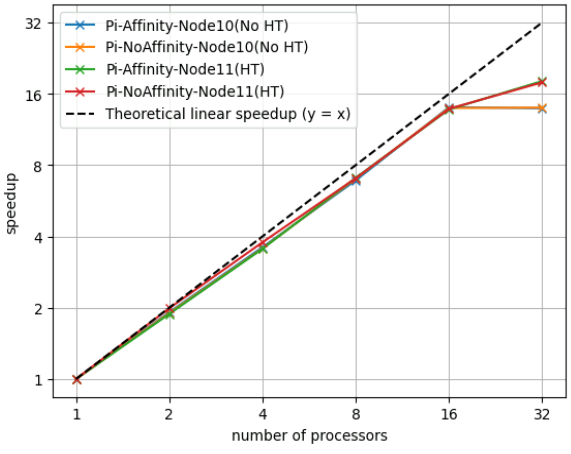
Nous allons maintenant présenter les résultats de la même expérience réalisée sur une puce Intel (hyper-threading). D'après les figures 25 et 26, nous pouvons voir que les résultats obtenus sur le nœud Intel sont similaires à ceux obtenus avec le processeur AMD plus « moderne ». Pour les applications intensives en calcul comme la simulation Monte Carlo de Pi, le benchmark Db12 ou SmallCrush de TestU01, l'hyper-threading est environ 30 % plus efficace (étonnamment, pas beaucoup pour Db12 ici). Pour le modèle épidémiologique, nous pouvons voir que sans hyper-threading, les performances sont légèrement meilleures. Ces résultats pour le cluster Intel confirment ce que nous avons obtenu sur les clusters AMD, avec des puces multicœurs plus grandes.



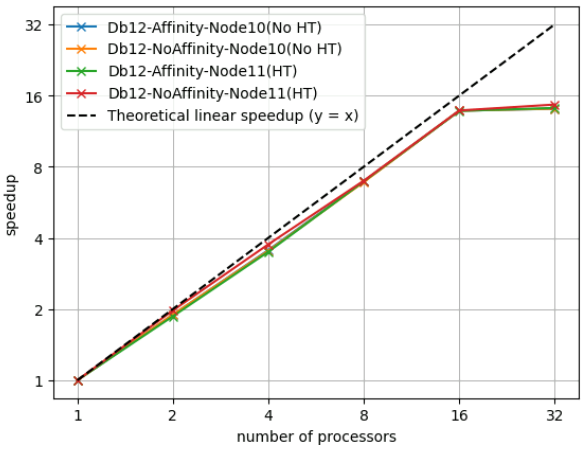
(A1)



(B1)



(A2)



(B2)

Figure 25: (A1 - A2) Accélération de la simulation de Monte Carlo pour estimer PI en fonction du SMT et de l'affinité ; (B1 - B2) Accélération de DB12 en fonction du SMT et de l'affinité - sur Intel (32 coeurs)

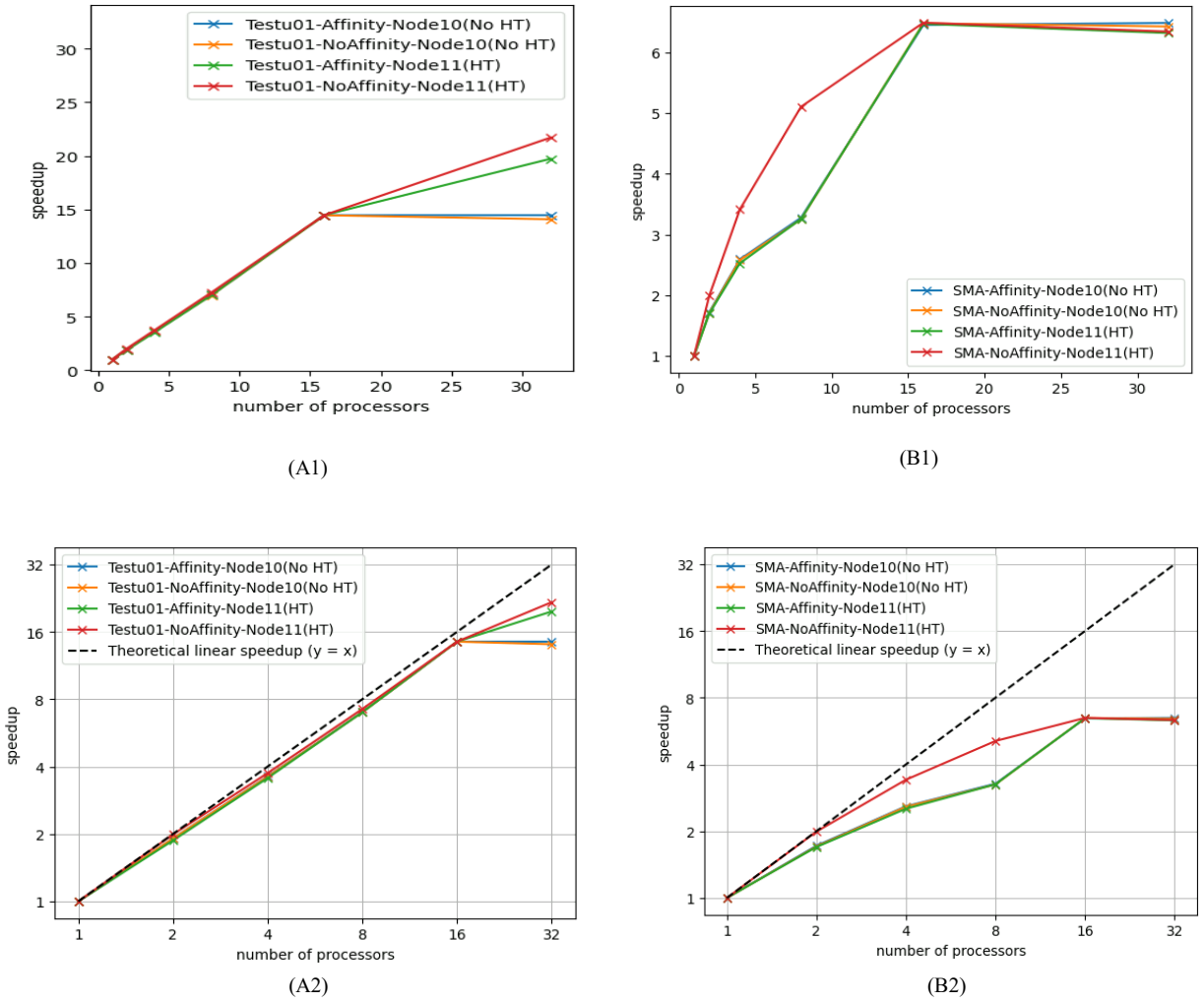


Figure 26: (A1 - A2) Accélération de TestU01en fonction du SMT et de l'affinité ; (B1 - B2) Accélération du SMA en fonction du SMT et de l'affinité - sur Intel (32 coeurs)

Avec les temps d'exécution donnés par les tables 20 et 21, nous pouvons voir avec les performances obtenues que l'utilisation de l'hyper-threading a eu un impact négatif pour l'application SMA, qui est liée à la mémoire. Pour les applications intensives en calcul, l'hyper-threading a eu un impact positif. En fait, sur AMD, nous pouvons voir qu'en termes de temps, le SMA prend environ 12 % de temps en plus avec SMT activé. Mais pour SmallCrush, il prend environ 27 % de temps en moins. Sur Intel, les résultats sont similaires, mais l'hyper-threading perd moins de performance pour le SMA, probablement en raison d'une surcharge moindre avec seulement 32 processus (comparé à 128). Cependant, les conclusions restent les mêmes : le SMT pourrait diminuer les performances dans le cas de simulations utilisant une grande quantité de mémoire.

	Temps SMA (minutes)	Temps TestU01 (minutes)	% augmentation temps (SMT)		Accélération SMA (speedup)	Accélération TestU01 (speedup)	% augmentation accélération	
			SMA	TestU01			SMA	TestU01
AMD Node-NoSMT Affi	248.42	0.2301	11.8%	-28.99%	16.62	56.06	-12.39%	40,62%
AMD Node-SMT Affi	277.74	0.1634			14.56	78.83		
AMD NoSMT NoAffi	248.27	0.221	12,02%	-25.34%	16.79	55.29	-12.03%	39.6%
AMD Node-SMT NoAffi	278.11	0.1650			14.62	78.26		

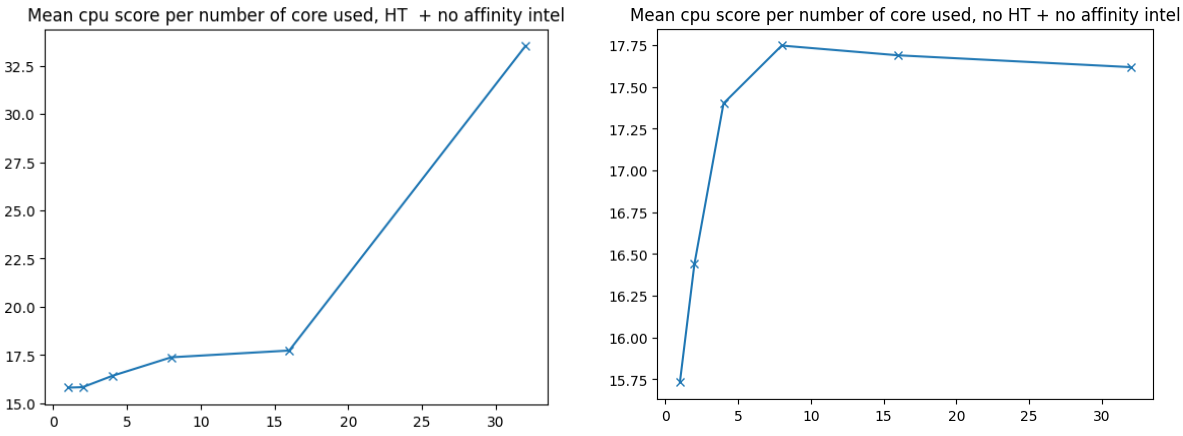
Table 20: Temps moyen et accélération pour 128 coeurs (SMT AMD) et pourcentage d'amélioration ou de dégradation des performances, pour le SMA et TestU01, en comparant avec le SMT acitvé ou désactivé

	Temps SMA (minutes)	Temps TestU01 (minutes)	% augmentation temps (SMT)		Accélération SMA (speedup)	Accélération TestU01 (speedup)	% augmentation accélération	
			SMA	TestU01			ABM	TestU01
Intel Node-NoHT Affi	178.3	0.25	2.05%	-26,8%	6.49	14.46	-2,62%	36.45%
Intel Node-HT Affi	181.95	0.183			6.32	19.73		
Intel Node-NoHT NoAffi	178.9	0.257	1.84%	-35,02%	6.43	14.08	-1.4%	54.12%
Intel Node-HT NoAffi	182.2	0.167			6.34	21.7		

Table 21: Temps moyen et accélération pour 32 coeurs (HT Intel) et pourcentage d'amélioration ou de dégradation des performances, pour le SMA et TestU01, en comparant avec le HT activé et désactivé

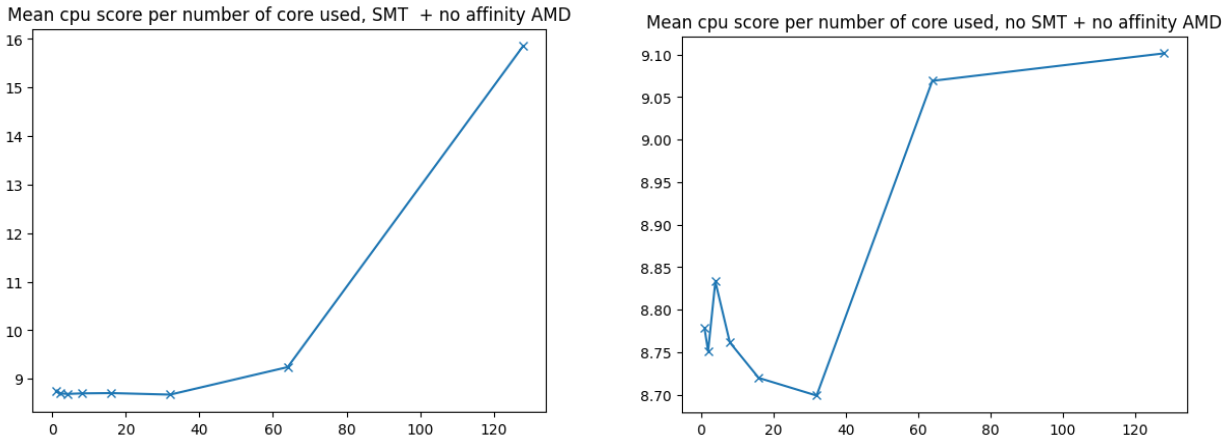
Dans nos résultats précédents, nous avons calculé le temps d'exécution moyen pour l'ensemble de l'exécution en utilisant tous les cœurs disponibles (128 et 32). Le temps de référence pour calculer l'accélération est le temps d'exécution de cette charge de travail sur un seul cœur. Nous avons effectué 30 réplications, chacune avec des intervalles de confiance de 95 %. Comme indiqué précédemment, les intervalles étaient trop petits pour être représentés visuellement sur les graphiques, ce qui indique que le temps de calcul présentait une faible variabilité dans notre cas. Cela est notable, car de nombreuses études se concentrent sur la variabilité introduite par l'utilisation du SMT, en soulignant l'existence.

Concernant les résultats du benchmark DB12 lui-même, ils montrent que la performance des cœurs diminue à mesure que nous augmentons le parallélisme. Une fois que nous atteignons la charge maximale sur une machine avec SMT activé, nous avons remarqué une différence de performance entre les cœurs physiques et logiques. Plus le score est élevé, moins le processeur est performant. Nous pouvons clairement voir sur les figures 27 et 28 que sans SMT activé, sur AMD ou Intel, la performance du CPU diminue légèrement à mesure que nous augmentons le parallélisme. Pendant ce temps, avec SMT activé, nous voyons clairement le point de déclenchement du calcul sur les cœurs logiques au lieu des cœurs physiques (à 16 pour Intel et à 64 pour AMD). Pour le benchmark DB12, un cœur logique est simplement deux fois moins performant qu'un cœur physique.



(A) (B)

Figure 27: Score du benchmark DB12 sur processeur Intel avec HT (A) et sans HT (B)



(A) (B)

Figure 28: Score du benchmark DB12 sur processeur AMD avec SMT activé (A) et SMT désactivé (B)

Deuxièmement, pour parler de la reproductibilité des mesures de DB12, nous avons vérifié la moyenne et l'écart-type sur 30 réplications. Nous avons ensuite observé combien de différences nous pouvons observer d'une exécution à l'autre dans exactement la même configuration. Cela peut bien sûr se produire avec de nombreux benchmarks CPU où nous ne nous attendons pas à obtenir exactement la même mesure. Étant donné que le temps est mesuré, les résultats les plus bas sont meilleurs. Nous pouvons remarquer que le processeur AMD récent est significativement plus rapide que l'Intel plus ancien. De plus, nous constatons que la variance est beaucoup plus faible avec la puce AMD. Cela est montré dans les figures 29 et 30 pour les processeurs AMD et Intel respectivement.

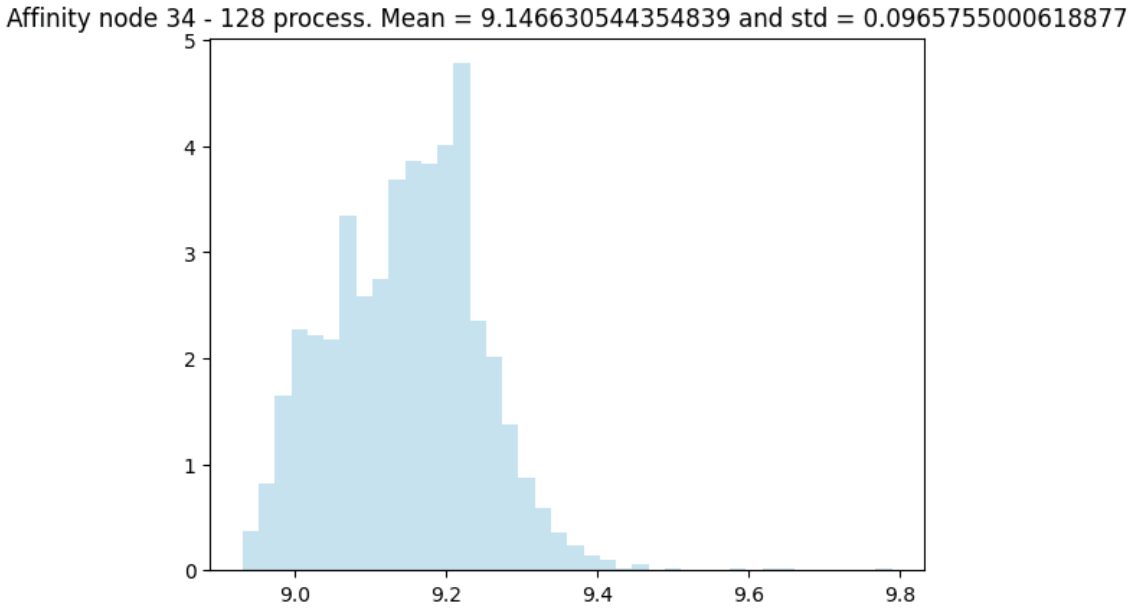


Figure 29: Score du benchmark DB12 sur les 30 réplications lors de l'utilisation de 128 coeurs, sans SMT

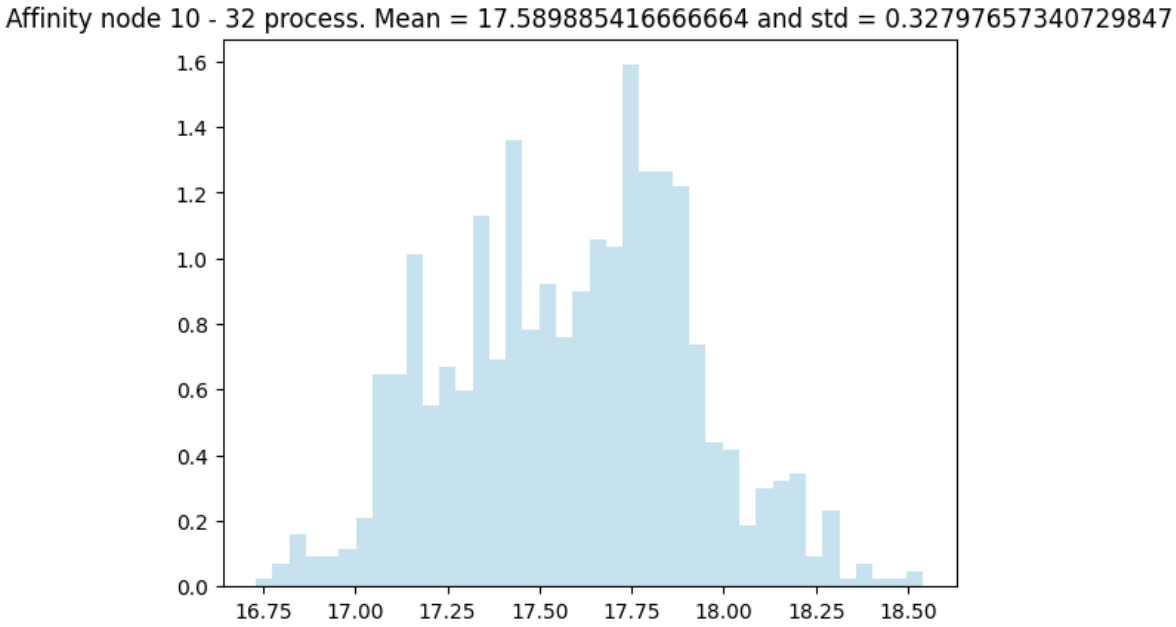


Figure 30: Score du benchmark DB12 sur les 30 réplications lors de l'utilisation de 32 coeurs, sans HT

IV.6.3. Discussion

D'après les résultats obtenus, nous pouvons émettre certaines hypothèses. Tout d'abord, sur AMD avec 128 processus, nous constatons que le réglage de l'affinité avec taskset n'est pas toujours efficace, en particulier lorsque les nœuds exécutent des charges lourdes sur tous les cœurs. Peut-être que l'optimisation des résultats en prenant en compte les nœuds NUMA et l'architecture matérielle du CPU lors du réglage de l'affinité pourrait donner de meilleures performances. Cependant, il est important de noter que les utilisateurs non experts sont peu susceptibles de posséder les connaissances ou la capacité de mettre en œuvre de telles optimisations. Nous observons la perte de temps lors de l'exécution de l'application SMA, qui est liée à la mémoire. Sans définir l'affinité des processus, nous avons obtenu une meilleure accélération lorsque 1 à 32 cœurs sont utilisés pour la parallélisation. Ensuite, l'accélération sans affinité diminue de 32 à 64 cœurs. Lorsque nous avons pris en main l'affinité, nous avons pu obtenir une augmentation constante de l'accélération au même rythme, avec ou sans SMT jusqu'à 64 cœurs. Les performances sans SMT sont légèrement meilleures pour ce type d'application SMA utilisant beaucoup de mémoire. Au-delà de 64 cœurs, nous atteignons approximativement le même plateau. À pleine charge, le SMT a augmenté le temps global d'environ 10 %. Cependant, la meilleure accélération a été obtenue avec seulement 32 cœurs et sans affinité. Nous avons utilisé le logiciel hwloc de l'INRIA (Broquedis *et al.* 2010) pour constater que, sur le cœur AMD, le cache L1 et L2 est dupliqué pour chaque cœur physique (respectivement 64KB et 512KB), donc partagé entre les cœurs logiques, et le cache L3 (16 MB) est partagé entre quatre cœurs physiques.

Deuxièmement, et surtout, nous pouvons voir que pour les applications intensives en calcul comme Pi, Db12 ou TestU01, nous avons obtenu une augmentation des performances d'environ 30 % avec le SMT. Cela peut être attendu selon la technologie de l'hyper-threading, comme montré dans (Marr *et al.* 2002). Cependant, pour les applications HPC nécessitant une grande quantité de mémoire, comme le SMA que nous avons utilisé, le SMT est moins efficace. La principale différence de calcul entre le SMA et les petits programmes comme Pi, Db12 ou TestU01 est probablement liée au goulot d'étranglement de la mémoire et aux « cache miss ». La raison pour laquelle le SMT peut entraîner une diminution des performances pour les applications gourmandes en mémoire pourrait être que les ressources d'exécution sont partagées entre les deux threads d'un cœur. Cela signifie que la taille du cache est réduite de moitié, ce qui peut entraîner davantage de cache miss (sauf pour le cas du cache L1 spécifique aux cœurs logiques, mais il est de très petite taille). Dans notre hypothèse, nous nous rapprochons des

conclusions de l'article de la NASA (Saini *et al.* 2011) qui traite d'une application de mécanique des fluides. Sur l'une de leurs applications, ils n'ont pas observé de gain de performance avec le SMT parce que « le SMT augmente la concurrence pour les ressources dans la hiérarchie de la mémoire, comme la bande passante mémoire ». De plus, les performances du SMT sont affectées par la pression de communication accrue à mesure que des processus supplémentaires se disputent les ressources du réseau telles que les puces HCA (Host Channel Adapter) et les commutateurs IB (InfiniBand). Par conséquent, nous devons prendre en compte les avantages et les inconvénients deu SMT. Nous convenons que la concurrence pour le cache et pour la mémoire en général se traduit par davantage de cache miss et plus de saturation du bus pour les applications HPC réalistes liées à la mémoire, et non pour les petits cas d'exemple. Cela remet en question l'utilité du SMT sur les clusters de calcul, car nous pouvons exécuter de larges programmes qui généreront des conflits de mémoire.

Enfin, en ce qui concerne l'accélération, nous pouvons voir que les applications pratiques ne répondent pas vraiment aux attentes théoriques. En fait, pour les petits programmes, nous pouvons voir que l'accélération augmente de manière presque linéaire (environ 20/30 % plus lente que linéaire avec le SMT et 50 % sans). Et avec le SMT activé, nous pouvons voir un réel avantage puisque les processeurs SMT sont capables d'approcher la loi d'Amdahl comme si leurs cœurs logiques étaient des cœurs physiques. C'est un bon point pour les processeurs SMT. Sans SMT, nous pouvons voir la stagnation attendue de l'accélération. Néanmoins, pour l'application SMA, la courbe d'accélération ne ressemble pas du tout à ce que nous attendions. Tout d'abord, nous voyons un impact négatif de l'affinité qui peut être surprenant. De plus, nous pouvons voir que l'accélération n'est pas seulement sublinéaire, mais diminue clairement après 32 cœurs. Cette différence entre l'attente théorique et la valeur expérimentale est, à notre avis, due au fait que la loi d'Amdahl ne prend en compte que la puissance de calcul. Nous devrions également considérer l'accès à la mémoire et son ratio avec la puissance de calcul. Le résultat est qu'augmenter les ressources de calcul sans prendre en compte l'interconnexion, la vitesse de la mémoire, les caches, etc., peut conduire à une véritable situation de perte de ressource. Nous perdons du temps de calcul et nous perdons de la consommation d'énergie. Comme décrit dans (Szalay *et al.* 2010), nous pouvons voir que pour optimiser le rapport consommation d'énergie/puissance de calcul, nous devons être très prudents quant à l'association de plus de puissance de calcul à un accès mémoire plus rapide. Et nous pouvons voir ici, avec le SMA, qu'augmenter les ressources de calcul est non seulement un gaspillage d'énergie, mais aussi une perte de temps (et plus de temps équivaut à plus de gaspillage d'énergie). Cela montre les limites

de la loi d'Amdahl pour estimer l'accélération pour les programmes HPC réels en fonction du nombre de cœurs lorsqu'il y a besoin d'une mémoire assez grande par cœur. Notre étude montre les limites de l'accélération pour les problèmes liés à la mémoire avec et sans affinité, avec et sans hyper-threading simultané. Le biais que nous pourrions avoir est que le cluster que nous avons utilisé n'est pas aussi optimisé qu'un superordinateur. Nous devons également garder à l'esprit que les processeurs que nous avons testés ici sont construits et optimisés pour le SMT. Désactiver le SMT sur ces cœurs pourrait ne pas conduire à des performances optimales. Avec des cœurs à un seul thread, les résultats auraient pu être plus significatifs pour les simulations importantes. En fait, le SMT a un coût lors de la fabrication des puces, et nous voyons maintenant que la tendance est de multiplier le nombre de cœurs tout en diminuant leur vitesse d'horloge. En faisant cela, nous mettons plus de pression sur d'autres composants comme la mémoire, le bus ou l'interconnexion, qui deviennent le facteur limitant. Nous montrons que cela peut affecter les performances lorsque nous exécutons des simulations liées à la mémoire.

IV.7. L'utilisation de bibliothèque scientifique transformation de fourrier, impact sur les performances et la reproductibilité

IV.7.1. Expérience mise en place

Notre recherche a été menée sur un serveur Intel fonctionnant sous la distribution Linux (Debian 6.1.76-1, kernel 6.1.0-18-amd64). Le serveur est équipé d'un processeur Intel Xeon Platinum 8160 à 2,10 GHz, doté de 192 cœurs répartis sur 4 sockets, chaque socket hébergeant 24 cœurs avec 2 threads par cœur. Le cache comprend 3 MiB L1d et L1i (96 instances chacun), 96 MiB L2 (96 instances) et 132 MiB de cache L3 (4 instances). Cette étude a été publié dans l'article (Antunes *et al* 2024).

Dans note étude, nous avons comparé plusieurs compilateurs et bibliothèques pour tester les performances, la consommation d'énergie et la reproductibilité numérique. Les compilateurs utilisés sont GNU Fortran (version 12.2.0), Intel Fortran Compiler Classic (IFORT, version 2021.11.1), et le nouveau compilateur Intel Fortran basé sur LLVM (IFX, version 2024.0.2). LLVM, initialement un acronyme pour Low Level Virtual Machine, a évolué au-delà de son objectif initial et sert désormais de cadre de compilation. Il prend en charge plusieurs langages de programmation et plateformes. Les calculs mathématiques ont utilisé des bibliothèques telles que FFTW version 3.3.10 et Math Kernel Library (MKL) version 2024.1.0, incluses dans le oneAPI Base Toolkit. OpenMP version 4.5 a été utilisé pour la gestion des threads.

Le code QDD a été compilé en utilisant diverses combinaisons de compilateurs, bibliothèques et paramètres pour étudier différents indicateurs de performance. Le makefile associé à QDD nous a permis de basculer entre les compilateurs (gfortran, ifort, ifx), les bibliothèques FFT (FFTW, MKL), les options de threading (OpenMP activé/désactivé, OpenMP dynamique) et les options de débogage (activé/désactivé). L'option « DYN » (DYNOMP dans le makefile) détermine si la parallélisation OpenMP est appliquée aux fonctions d'onde à particule unique (DYNOMP = YES) ou aux opérations FFT utilisant la bibliothèque FFTW3 (DYNOMP = NO).

Les options de compilation fournies par QDD ont été spécifiées pour optimiser les performances pour différents environnements. Pour le compilateur GNU Fortran (gfortran), les paramètres de compilation standard incluent des indicateurs d'optimisation tels que -Ofast pour une optimisation maximale, -mfpmath=sse et -msse4.2 pour spécifier l'utilisation des instructions SSE4.2, et -fdefault-real-8 et -fdefault-double-8 pour imposer la double précision. Le traitement parallèle est activé avec l'option -fopenmp. Ces paramètres sont complétés par les bibliothèques FFTW ou MKL. Pour les besoins de débogage, les indicateurs -pg pour le profilage, -g pour générer des informations de débogage, et -fbacktrace pour la journalisation de la pile sont utilisés en complément de -w pour supprimer les avertissements. De même, pour les compilateurs Intel, la configuration inclut -fpp pour le prétraitement, -w pour supprimer les avertissements, et -xsse4.2 et -Ofast pour les instructions SIMD optimisées et la vitesse maximale. Des indicateurs supplémentaires tels que -ip pour l'optimisation interprocédurale, -no-prec-div pour désactiver la division précise, et -align all pour l'alignement des données sont spécifiés. L'indicateur spécifique à Intel -qopenmp est utilisé pour les directives OpenMP, et des options pour FFTW ou MKL sont également disponibles. Les configurations de débogage intègrent -pg pour le profilage, -g pour les informations de débogage, -CB pour la vérification des limites de tableau, -traceback pour des informations de trace améliorées, et -align all et -autodouble pour l'alignement et l'imposition de la double précision, respectivement.

Nous avons compilé QDD avec toutes les combinaisons d'options de configuration disponibles. Cela a résulté en trente-six conditions expérimentales distinctes, chacune correspondant à une combinaison unique de paramètres de compilation, de modes de débogage, d'utilisation d'OpenMP et de bibliothèques de transformées de Fourier. Pour chaque configuration, un répertoire correspondant a été établi, nommé systématiquement pour refléter les paramètres de compilation spécifiques (par exemple, gfortran-FFTW-noOMP-noDebug, ou ifort-MKL-OMP-noDebug).

Chaque répertoire contient l'exécutable compilé et est subdivisé en cinquante sous-répertoires, un pour chaque réplication. Grâce aux cinquante réplications, nous pouvons évaluer le temps et la consommation énergétique, et nous pouvons évaluer la répétabilité des résultats sur plusieurs exécutions (dans la même configuration expérimentale exacte), et grâce aux différents dossiers d'options de compilation, nous pouvons évaluer la reproductibilité entre différentes configurations (avec des différences dans la configuration expérimentale, par exemple, les options de compilation). Les principaux indicateurs évalués étaient les performances en termes de temps, la consommation d'énergie et la consistance des données de sortie.

Pour automatiser le processus de compilation et d'exécution, des scripts Bash ont été développés. Ces scripts ajustaient dynamiquement les paramètres du Makefile en fonction des valeurs prédéfinis de types de compilateurs, paramètres de débogage, configurations OpenMP et bibliothèques FFT. Plus précisément, la commande sed a été utilisée pour modifier le Makefile afin de refléter les options de compilation souhaitées, garantissant que chaque exécutable était configuré correctement avant d'être placé dans son répertoire désigné. Un autre script Bash a facilité l'exécution de ces expériences. Il utilisait des variables d'environnement pour configurer les paramètres du système, naviguait dans la structure des répertoires et exécutait chaque QDD compilé dans chaque sous-répertoire. Les temps d'exécution de chaque réplication sont enregistrés. Un script supplémentaire est dédié à la mesure de la consommation d'énergie. Cela implique de parcourir les répertoires correspondant à chaque réplication, où l'exécutable est exécuté en même temps que l'outil PowerJoular pour surveiller la consommation d'énergie. Le script gère des répertoires temporaires pour isoler chaque test, garantissant que les mesures d'énergie ne modifient pas les résultats numériques des expériences temporelles. Une fois tous les résultats obtenus dans chaque dossier, nous utilisons un Jupyter Notebook pour compiler et analyser tous les résultats.

Pour mesurer la consommation d'énergie des programmes, nous utilisons l'outil PowerJoular (Noureddine 2022). Nous avons dû modifier le code de l'outil PowerJoular pour prendre en compte la consommation d'énergie en multithreading. À l'origine, PowerJoular était conçu pour surveiller la consommation de ressources des processus en se basant uniquement sur l'ID de processus principal (PID), en extrayant les données de /proc/PID/stat. Cette approche ne tenait pas compte de l'utilisation supplémentaire des ressources des multiples threads au sein d'un processus, le multithreading léger étant courants dans les applications modernes pour améliorer les performances, et spécifiquement dans notre cas de parallélisation OpenMP pour

la simulation de physique quantique. Pour surmonter cette limitation, nous avons amélioré PowerJoular pour inclure la surveillance de tous les threads associés à un processus. Cela a été réalisé en ajoutant une fonctionnalité pour lire à partir de /proc/PID/task, un répertoire contenant des sous-répertoires pour chaque ID de thread (TID) lié au PID. PowerJoular parcourt maintenant ces sous-répertoires, collectant des données à partir de chaque fichier stat de TID et agrégeant ces informations pour capturer plus précisément la consommation totale de ressources. Notre modification est présente sur Github : https://github.com/joular/powerjoular/pull/36.

Le code de cet article est disponible en ligne à l'adresse : https://gitlab.isima.fr/beantunes/qdd-reproducibility

IV.7.2. Résultats

Toutes les données de temps et d'énergie ci-après sont considérées avec les options de compilation originale pour la performance donnée avec QDD que nous avons présentée ci-dessus (-Ofast, -sse4.2, etc.). Nous discuterons dans la section sur la reproductibilité numérique de certaines modifications que nous avons apportées à ces options et de la manière dont elles ont influencé les résultats de la reproductibilité numérique et des performances. Comme mentionné précédemment, chaque expérience a été répliquée cinquante fois. Nous avons utilisé des tests statistiques, tels que le test de Levene et le test T de Student, pour établir l'équivalence des variances et des moyennes, respectivement, en se basant sur une valeur p de 0,005 (niveau de confiance de 99,5 %).

Le test de Levene est utilisé pour évaluer l'égalité des variances pour une variable calculée pour deux groupes ou plus. L'hypothèse nulle du test de Levene est que les variances sont égales entre les groupes. Si la valeur p est inférieure à 0,005, nous rejetons l'hypothèse nulle, indiquant que les variances ne sont pas égales. Lorsque les variances se sont avérées inégales, nous avons utilisé le test T de Welch au lieu du test T de Student standard.

Le test T de Student, qui suppose des variances égales, est utilisé pour déterminer s'il existe une différence significative entre les moyennes de deux groupes. L'hypothèse nulle du test T de Student est que les moyennes des deux groupes sont égales. Une valeur p inférieure à 0,005 conduit au rejet de cette hypothèse nulle, suggérant une différence significative entre les moyennes des groupes.

Notre objectif est de mesurer l'influence du compilateur (gfortran, ifort et ifx) et de la bibliothèque FFT (FFTW et MKL) sur le temps et la consommation d'énergie. Pour y parvenir, nous isolons le paramètre dont nous souhaitons mesurer l'influence, en veillant à ce que toutes les autres variables restent constantes.

Dans la table 22, nous comparons la moyenne et la variance de chaque expérience, en ne variant que gfortran, ifort et ifx. En vert, nous avons les valeurs moyennes qui sont statistiquement différentes les unes des autres (avec un niveau de confiance de 99,5 %). Par exemple, l'expérience avec les options FFTW-OMP-Dyn-Debug, en utilisant gfortran, a pris en moyenne 702 secondes pour s'exécuter, tandis qu'elle a pris 1094 secondes et 1461 secondes en utilisant respectivement ifort et ifx. Comme les valeurs sont écrites en vert, cela suppose que ces différences sont statistiquement significatives, donc nous pouvons conclure que gfortran conduit à de meilleures performances dans ce cas.

Expérience	**Temps réel moyen (s) – gfortran**	**Std temps réel (s) – gfortran**	**Temps réel moyen (s) – ifort**	**Std temps réel (s) – ifort**	**Temps réel moyen (s) – ifx**	**Std temps réel (s) – ifx**
(gfortran/ifort/ifx)-FFTW-OMP-DYN-Debug	702,00	7,53	1094,88	4,84	1461,76	5,24
(gfortran/ifort/ifx)-FFTW-OMP-DYN-noDebug	365,28	8,26	413,82	6,12	401,54	6,11
(gfortran/ifort/ifx)-FFTW-OMP-Debug	764,80	12,37	1258,16	8,66	1603,26	9,86
(gfortran/ifort/ifx)-FFTW-OMP-noDebug	404,22	12,56	452,64	9,67	443,18	6,73
(gfortran/ifort/ifx)-FFTW-noOMP-Debug	2685,84	58,56	3762,24	22,23	4808,76	30,53
(gfortran/ifort/ifx)-FFTW-noOMP-noDebug	1416,10	14,49	1337,78	11,95	1341,68	13,61
(gfortran/ifort/ifx)-MKL-OMP-DYN-Debug	637,82	6,94	1039,08	4,62	1365,10	7,19
(gfortran/ifort/ifx)-MKL-OMP-DYN-noDebug	296,10	7,05	313,92	7,00	329,92	6,10
(gfortran/ifort/ifx)-MKL-OMP-Debug	747,92	11,61	1236,00	8,43	1559,24	8,95

(gfortran/ifort/ifx)-MKL-OMP-noDebug	388,84	8,49	393,34	9,41	415,96	8,74
(gfortran/ifort/ifx)-MKL-noOMP-Debug	2238,18	5,96	3349,02	5,56	4386,84	10,24
(gfortran/ifort/ifx)-MKL-noOMP-noDebug	986,60	5,21	912,02	6,88	927,08	6,37
Moyenne globale	969,48	19,24	1296,91	9,89	1587,03	11,96

Table 22: Influence de gfortran, ifort et ifx sur les temps d'éxecution réel. En vert, les valeurs moyennes pour lesquelles les différences sont statistiquement significatives. En bleu, quand elles ne le sont pas

Comme observé, gfortran présente des avantages non-négligeable en termes de temps de calcul pour les expériences. Étant donné que les expériences ont été menées sur un processeur Intel, il était initialement prévu que les compilateurs Intel puissent offrir un avantage concurrentiel. Cependant, cela ne semble pas être le cas.

Une autre observation notable est que le nouveau compilateur Intel Fortran, ifx, affiche des performances plus lentes par rapport à ifort. Alors qu'ifort est 34 % plus lent en temps réel par rapport à gfortran (en considérant la moyenne globale), ifx accuse un retard de 64 %, indiquant un avantage significatif en termes d'efficacité pour gfortran.

Dans la table 23, nous explorons la même expérience, mais nous concentrons notre attention sur les performances de FFTW par rapport à MKL.

Expérience	**Temps réel moyen (s) – FFTW**	**Std temps réel (s) – FFTW**	**Temps réel moyen (s) – MKL**	**Std temps réel (s) – MKL**
gfortran-(FFTW/MKL)-OMP-DYN-Debug	702,00	7,53	637,82	6,94
gfortran-(FFTW/MKL)-OMP-DYN-noDebug	365,28	8,26	296,10	7,05
gfortran-(FFTW/MKL)-OMP-Debug	764,80	12,37	747,92	11,61
gfortran-(FFTW/MKL)-OMP-noDebug	404,22	12,56	388,84	8,49
gfortran-(FFTW/MKL)-noOMP-Debug	2685,84	58,56	2238,18	5,96
gfortran-(FFTW/MKL)-noOMP-noDebug	1416,10	14,49	986,60	5,21
ifort-(FFTW/MKL)-OMP-DYN-Debug	1094,88	4,84	1039,08	4,62

ifort-(FFTW/MKL)-OMP-DYN-noDebug	413,82	6,12	313,92	7,00
ifort-(FFTW/MKL)--OMP-Debug	1258,16	8,66	1236,00	8,43
ifort-(FFTW/MKL)--OMP-noDebug	452,64	9,67	393,34	9,41
ifort-(FFTW/MKL)-noOMP-Debug	3762,24	22,23	3349,02	5,56
ifort-(FFTW/MKL)--noOMP-noDebug	1337,78	11,95	912,02	6,88
ifx-(FFTW/MKL)-OMP-DYN-Debug	1461,76	5,24	1365,10	7,19
ifx-(FFTW/MKL)-OMP-DYN-noDebug	401,54	6,11	329,92	6,10
ifx-(FFTW/MKL)-OMP-Debug	1603,26	9,86	1559,24	8,95
ifx-(FFTW/MKL)-OMP-noDebug	443,18	6,73	415,96	8,74
ifx-(FFTW/MKL)-noOMP-Debug	4808,76	30,53	4386,84	10,24
ifx-(FFTW/MKL)-noOMP-noDebug	1341,68	13,61	927,08	6,37
Moyenne globale	1373,22	18,66	1195,72	7,70

Table 23: Influence de FFTW et MKL sur le temps d'éxecution réels

Dans ce tableau, MKL semble être systématiquement plus rapide pour toutes les expériences, en comparaison avec FFTW. MKL est globalement environ 13 % plus efficace.

Concernant la consommation d'énergie par minute, aucune différence notable n'a été observée : des calculs plus rapides, résultant apriori d'une utilisation plus efficace des ressources physiques existantes, ne correspondent pas à une augmentation de la consommation d'énergie par minute. Par conséquent, la consommation d'énergie globale est principalement déterminée par la durée totale nécessaire pour qu'un programme termine sa tâche. En essence, cela suggère que les programmes qui s'exécutent plus rapidement sont également moins gourmand en énergie. Ces résultats corroborent ceux rapportés par (Memeti *et al.* 2017) et (Pereira *et al.* 2017). La seule différence significative observée est liée à l'activation de OpenMP par rapport à noOpenMP. En effet, l'utilisation simultanée de plusieurs CPUs augmente la consommation d'énergie par minute. Cependant, cette augmentation est relativement mineure comparée aux gains de temps offerts par OpenMP. On peut en conclure que le matériel, lorsqu'il est alimenté en énergie, entraîne une dépense énergétique même s'il reste inactif. La stratégie optimale pour minimiser le gaspillage d'énergie consiste à mieux utiliser le matériel disponible, en évitant de le laisser inactif. Nous pouvons voir dans la table 24 que presque toutes les valeurs sont en bleu, ce qui signifie que nous ne voyons aucun avantage de gfortran, ifort ou ifx concernant la

consommation par minute. Cependant, comme gfortran semble être plus rapide, il conduirait à une consommation d'énergie globale moindre.

Expérience	Consommation énegétique moyenne (J) – gfortran	Std consommation énergétique (J) – gfortran	Consommation énegétique moyenne (J) – ifort	Std consommation énergétique (J) – ifort	Consommation énegétique moyenne (J) – ifx	Std consommation énergétique (J) – ifx
(gfortran/ifort/ifx)-FFTW-OMP-DYN-Debug	3489,23	566,39	3909,33	576,31	3765,81	602,46
(gfortran/ifort/ifx)-FFTW-OMP-DYN-noDebug	3902,83	560,89	4462,09	564,98	4522,11	759,47
(gfortran/ifort/ifx)-FFTW-OMP-Debug	4281,07	498,90	4504,28	507,90	4517,33	566,94
(gfortran/ifort/ifx)-FFTW-OMP-noDebug	4469,58	509,26	4613,95	637,51	4517,08	541,76
(gfortran/ifort/ifx)-FFTW-noOMP-Debug	3389,82	607,27	3379,13	578,19	3065,26	629,05
(gfortran/ifort/ifx)-FFTW-noOMP-noDebug	3388,09	599,72	3220,56	676,94	3025,12	565,11
(gfortran/ifort/ifx)-MKL-OMP-DYN-Debug	3520,18	482,43	4015,35	545,70	3581,70	460,08
(gfortran/ifort/ifx)-MKL-OMP-DYN-noDebug	4015,48	481,77	4423,95	432,14	4302,50	376,83
(gfortran/ifort/ifx)-MKL-OMP-Debug	4092,32	370,89	4430,80	434,36	4405,47	351,06
(gfortran/ifort/ifx)-MKL-OMP-noDebug	4282,08	416,46	4525,75	567,16	4365,06	351,71
(gfortran/ifort/ifx)-MKL-noOMP-Debug	3018,74	475,80	3444,45	683,74	3069,63	640,62
(gfortran/ifort/ifx)-MKL-noOMP-noDebug	3032,50	576,00	3355,50	533,83	2918,67	458,51
Moyenne globale	3740,16	516,89	4023,76	566,83	3837,98	539,36

Table 24: Influence de gfortran, ifort et ifx sur la consommation énergétique par minute

Dans la table 25, en étudiant FFTW et MKL, nous obtenons les mêmes conclusions que ci-dessus : nous ne pouvons voir aucune différence, sauf dans le cas avec gfortran et sans OpenMP.

Expérience	Consommation énegétique moyenne (J) – FFTW	Std consommation énergétique (J) – FFTW	Consommation énegétique moyenne (J) – MKL	Std consommation énergétique (J) – MKL
gfortran-(FFTW/MKL)-OMP-DYN-Debug	3489,23	566,39	3520,18	482,43
gfortran-(FFTW/MKL)-OMP-DYN-noDebug	3902,83	560,89	4015,48	481,77
gfortran-(FFTW/MKL)-OMP-Debug	4281,07	498,90	4092,32	370,89
gfortran-(FFTW/MKL)-OMP-noDebug	4469,58	509,26	4282,08	416,46
gfortran-(FFTW/MKL)-noOMP-Debug	3389,82	607,27	3018,74	475,80
gfortran-(FFTW/MKL)-noOMP-noDebug	3388,09	599,72	3032,50	576,00
ifort-(FFTW/MKL)-OMP-DYN-Debug	3909,33	576,31	4015,35	545,70
ifort-(FFTW/MKL)-OMP-DYN-noDebug	4462,09	564,98	4423,95	432,14
ifort-(FFTW/MKL)-OMP-Debug	4504,28	507,90	4430,80	434,36
ifort-(FFTW/MKL)-OMP-noDebug	4613,95	637,51	4525,75	567,16
ifort-(FFTW/MKL)-noOMP-Debug	3379,13	578,19	3444,45	683,74
ifort-(FFTW/MKL)-noOMP-noDebug	3220,56	676,94	3355,50	533,83
ifx-(FFTW/MKL)-OMP-DYN-Debug	3765,81	602,46	3581,70	460,08
ifx-(FFTW/MKL)-OMP-DYN-noDebug	4522,11	759,47	4302,50	376,83
ifx-(FFTW/MKL)-OMP-Debug	4517,33	566,94	4405,47	351,06
ifx-(FFTW/MKL)-OMP-noDebug	4517,08	541,76	4365,06	351,71
ifx-(FFTW/MKL)-noOMP-Debug	3065,26	629,05	3069,63	640,62
ifx-(FFTW/MKL)-noOMP-noDebug	3025,12	565,11	2918,67	458,51
Moyenne globale	3912,37	589,26	3822,23	488,91

Table 25: Influence de FFTW et MKL sur la consommation énergétique par minute

Comme mentionné précédemment, nous avons effectué cinquante réplications de chaque expérience pour évaluer la répétabilité et identifier les facteurs susceptibles d'influencer ses

performances. Nous avons observé une perte de répétabilité dans certains fichiers de résultats. Par exemple, en utilisant la commande « diff » pour comparer les fichiers de résultats, nous avons trouvé des divergences telles que :

*Fichier1 : 0.00000 0.2552937**3**E-08 0.8355134**34**E-09 0.835**53**563E-09*

*Fichier2 : 0.00000 0.2552937**6**E-08 0.8355138**6**E-09 0.835**47**498E-09*

Les fichiers ./gfortran-FFTW-OMP-DYN-Debug/repli1/pdip.Na2-egs et ./gfortran-FFTW-OMP-DYN-Debug/repli2/pdip.Na2-egs sont différents.

Ou :

*Fichier1 : 0.00000 0.2552937**4**E-08 0.8355134**5**E-09 0.8354**7**134E-09*

*Fichier2 : 0.00000 0.2552937**2**E-08 0.8355150**6**E-09 0.8354**6**803E-09*

Les fichiers ./gfortran-FFTW-OMP-Debug/repli1/pdip.Na2-egs et ./gfortran-FFTW-OMP-Debug/repli7/pdip.Na2-egs sont différents.

Les différences étaient relativement mineures, de l'ordre de 10^{-7} à 10^{-4}. D'un point de vue physique, ces variations peuvent être significatives ou insignifiantes, selon les applications spécifiques. Néanmoins, notre objectif est d'obtenir des résultats répétables, car les machines déterministes sont conçues à cet effet. Cette répétabilité est cruciale pour le débogage et l'identification de toute erreur silencieuse qui pourrait survenir.

Après quelques investigations, nous avons découvert que les problèmes de répétabilité étaient associés à la bibliothèque FFTW. Nous avons examiné pourquoi FFTW nous faisait perdre la répétabilité. Nous avons trouvé dans la documentation de FFTW : « Si vous utilisez le mode FFTW_MEASURE ou FFTW_PATIENT, l'algorithme employé par FFTW n'est pas déterministe : il dépend des mesures de performance à l'exécution. Cela entraînera des variations légères des résultats d'une exécution à l'autre. Cependant, les différences devraient être minimes, de l'ordre de la précision en virgule flottante, et donc ne devraient pas avoir d'impact pratique sur la plupart des applications. Si vous utilisez des plans sauvegardés (wisdom) ou le mode FFTW_ESTIMATE, l'algorithme est déterministe et les résultats devraient être identiques d'une exécution à l'autre. » (https://www.fftw.org/faq/section3.html#nondeterministic).

Il apparaît que, par défaut, la bibliothèque FFTW n'est pas déterministe en raison de sa sélection d'algorithmes à la volée, qui varient en fonction des mesures de performance à

l'exécution. Suivant les recommandations fournies dans la documentation, nous avons modifié la base de code QDD pour engager le mode déterministe de FFTW.

Maintenant, nous considérons les performances lorsque nous modifions légèrement le code de QDD pour obtenir des résultats répétables avec FFTW. Dans la table 26, nous pouvons voir que les performances sont légèrement moins bonnes que dans la table 22. Cependant, les proportions entre gfortran, ifort et ifx restent égales, comme prévu.

Expérience	Temps réel moyen (s) – gfortran	Std temps réel (s) – gfortran	Temps réel moyen (s) – ifort	Std temps réel (s) – ifort	Temps réel moyen (s) – ifx	Std temps réel (s) – ifx
(gfortran/ifort/ifx)-FFTW-OMP-DYN-Debug	779,42	6,29	1173,54	6,66	1534,82	5,11
(gfortran/ifort/ifx)-FFTW-OMP-DYN-noDebug	441,18	7,35	494,36	7,41	477,12	4,97
(gfortran/ifort/ifx)-FFTW-OMP-Debug	825,62	10,44	1315,78	8,15	1667,52	9,36
(gfortran/ifort/ifx)-FFTW-OMP-noDebug	458,90	15,73	504,56	18,51	500,46	10,19
(gfortran/ifort/ifx)-FFTW-noOMP-Debug	3211,00	14,62	4366,38	71,69	5423,02	94,39
(gfortran/ifort/ifx)-FFTW-noOMP-noDebug	2002,36	13,63	1927,26	15,66	1932,06	9,16
(gfortran/ifort/ifx)-MKL-OMP-DYN-Debug	637,66	7,72	1046,90	39,95	1366,24	6,79
(gfortran/ifort/ifx)-MKL-OMP-DYN-noDebug	296,00	7,63	314,06	7,31	329,96	6,05
(gfortran/ifort/ifx)-MKL-OMP-Debug	749,96	9,81	1233,46	6,96	1560,56	9,28
(gfortran/ifort/ifx)-MKL-OMP-noDebug	387,78	7,58	396,08	11,06	415,64	9,03
(gfortran/ifort/ifx)-MKL-noOMP-Debug	2262,38	57,07	3375,74	66,67	4425,20	83,64
(gfortran/ifort/ifx)-MKL-noOMP-noDebug	993,64	20,92	911,76	16,12	927,38	10,52
Moyenne globale	1087,16	14,90	1421,66	23,01	1713,33	21,54

Table 26: Influence de gfortran, ifort et ifx sur le temps d'execution réel, lorsque l'on rend FFTW répétable

Dans la table 27, nous nous concentrons sur les performances de FFTW et MKL, comme dans le tableau 2. Les données résultantes ont montré une baisse des performances moyennes d'environ 18 % pour FFTW (1373 secondes dans le tableau 2 et 1620 secondes dans le tableau 6), tandis que MKL a maintenu des niveaux de performance constants, comme prévu. Cependant, cette modification a permis d'obtenir une répétabilité bit à bit sur toutes les réplications pour chaque expérience utilisant FFTW.

Expérience	Temps réel moyen (s) – FFTW	Std temps réel (s) – FFTW	Temps réel moyen (s) – MKL	Std temps réel (s) – MKL
gfortran-(FFTW/MKL)-OMP-DYN-Debug	779,42	6,29	637,66	7,72
gfortran-(FFTW/MKL)-OMP-DYN-noDebug	441,18	7,35	296,00	7,63
gfortran-(FFTW/MKL)-OMP-Debug	825,62	10,44	749,96	9,81
gfortran-(FFTW/MKL)-OMP-noDebug	458,90	15,73	387,78	7,58
gfortran-(FFTW/MKL)-noOMP-Debug	3211,00	14,62	2262,38	57,07
gfortran-(FFTW/MKL)-noOMP-noDebug	2002,36	13,63	993,64	20,92
ifort-(FFTW/MKL)-OMP-DYN-Debug	1173,54	6,66	1046,90	39,95
ifort-(FFTW/MKL)-OMP-DYN-noDebug	494,36	7,41	314,06	7,31
ifort-(FFTW/MKL)--OMP-Debug	1315,78	8,15	1233,46	6,96
ifort-(FFTW/MKL)--OMP-noDebug	504,56	18,51	396,08	11,06
ifort-(FFTW/MKL)-noOMP-Debug	4366,38	71,69	3375,74	66,67
ifort-(FFTW/MKL)--noOMP-noDebug	1927,26	15,66	911,76	16,12
ifx-(FFTW/MKL)-OMP-DYN-Debug	1534,82	5,11	1366,24	6,79
ifx-(FFTW/MKL)-OMP-DYN-noDebug	477,12	4,97	329,96	6,05
ifx-(FFTW/MKL)-OMP-Debug	1667,52	9,36	1560,56	9,28
ifx-(FFTW/MKL)-OMP-noDebug	500,46	10,19	415,64	9,03
ifx-(FFTW/MKL)-noOMP-Debug	5423,02	94,39	4425,20	83,64
ifx-(FFTW/MKL)-noOMP-noDebug	1932,06	9,16	927,38	10,52
Moyenne globale	1613,08	18,29	1201,69	21,34

Table 27: Influence de FFTW et MKL sur le temps réel d'execution, lorsque l'on rend FFTW répétable

Concernant la reproductibilité entre les différentes expériences, malgré ces ajustements de paramètres pour FFTW, nous avons observé que la reproductibilité reste inaccessible pour toutes les options, y compris l'option Debug, car les résultats varient entre chaque expérience sans exception (nous n'obtenons pas de résultats identiques au bit près entre les différentes expériences, mais nous avons des résultats identiques au bit près entre les réplications d'une même expérience. Nous avons donc la répétabilité, mais pas la reproductibilité). Cela signifie, par exemple, que lorsque nous exécutons l'expérience avec les options ifx-MKL-noOMP-noDebug, nous n'obtenons pas de résultats identiques au bit près par rapport à l'exécution avec les options ifx-MKL-noOMP-Debug (la seule chose qui change est l'option Debug).

Par la suite, nous avons tenté de déterminer si la reproductibilité pouvait être atteinte entre différentes configurations. Bien que nous ne nous attendions pas à des résultats identiques entre les différents compilateurs et bibliothèques FFT, étant donné leur rôle principal dans les calculs, nous espérions obtenir la reproductibilité entre les expériences menées avec des paramètres différents de OpenMP versus NoOpenMP et Debug versus NoDebug.

Dans cette quête, nous avons suivi les recommandations de (Charpilloz *et al.* 2017), (Li *et al.* 2016) et (Lionel 2013) pour modifier les options de compilation. (Li *et al.* 2016) ont fourni plusieurs paramètres de compilation restrictifs pour gfortran et ifort.

Pour gfortran, nous avons implémenté les paramètres suivants :

-w -O0 -ffloat-store -fno-unsafe-math-optimizations -fno-associative-math -fno-reciprocal-math -fno-finite-math-only -fno-rounding-math -fno-cx-limited-range

Pour le mode debug :

-pg -w -g -fbacktrace -O0 -ffloat-store -fno-unsafe-math-optimizations -fno-associative-math -fno-reciprocal-math -fno-finite-math-only -fno-rounding-math -fno-cx-limited-range

Pour ifort et ifx, nous avons utilisé :

-O0 -fp-model strict -fp-speculation=strict -mp1 -no-vec -no-simd

Pour le mode debug :

-g -CB -traceback -O0 -fp-model strict -fp-speculation=strict -mp1 -no-vec -no-simd

Il est toutefois à noter que les options -no-simd et -mp1 n'étaient pas disponibles pour le compilateur ifx. Cela pourrait conduire à un contexte plus restrictif pour gfortran ou ifort que pour ifx.

En désactivant toutes les options de compilation permettant l'optimisation, nous pouvons observer dans la table 28 que les compilateurs gfortran, ifort et ifx offrent des performances comparables. Cette similarité de performance peut suggérer que les performances initialement supérieures de gfortran pourraient être attribuées aux options de compilation fournies avec QDD étant plus compatibles avec gfortran, indiquant potentiellement que les performances du compilateur Intel Fortran pourraient être améliorées de manière similaire. Ou cela pourrait être dû à des options de compilation plus restrictives sur gfortran, que nous essayons d'appliquer pour obtenir la reproductibilité entre les options Debug et OMP. Cependant, MKL continue de surpasser FFTW, ce dernier restant en mode répétable, sacrifiant ainsi une partie de la performance.

Expérience	Temps réel moyen (s) – gfortran	Std temps réel (s) – gfortran	Temps réel moyen (s) – ifort	Std temps réel (s) – ifort	Temps réel moyen (s) – ifx	Std temps réel (s) – ifx
(gfortran/ifort/ifx)-FFTW-OMP-DYN-Debug	1157,80	6,02	1210,82	4,41	1537,56	6,19
(gfortran/ifort/ifx)-FFTW-OMP-DYN-noDebug	1157,34	6,06	842,94	6,39	728,98	5,38
(gfortran/ifort/ifx)-FFTW-OMP-Debug	1285,22	9,07	1345,42	7,13	1682,96	17,30
(gfortran/ifort/ifx)-FFTW-OMP-noDebug	1283,40	10,45	966,82	8,98	807,32	10,10
(gfortran/ifort/ifx)-FFTW-noOMP-Debug	4489,76	14,01	4833,34	16,80	5517,20	15,78
(gfortran/ifort/ifx)-FFTW-noOMP-noDebug	4487,72	12,47	3716,08	24,89	3309,40	135,87
(gfortran/ifort/ifx)-MKL-OMP-DYN-Debug	1009,60	7,19	1059,08	5,03	1421,32	6,15
(gfortran/ifort/ifx)-MKL-OMP-DYN-noDebug	1011,24	7,96	699,90	5,30	585,90	6,06
(gfortran/ifort/ifx)-MKL-OMP-Debug	1187,14	8,48	1264,46	7,16	1595,02	8,56

(gfortran/ifort/ifx)-MKL-OMP-noDebug	1189,22	8,48	878,12	9,33	727,50	10,08
(gfortran/ifort/ifx)-MKL-noOMP-Debug	3504,38	6,04	3850,58	16,43	4566,86	16,61
(gfortran/ifort/ifx)-MKL-noOMP-noDebug	3506,68	7,71	2739,50	7,38	2356,56	23,48
Overall Mean	2105,79	8,66	1950,59	9,93	2069,72	21,80

Table 28: Influence de gfortran, ifort et ifx sur le temps d'execution réel, lorsque l'on désactive toutes les opmisations lors de la compilation

Expérience	**Temps réel moyen (s) – FFTW**	**Std temps réel (s) – FFTW**	**Temps réel moyen (s) – MKL**	**Std temps réel (s) – MKL**
gfortran-(FFTW/MKL)-OMP-DYN-Debug	1157,80	6,02	1009,60	7,19
gfortran-(FFTW/MKL)-OMP-DYN-noDebug	1157,34	6,06	1011,24	7,96
gfortran-(FFTW/MKL)-OMP-Debug	1285,22	9,07	1187,14	8,48
gfortran-(FFTW/MKL)-OMP-noDebug	1283,40	10,45	1189,22	8,48
gfortran-(FFTW/MKL)-noOMP-Debug	4489,76	14,01	3504,38	6,04
gfortran-(FFTW/MKL)-noOMP-noDebug	4487,72	12,47	3506,68	7,71
ifort-(FFTW/MKL)-OMP-DYN-Debug	1210,82	4,41	1059,08	5,03
ifort-(FFTW/MKL)-OMP-DYN-noDebug	842,94	6,39	699,90	5,30
ifort-(FFTW/MKL)--OMP-Debug	1345,42	7,13	1264,46	7,16
ifort-(FFTW/MKL)--OMP-noDebug	966,82	8,98	878,12	9,33
ifort-(FFTW/MKL)-noOMP-Debug	4833,34	16,80	3850,58	16,43
ifort-(FFTW/MKL)--noOMP-noDebug	3716,08	24,89	2739,50	7,38
ifx-(FFTW/MKL)-OMP-DYN-Debug	1537,56	6,19	1421,32	6,15
ifx-(FFTW/MKL)-OMP-DYN-noDebug	728,98	5,38	585,90	6,06
ifx-(FFTW/MKL)-OMP-Debug	1682,96	17,30	1595,02	8,56

ifx-(FFTW/MKL)-OMP-noDebug	807,32	10,10	727,50	10,08
ifx-(FFTW/MKL)-noOMP-Debug	5517,20	15,78	4566,86	16,61
ifx-(FFTW/MKL)-noOMP-noDebug	3309,40	135,87	2356,56	23,48
Overall Mean	2242,23	17,63	1841,84	9,30

Table 29: Influence de FFTW et MKL sur le temps d'execution réel, lorsque l'on désactive toutes les optimisations lors de la compilation

Nous avons réussi à atteindre la reproductibilité entre les paramètres OpenMP et Debug. La présence de répétabilité numérique indique que le manque de reproductibilité n'est pas dû à des incertitudes intrinsèques au code, mais à des différences dans l'exécution, basées sur les options de compilation. Les instructions du programme ne sont pas exécutées dans exactement le même ordre (out of order), ce qui peut entraîner des divergences, notamment dans les calculs en virgule flottante.

Les paramètres de compilation actuels nous permettent désormais d'obtenir des résultats identiques au bit près entre les expériences menées avec et sans OpenMP, ainsi qu'entre celles effectuées en mode debug et non-debug (ce qui peut être utile pour déboguer efficacement, en évitant que le programme non-debug ait un comportement différent de celui en mode debug). Néanmoins, en ce qui concerne les compilateurs et les bibliothèques FFT, les résultats identiques au bit près ne sont pas atteints, bien que cette divergence soit attendue, car les compilateurs et les bibliothèques FFT ne sont pas censés exécuter les instructions dans le même ordre exact, étant donné qu'ils fonctionnent différemment.

IV.7.3. Discussion

Nos observations révèlent que MKL non seulement offre de meilleures performances, mais maintient également la répétabilité, contrairement à FFTW, qui montre une dégradation des performances lorsque la répétabilité est imposée. Malgré la nature propriétaire de MKL, il reste disponible gratuitement. D'un autre côté, FFTW est open source. Nos résultats corroborent les conclusions de (Gambron et Thorne 2020), qui ont rapporté de meilleures performances de MKL par rapport à FFTW. En termes d'efficacité des compilateurs, le compilateur gfortran montre des performances élevées par rapport à ses homologues commerciaux, remettant en question les données prédominantes des benchmarks Polyhedron (Polyhedron benchmarks, n. d.) et les résultats de (Young-S et al. 2017). Par conséquent, nous recommandons l'intégration de gfortran avec MKL pour des performances optimales. L'utilisation d'OpenMP a démontré

des avantages substantiels. Malgré des préoccupations potentielles, la parallélisation avec cette bibliothèque n'a pas compromis la répétabilité et a considérablement amélioré les performances, en particulier sur un système où nous avons utilisé 16 des 96 cœurs physiques pour la parallélisation OpenMP.

Les considérations de consommation d'énergie suggèrent que l'optimisation de l'utilisation des ressources matérielle et l'accélération de l'exécution du code peuvent être les stratégies les plus efficaces pour minimiser la consommation d'énergie. Cependant, il faut prendre en compte l'effet rebond et le paradoxe de Jevons (Alcott 2005 ; Sorrell 2009), qui stipule que l'optimisation de l'utilisation de l'énergie conduira, au final, à une augmentation de la consommation d'énergie. Dans notre cas, des programmes qui s'exécutent plus vite et qui consomment moins d'énergie, donneront lieu à des programmes plus gros, des simulations plus larges, ou simplement plus de programmes qui s'exécuteront sur les clusters, et donc finalement, la consommation énergétique globale ne diminuera pas avec l'optimisation énergétique individuelle des programmes.

La quête de vitesses de traitement plus rapides pourrait entrer en conflit avec les objectifs de répétabilité et de reproductibilité, présentant un dilemme philosophique dans la pratique scientifique du calcul. Les défis de la reproductibilité avec OpenMP/No-OpenMP et Debug/No-Debug (obtenir des résultats bit à bit identiques avec ces options) ont notablement diminué les performances, similaires aux problèmes rencontrés avec FFTW.

Il reste cependant des limitations à notre étude. L'évaluation des performances de la FFT, en particulier en utilisant FFTW et MKL, pourrait être influencée par des facteurs dimensionnels. Dans notre analyse, l'application était confinée à QDD, où le problème est en 3D. Cependant, (Gambron et Thorne 2020) corroborent nos conclusions. De plus, la comparaison des performances de gfortran et ifort pourrait varier en fonction du type d'application, comme le suggèrent (Young-S et al. 2017), qui ont observé des performances supérieures avec ifort sur plusieurs applications, tandis que nous observons la conclusion opposée dans notre cas avec QDD. Nous avons conservé les options de compilation initiales des packages QDD. Cependant, étant donné qu'ifx est nouveau, certaines optimisations peuvent exister qui ne sont pas disponibles avec ifort, ce qui pourrait conduire à de meilleures performances d'ifx par rapport à ifort. De plus, ifx pourrait mieux performer avec des fonctionnalités plus modernes des CPU Intel. Le CPU que nous avons utilisé dans nos tests date de 2017 (il y a 7 ans).

IV.8. Etude de reproductibilité de calcul quantique sur simulateur et sur machine réelles

IV.8.1. Courte introduction à l'informatique quantique

La physique classique nous permet de comprendre les phénomènes à une échelle macroscopique, mais elle devient insuffisante lorsqu'on passe à une échelle microscopique. C'est là qu'intervient la physique quantique, et plus précisément la mécanique quantique, pour décrire le comportement des particules. Les lois de la physique quantique, telles que la quantification des niveaux d'énergie, la dualité onde-particule et le principe d'incertitude de Heisenberg, nous offrent un cadre pour comprendre ces comportements. Cependant, en informatique quantique, trois principes sont particulièrement essentiels : la superposition quantique, l'intrication quantique et la réduction du paquet d'onde.

La superposition quantique stipule qu'une particule peut exister simultanément dans plusieurs états. Contrairement à la physique classique où une particule a un état déterminé à tout moment, la superposition permet à une particule d'être dans une combinaison de plusieurs états. L'intrication quantique décrit un phénomène où deux particules deviennent liées de telle manière que l'état de l'une détermine instantanément l'état de l'autre, indépendamment de la distance qui les sépare. Cela crée une corrélation entre les particules qui est fondamentale pour le traitement de l'information quantique. La réduction du paquet d'onde, enfin, souligne que la mesure d'un état quantique affecte ce dernier. En mesurant un système quantique, on fait « collapser » son état superposé en un état spécifique, ce qui signifie que l'état de superposition original ne peut plus être restauré après la mesure.

En informatique quantique, l'unité de base de l'information est le qubit, ou bit quantique. Contrairement au bit classique qui est soit 0 soit 1, un qubit peut être dans un état de superposition de $|0\rangle$ et $|1\rangle$. Les états d'un qubit sont définis par des coefficients complexes qui déterminent les probabilités des résultats des mesures. Les qubits sont manipulés par des portes quantiques, analogues aux portes logiques en informatique classique, mais opérant sur les états quantiques sans causer de décohérence. Chaque porte quantique est définie par une matrice unitaire qui modifie l'état des qubits. Ces portes sont réversibles, ce qui permet de restaurer l'état initial du système en appliquant l'opération inverse.

IBM a développé des ordinateurs quantiques dont les processeurs, bien que de taille comparable à ceux des ordinateurs classiques, nécessitent des systèmes de refroidissement

sophistiqués pour maintenir des températures ultra-basses proches du zéro absolu. Ces conditions permettent d'éviter la décohérence et de maintenir les états quantiques nécessaires aux calculs. Les qubits dans les ordinateurs quantiques IBM sont formés à partir de jonctions Josephson, utilisant des supraconducteurs. En manipulant ces qubits avec des photons à micro-ondes, on peut contrôler et lire les informations quantiques.

La puissance des ordinateurs quantiques réside dans leur capacité à exploiter la superposition et l'intrication pour explorer simultanément de multiples configurations. En utilisant l'interférence, les calculs quantiques permettent de renforcer ou d'annuler certains résultats, isolant ainsi les solutions correctes à des problèmes complexes. Ce processus permet aux ordinateurs quantiques de traiter des algorithmes multidimensionnels beaucoup plus rapidement que les ordinateurs classiques. Nous retenons pour la suite de notre étude un des algorithmes essentiels à ce paradigme de calcul.

IV.8.2. L'algorithme de Grover

L'algorithme de Grover, proposé par Lov Grover en 1996, est un algorithme quantique destiné à accélérer la recherche dans des bases de données non structurées. Son importance réside dans sa capacité à offrir une accélération quadratique par rapport aux algorithmes classiques, réduisant ainsi le temps de recherche de O(N) à O(√N), où N est la taille de la base de données. Cet exploit repose sur les principes fondamentaux de la mécanique quantique, notamment la superposition et l'interférence.

L'algorithme commence par préparer un état de superposition égale de tous les éléments de la base de données. Cela est réalisé en appliquant une porte Hadamard sur chacun des n qubits, où $N=2^n$. Chaque qubit passe ainsi de l'état $|0\rangle$ à une superposition $(|0\rangle+|1\rangle)/\sqrt{2}$, créant une superposition de tous les N états possibles du registre quantique. Cette étape initiale est cruciale car elle permet à l'algorithme de tester simultanément toutes les entrées de la base de données.

L'étape suivante implique l'application de l'oracle quantique, une fonction spéciale qui marque l'état cible. L'oracle est implémenté de manière à inverser la phase de l'état correspondant à l'élément recherché, transformant $|x\rangle$ en $-|x\rangle$ pour la solution correcte, tandis que les autres états restent inchangés. Cet oracle est souvent spécifique au problème, mais son rôle central est de distinguer la solution correcte des autres états par un changement de phase.

Après l'oracle, l'algorithme applique l'amplification d'amplitude, également connue sous le nom de diffusion de Grover. Cette étape commence par l'application d'une transformation de

Hadamard sur tous les qubits, suivie d'une inversion par rapport à la moyenne. Concrètement, cette opération inverse les amplitudes des états par rapport à la moyenne de toutes les amplitudes, augmentant ainsi l'amplitude de l'état marqué et réduisant celles des autres états. En termes plus simples, cela amplifie la probabilité de mesurer l'état correct lors de la mesure finale.

Ces deux étapes, l'oracle et l'amplification d'amplitude, sont répétées environ $\pi/4\sqrt{N}$ fois pour maximiser la probabilité de trouver la solution correcte. Le nombre optimal d'itérations est un point crucial de l'algorithme, car une trop grande répétition peut diminuer cette probabilité après un certain seuil. Cette optimisation est une démonstration élégante de l'équilibre entre les forces d'attraction et de répulsion des états quantiques.

Enfin, l'algorithme se termine par une mesure des qubits, projetant l'état quantique sur l'un des N états classiques. Grâce aux répétitions de l'amplification d'amplitude, l'état mesuré est, avec une grande probabilité, celui de l'élément recherché. Cette probabilité peut être très proche de 1 pour de grandes bases de données, démontrant l'efficacité de l'algorithme.

Ces différentes étapes sont illustrées dans la figure 31.

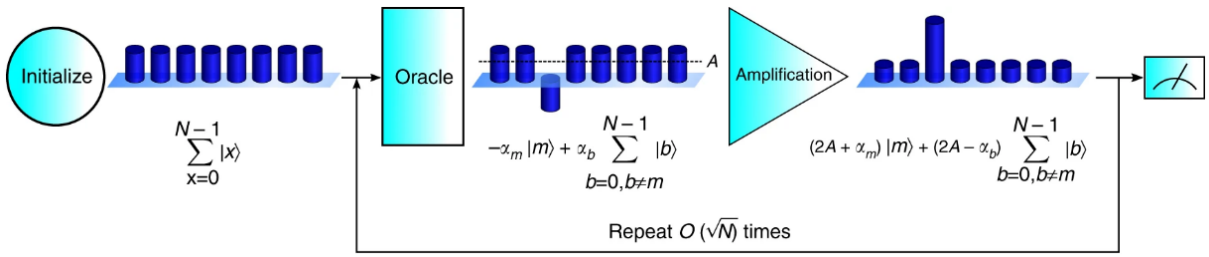


Figure 31: Schéma de fonctionne de l'algorithme de Grover (Figgatt et al. 2017)

IV.8.3. Nos expériences de reproductibilité sur les machines IBM

Pour répondre au problème évoqué dans la section proposition, nous avions commencé par des implémentations de cet algorithme avec 3 qubits (Hill *et al.* 2023), et nous avons également abordé une version à 5 qubits (Antunes *et al.* 2024) (Figure 32 ci-après) sur une nouvelle machine quantique d'IBM proposant, depuis 2023, 127 qubits, ce qui est assez exceptionnel pour des machines libres d'accès permettant au monde de la recherche de progresser en se confrontant réellement aux machines physiques, que ce soit pour l'éducation ou pour la recherche. Nous saluons grandement l'initiative d'IBM à ce sujet.

Nous cherchons à comparer la reproductibilité de ces algorithmes dans des environnements quantiques simulés et réels. Cette comparaison sert non seulement à mettre en lumière l'état actuel du matériel informatique quantique dont nous pouvons disposer, mais elle nous éclaire aussi sur les défis potentiels qui sont encore devant nous pour réaliser un calcul quantique fiable en pratique. Nous présentons sommairement les 2 versions de l'algorithme qui ont été testées (3 et 5 qubits). La version plus complexe à 5 qubits nécessite en fait 10 qubits en raison des qubits de contrôles et des exigences des portes quantiques retenues. Dans un premier temps, nous avons exécuté les deux versions de l'algorithme de Grover sur un simulateur de machine quantique fourni par IBM. Cet environnement de simulation est conçu pour imiter les opérations d'un ordinateur quantique sans les contraintes physiques et les interactions environnementales du matériel quantique réel. Dans un second temps, les mêmes algorithmes ont été exécutés sur une vraie machine quantique disponible sur la plateforme IBM. Nous avons exécuté les deux implémentations de Grover sur la plus grande machine quantique d'IBM disponible au public, Brisbane, avec 127 qubits. Nous avons réalisé 20 réplications pour chaque expérience. Pour l'implémentation à 3 qubits, nous avons utilisé une machine plus petite et plus ancienne avec 7 qubits. Le principal indicateur d'intérêt est la fréquence à laquelle l'algorithme identifiait correctement l'état cible (sur 1024 « shots », le réglage par défaut pour les machines quantiques IBM).

Notre étude empirique sur la reproductibilité de l'algorithme de Grover dans les environnements informatiques quantiques révèle des disparités significatives entre les simulations et les résultats des machines quantiques (Aashish *et al.* 2010). Ces différences dépendent du nombre de qubits et du nombre de portes utilisés dans le circuit.

Les résultats simulés pour la version à 5 qubits de l'algorithme de Grover donnent une probabilité de 99,7% d'obtenir la solution recherchée ciblé/d'identifier l'état 00101. Il faut noter que l'Oracle utilisé utilise très peu de qubit par rapport aux nombres de qubits nécessaires pour un TSP ou un problème SAT. Le circuit utilisé pour l’algorithme de Grover 5 qubits est illustré dans la Figure 32. Sur ce circuit, il manque les 4 boucles d’amplification répétées de Grover.

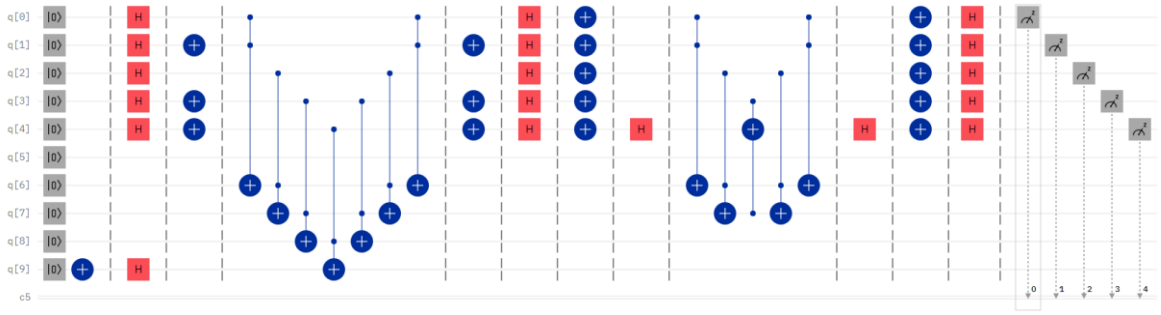


Figure 32: Circuit pour l'algorithme de Grover que nous avons implémenté sur 5 qubits avec 1 itération

Cependant, lorsque cet algorithme a été implémenté les machines IBM, aucune solution n'a émergé, résultant en une distribution presque uniforme comme le montre la figure 34, en comparaison avec la figure 33 montrant le résulat sur simulateur.

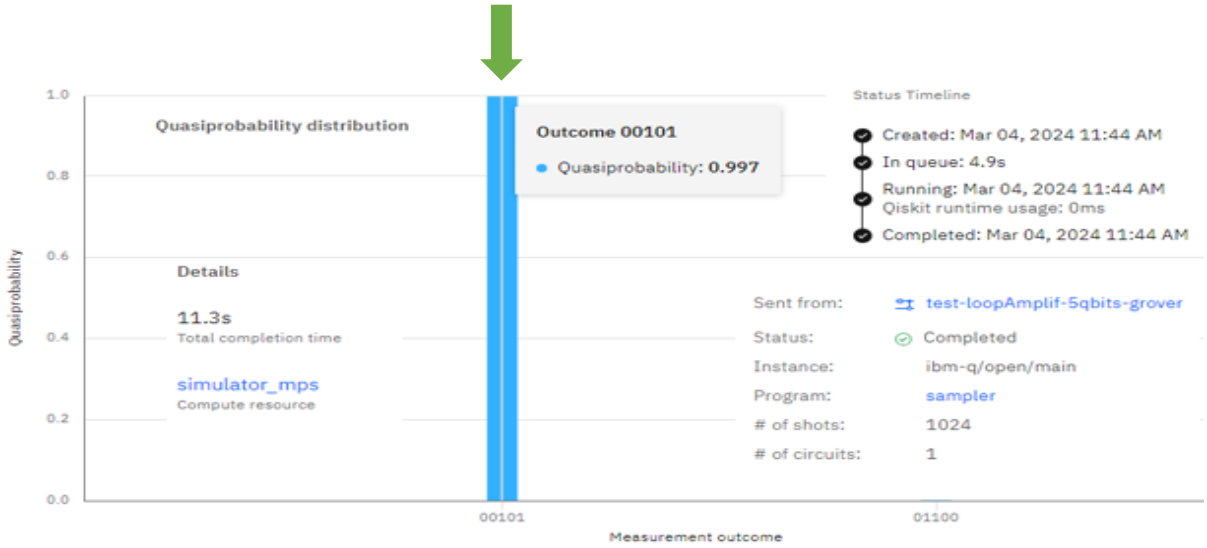


Figure 33: Résultat de Grover 5 qubits sur simulateur

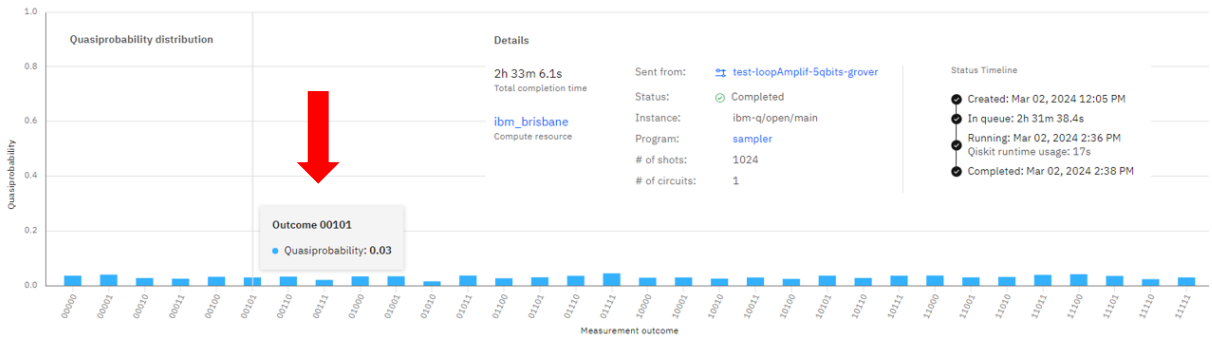


Figure 34: Résultat de Grover 5 qubits sur machine réelle (Brisbane 127 qubits)

Figure 35: Reproductibilité de Grover sur 3 machines IBM différentes

Afin de mesurer la reproductibilité, nous avons réaliser cette expérience (l'execution de notre algorithme de Grover sur 5 qubit) sur différentes machines quantiques réelles disponible.

Les résultats, présentées sur les figures suivantes, montrent que les taux d'erreurs ne permettent d'avoir de résultats probants sur aucunes des machines disponibles (mars 2024), comme présenté sur le figure 35.

Pour l'implémentation à 3 qubits, alors que le simulateur donne une probabilité de 95% d'obtenir l'état marqué par l'Oracle, l'exécution sur les machines à 7 ou à 127 qubits donne un résultat compris entre 60% et 70%, comme montré sur la Figure 36.

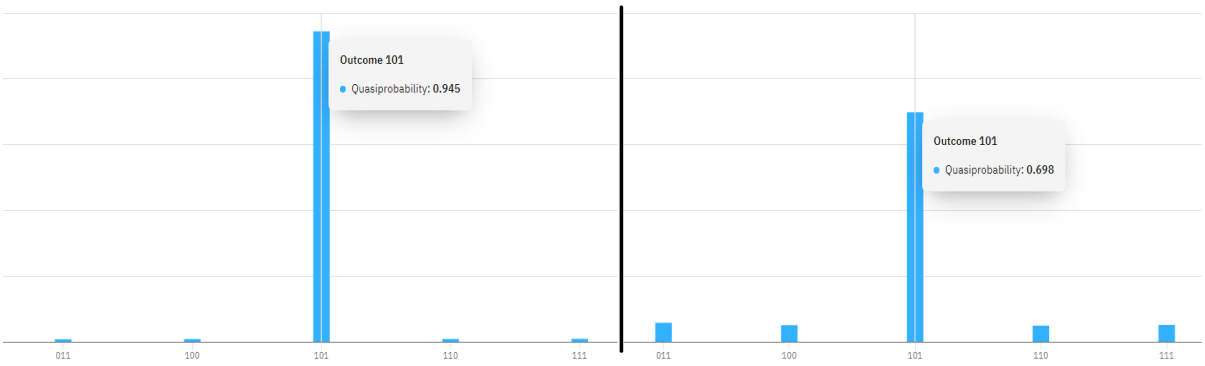


Figure 36: Résulat de Grover 3 qubits sur simulateur VS sur machine réelle (Brisbane 127 qubits)

Les différences entre l'implémentation sur 3 et 5 qubits viennent des bruits et du taux d'erreur des différentes portes. Nous constatons par exemple que l'intervalle des taux d'erreurs sur la porte Pauli-X varie de 0.09 à $8.730.10^{-4}$ selon les qubits de la machine 127 qubits (mesures en mars 2024. Ces taux d'erreurs ne sont pas fixes et peuvent varier selon les jours, mais ils progressent, les mauvaises mesures sont à 10^{-3} en septembre 2024. Cependant, concernant l'algorithme de Grover à 3 qubits, nous observons une amélioration notable des performances par rapport à nos travaux précédents. Dans cette étude antérieure, l'exécution de l'algorithme de Grover à 3 qubits sur diverses machines à 7 qubits donnait des résultats très éloignées des résultats théoriques. Cela suggère que la précision des machines actuellement disponibles s'est améliorée par rapport à celles qui étaient à notre disposition il y a un an. Pour réaliser le niveau d'avancée technologique, la figure ci-après montre par exemple de façon visuelle les probabilités de mesures correctes sur les qubits d'une machine IBM à 127 qubits (en mars 2024). Les colorations foncées ont des taux d'erreur de lecture de 10^{-3}, et les plus claires montrent 2 qubits présentant un taux d'erreurs de 10^{-1} sur la lecture de ces 2 qubits. Ce type de problème se retrouve au niveau de l'application des portes quantiques.

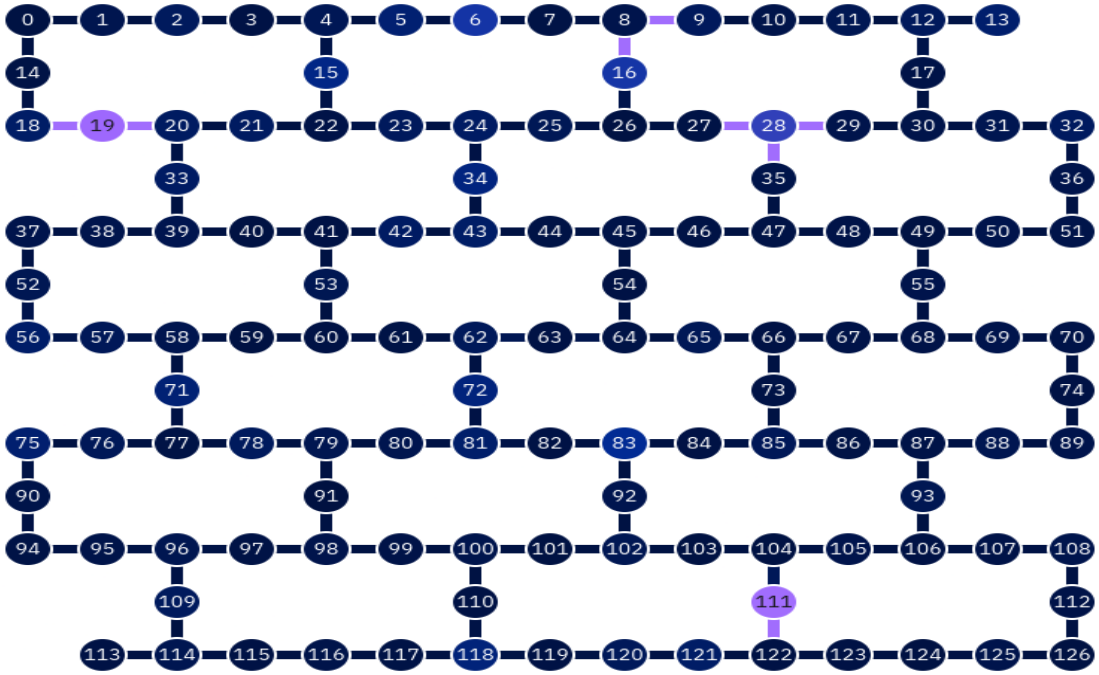


Figure 37: Représentation visuelle du taux d'erreur de lecture de chaque qubit

Sur la machine IBM testée, le taux de succès d'une porte CNOT est de 99,2%, ce qui correspond à un taux d'erreur de 0,8%. Ce taux est variable, mais partons de cette hypothèse. Dans notre circuit, qui utilise jusqu'à 4 portes Toffoli sur certains qubits, chaque porte Toffoli est transpilée en 7 portes CNOT (cela peut également varier), entraînant un total de 28 portes CNOT pour certains qubits. Cela se traduit par une probabilité de succès de 99,2% à la puissance 28, soit 79,85% pour chaque qubit étant affecté par 4 portes Toffoli. Pour un circuit utilisant un registre de 10 qubits (incluant des qubits auxiliaires), on peut donc estimer que la probabilité que chaque qubit soit correct lors de la mesure est d'environ 80% (les 79,85% vus précemment). Elevé à la puissance 10, il y a environ 10% de chance que la mesure soit correcte sur tous les qubits. Avec une seule itération de l'algorithme de Grover nous obtenons la cible serait obtenue avec 25% (0,25) dans le cas d'une machine parfaite. C'est ce que nous trouvons avec les différents simulateurs utilisés (reproductibilité numérique acquise au niveau simulation). Ainsi, lors de la mesure finale, dans 10% des cas, le vecteur de probabilité est correct, et nous avons 25% de chance de mesurer la bonne valeur. Néanmoins, pour connaitre la probabilité de mesurer la bonne valeur, il faut également additioner la probabilité de mesurer la bonne valeur dans les 90% des cas où le vecteur de probabilité final n'est pas correct, suite à une ou plusieurs erreurs sur les qubits durant l'execution. Nous pouvons partir de l'hypothèse

que dans ce cas, chaque solution finale sera équiprobable (100/32 = 3.125% pour chaque état). Cela nous donne 0.25 * 0.1 + 0.03125 * 0.9 = 0.053. Nous pouvons en déduire finalement qu'en prenant en compte les erreurs, nous n'avons que 0,053 de quasiprobabilité que la mesure finale soit correcte (mesure l'état recherché). En pratique, les erreurs semblent encore plus importantes, car nous ne voyons pas émerger la bonne solution.

Si l'on considère l'utilisation correcte de Grover, est donc l'application optimale de 4 itérations pour 5 qubits, la probabilité théorique d'obtenir la bonne solution monte à 99,7% au lieu de 25%. Néanmoins, dans la pratique, il faut pour cela avoir 16 portes Toffoli au lieu de 4 pour implémenter les 4 itérations à la suite, et donc 112 CNOT au lieu de 28. Les erreurs vont donc s'accumuler, et bien que théoriquement, le résultat est largement meilleur, ce n'est pas le cas en pratique lors d'une mesure sur machine réelle. Supposons un taux de réussite à 99,2% = 0.992^{112} = 40% par qubits. On a 10 qubits : 0.4^{10} = 0.0001. Il devient donc en pratique impossible d'avoir la bonne solution, car la probabilité d'avoir le bon vecteur avec 4 itérations de Grover nous donne une valeur extremement faible, plus basse que l'équiprobabilité (0.0001 * 0,997 = 0,0000997). Cette valeur est donc négligeable par rapport à 0.03125, qui est l'équiprobabilité. Tant que la fiabilité de l'application des portes ne sera pas plus élevée, nous ne pourrons pas reproduire correctement nos expériences. A l'heure de l'écriture de ce manuscrit, les taux d'erreurs sur les portes ont progressé (10^{-3} et 10^{-4} sur Pauli-X par exemple pour les machines IBM ouvertes pour la recherche).

L'étude comparative de l'algorithme de Grover dans les environnements de calcul quantique simulés et réels révèle des défis techniques auxquels les différents constructeurs sont confrontés. Compte tenu de ces limitations, les recherches actuelles utilisent essentiellement les simulateurs quantiques dans lesquels parfois on ajoute des modèles de simulation de bruits.

La technologie des ordinateurs quantique étant relativement récente, il est normal de trouver que dans la pratique, les résultats sur machines réelles ne sont pas aussi bons que nous pourrions l'espérer via les simulateurs. IBM fait partie des seuls constructeurs à permettre l'utilisation en ligne de vraies machines quantiques. Les simulateurs permettent dès à présent aux chercheurs de se former sur l'informatique quantique, avec l'espoir que dans un futur proche, les évolutions technologiques permettront aux machines quantiques d'atteindre la fiabilité requise pour réaliser une recherche applicable et reproductible.

IV.8.4. Discussion

Lors de notre 1er test en 2022, nous avons travaillé sur l'algortihme de Grover avec 3 qubits. Les résultats montrent la stabilité des résultats de simulation de cet algorithme, ce qui confirme que notre implémentation de l'algorithme de Grover était correcte. Cependant, nous n'étions pas en mesure d'obtenir des résultats satisfaisants sur des machines quantiques réelles.

Ensuite, en 2024, nous avons ré-itéré les expériences précédentes et nous sommes passés à un algortihme de Grover sur 5 qbits. Les résultats de l'étude, montrant notamment la sous-performance de l'algorithme de Grover à 5 qubits sur du matériel quantique réel, mettent en évidence des défis significatifs dans la mise en œuvre pratique des algorithmes quantiques sur les machines réelles. Nous sommes à l'époque des machines NISQ (Noisy Intermediate Scale Quantum), des ordinateurs quantiques imparfaits (bruités) dits de taille intermédiaire.

L'algorithme de Grover étant un algorithme « basique » mais essentiel, nous l'avons retenu pour tester facilement les machines quantiques. Plus nous utilisons de portes sensibles avec un taux d'erreur important, et plus nous avons de risque d'avoir un mauvais résultat. Il nous semble que les machines IBM se soient améliorées durant les trois dernières années, puisque nos tests avec un Grover 3 qubits, qui n'était pas concluant au départ, le sont devenus. Néanmoins, lors de nos essais avec un algortihme de Grover 5 qubits, utilisant 10 qubits pour l'implémentation de l'ensemble des portes quantiques, nous avons constaté que les résultats ne sont pas encore satisfaisants. L'informatique quantique ayant les capacités de révolutionner bien des domaines en termes de performances, des avancées seront à attendre dans les années qui viennent.

IV.9. Conclusion

Dans ce chapitre, nous avons présenté nos différentes expériences visant à explorer et améliorer la reproductibilité dans divers contextes technologiques. Nous avons d'abord développé un modèle multi-agent pour des simulations épidémiologiques, afin d'étudier la propagation des maladies de manière reproductible. Ensuite, nous avons exploré les différentes méthodes de parallélisation des PRNGs, en évaluant les performances en termes de temps et de qualités statistiques de ces dernières. Nous avons poursuivi en examinant la reproductibilité des PRNGs dans les technologies du machine learning, suivie d'un cas d'étude spécifique qui a mis en évidence les défis de reproductibilité dans une application de machine learning réelle. Par ailleurs, nous avons étudié l'impact du matériel physique, en particulier le simultaneous multi-threading, sur la reproductibilité des mesures de performances, démontrant que le matériel peut

influencer de manière significative les résultats, et varier suivant le type d'application. Nous avons également analysé l'utilisation des bibliothèques scientifiques pour la transformation de Fourier, en examinant leur impact sur les performances et la reproductibilité des calculs. Enfin, nous avons mené une étude de reproductibilité utilisant le paradigme du calcul quantique, avec des simulations, mais aussi sur des machines quantiques réelles mises gracieusement à disposition par IBM. Ces tests de reproductibilité ont souligné les défis et les opportunités que présentent cette technologie émergente.

V. Conclusion générale

V.1. Rappel du contexte

Durant cette thèse, nous avons étudié les différentes raisons pour lesquelles nous pouvions perdre la reproductibilité dans le cadre du calcul scientifique. Dans un premier temps, nous avons proposé un état de l'art à jour des différentes causes de la perte de reproductibilité dans un contexte d'utilisation des outils informatiques. Nous avons détaillé les problèmes propres au calcul intensif. Ensuite, nous avons présenté les différentes solutions déjà existantes pour répondre à ses problèmes. Un tel état de l'art devrait permettre aux chercheurs de pouvoir s'informer et mesurer l'enjeu des problématiques de la recherche reproductible, sans avoir besoin de trop fouiller. Nous avons ensuite fait des propositions mises en œuvre sur différents cas d'études de reproductibilité avec des applications concrètes. Cela nous a permis d'assoir la fiabilité des solutions mises en oeuvre. Nous avons par exemple proposé un modèle de simulation Covid permettant la reproductibilté des résultats numériques (et la répétabilité sur la même machine dans un cadre expérimental identique). Ce travail a été réalisé après avoir constaté le manque d'accessibilité des modèles déjà existant. De plus, lors d'une collaboration nationale avec certains collègues chercheurs, nous avons constaté que pour l'implémentation d'un modèle épidémiologique très simple, le fait d'avoir différents développeurs et différentes technologies va entrainer des résultats de simulations statistiquement différents, même si l'intention était d'obtenir les mêmes résultats statistiques. Cela montre parfois la difficulté de l'obtention de la reproductibilité (par une équipe différente) et de la réplicabilité (avec également une expérience différente), et ce malgré des efforts dans ce sens. Toujours dans la partie logicielle, nous avons étudié la qualité statistique des états du générateur Mersenne Twister lors de son utilisation pour des simulations parallèles. Nous avons trouvé que dans plus de 30% des cas, le flux de nombre aléatoire généré rate plus de deux tests de la batterie de test TestU01 ; or nous avions trouvé cette information nulle part ailleurs. Concernant le générateur Mersenne Twister, après expérimentation, nous apportons une recommandation sur l'utilisation en ce qui concerne son initialisation. La méthode fournit par les auteurs, qui est légèrement plus fiable statistiquement que les autres techniques, de plus elle est rapide. Nous avons également étudié la reproductibilité des cadriciels (frameworks) et des bibiothèques de machine learning. Nous avons constaté la difficulté d'obtention de résultats ne serait-ce que répétable pour le débogage. Ceci était dû à la difficulté de gestion des différentes sources pseudo-aléatoires utilisées pour les différents algorithmes mis en œuvre en machine learning. Nous apportons des

informations pratiques sur les algorithmes de PRNG (Pseudo random number generator, générateur de nombres pseudo-aléatoires) implémentés dans les principaux outils de machine learning, ainsi que sur leurs performances et leur qualité. Par exemple, dans une question de reproductibilité, nous devrions nous attendre à obtenir la même qualité statistique pour un même générateur implémenté dans différentes technologies. Néanmoins, ce n'est pas si évident, des études plus approfondies pour chaque générateur seront utiles. Dans le cadre de simulations de physique quantique, nous avons également étudié la reproductibilité et les performances des compilateurs et des bibliothèques scientifiques. Nous avons apporté des informations utiles pour les physiciens, avec des soucis de répétabilité avec la bibliothèque FFTW (Fastest Fourier Transform in the West), mais qui peuvent être négligeables selon le type d'application. Le but est principalement de sensibiliser nos collègues scientifiques à la problématique de la reproductibilité, et ensuite d'établir les solutions au cas par cas, suivant le contexte. Nous nous sommes également intéressés à la reproductibilité concernant le matériel physique, que ce soit en termes de résultat ou en terme de temps de calcul. Nous avons étudié l'impact sur les temps de calcul du Simultaneous Multi-Threading (SMT), selon le type d'application, et nous avons montré que le bénéfice que nous pouvons en tirer n'en pas systématique, et que cela dépend de la configuration matérielle global de la machine. Si le goulot d'étranglement se situe au niveau de l'utilisation de la mémoire, alors l'utilisation du SMT pourra dégrader les performances. Enfin, nous nous sommes interessé à la nouvelle technologie qui pourrait révolutionner le calcul haute performance : l'informatique quantique. Nous avons utilisé l'algorithme de Grover pour comparer les résultats sur des machines quantiques simulés, et sur des machines quantiques réelles. L'objectif était de voir si la fiabilité des machines quantiques actuelles permettait de réaliser des expériences reproductibles. Nous avons constaté que ce n'est pas encore vraiment le cas, mais que néanmoins des progrès rapides sont à espérer.

V.2. Problèmes ouverts et perspectives

V.2.1. Quelques problèmes ouverts

V.2.1.1. Reproductibilité des données dans le cadre du « Big data »

La prolifération des données volumineuses dans le monde du calcul à hautes performances introduit des défis variés en matière de reproductibilité des données. À mesure que les ensembles de données deviennent de plus en plus volumineux, les systèmes de gestion des versions de données conventionnels peinent à suivre le rythme, ce qui entraîne d'importants

obstacles à la maintenance d'un environnement de données consistant et reproductible. Il s'agit non seulement d'un problème technique, mais aussi méthodologique, où la nécessité de gérer de manière évolutive et efficace ces ensembles de données volumineux est primordiale. De plus, avec l'accent de plus en plus mis sur la science basée sur les données, il est impératif d'établir des pratiques robustes de partage de données adaptées à l'échelle et à la complexité immenses des écosystèmes de données volumineuses du HPC. Ces pratiques doivent non seulement faciliter le partage, mais aussi garantir l'intégrité et la reproductibilité des données. La mise à jour de bases de données volumineuses peut entraîner une perte de reproductibilité, et il est difficile de gérer l'archivage de toutes les versions d'une telle quantité de données. Aborder ces défis est essentiel pour permettre la découverte scientifique collaborative et maintenir la crédibilité de la recherche computationnelle. A titre d'exemple, lors de nos expériences sur le modèle épidémiologique que nous avons développé, nous avons généré 220Go de données (nous sommes très loin du Big Data), et pourtant, les plateformes d'archivage comme Zenodo proposent un stockage à hauteur maximale de 50Go. Il est donc nécessaire de trouver des solutions afin de pouvoir partager ces données lors de la publication de l'article.

V.2.1.2. Reproductibilité des temps de calculs et optimisations

La quête de l'obtention toujours plus rapide de résultats dans le monde du calcul haute performance conduit souvent à un compromis avec la reproductibilité, en particulier lorsque nous sommes en présence de systèmes hautement parallèles et distribués. Les techniques d'optimisation qui améliorent les performances, telles que l'adaptation du code à des architectures spécifiques ou l'exploitation du parallélisme, peuvent rendre les résultats moins reproductibles. Cette dichotomie pose un problème ouvert critique : comment pouvons-nous équilibrer la recherche de performances maximales avec l'assurance de résultats consistant ? C'est un acte délicat d'équilibrage qui nécessite une compréhension plus approfondie de l'interaction entre l'architecture du système, les stratégies d'optimisation et la nature des tâches. Les cas donnés dans notre état de l'art étaient par exemple, l'utilisation de la fonctionnalité « Fused Multiply-Add » (FMA) ou des « Advanced vector extensions » (AVX), l'utilisation de GP-GPU (General-Purpose Graphics Processing Unit) ou encore des unités de calcul tensoriel, ces utilisations peuvent entraîner une perte de reproductibilité en HPC. Sur les clusters et les supercalculateurs, cela suppose par exemple de désactiver les instructions AVX et/ou FMA ainsi que d'autres « optimisations ». La recherche dans ce domaine est active et le développement de nouvelles méthodes qui peuvent garantir des résultats reproductibles sans

sacrifier trop les performances fera le bonheur des utilisateurs de systèmes de calculs à haute performance.

V.2.1.3. Education, collaboration et intégration

L'intégration de la reproductibilité au cœur même de la méthode scientifique est un défi qui n'entre pas seulement dans la case des solutions techniques, cela touche aussi aux aspects éducatifs et collaboratifs. Il est impératif d'inculquer les principes de la recherche reproductible au sein des programmes éducatifs, pas seulement au niveau du doctorat, mais également en Master, en école d'ingénieur et même au niveau Licence. Il faut éduquer une nouvelle génération de scientifiques qui intègrent naturellement ces pratiques dans leur travail. Au-delà du monde universitaire, promouvoir des normes de collaboration est essentiel pour développer et maintenir des « workflows » scientifiques reproductibles. Cela concerne non seulement le chercheur individuel, mais l'ensemble de l'écosystème scientifique, y compris les éditeurs de journaux scientifiques, les financeurs et les institutions. L'objectif est de créer une culture scientifique où la reproductibilité ne soit pas une réflexion après coup, mais que cela devienne un composant fondamental du processus scientifique, de l'acquisition des données à l'analyse.

V.2.2. Quelques perspectives concrètes

V.2.2.1. Etude de l'importance de la qualité statistique des PRNGs sur les résultats d'entrainement de réseaux de neurones

Comme nous l'avons vu au cours de cette thèse, les générateurs de nombres pseudo-aléatoires (PRNGs) jouent un rôle crucial dans les algorithmes de machine learning, notamment avec des techniques telles que le dropout ou la descente de gradient stochastique pour l'entraînement des réseaux de neurones. Une direction intéressante à explorer plus en profondeur serait de déterminer dans quelle mesure la qualité statistique des PRNGs et leur parallélisation influence la performance et la robustesse des résultats d'entraînement.

V.2.2.2. Etude de la non-reproductibilité numérique entres les PRNGs sur différentes technologies (langages, cadriciels)

Un autre axe de recherche consiste à investiguer les différences de reproductibilité numérique entre les PRNGs utilisés sur diverses technologies (langages, API, cardriciels, ...). Les problèmes observés pourraient provenir de la fonction d'initialisation (souvent nommée seeding pour des raisons historiques) ou encore d'autres facteurs techniques. Mener une analyse approfondie permettrait de mieux cerner les causes sous-jacentes des variations dans les

résultats d'exécution, ouvrant ainsi la voie à des solutions pour garantir une plus grande cohérence entre les implémentations.

V.2.2.3. Etude plus approfondie des qualités statistiques de PCG et Philox

Lors de notre analyse du générateur Mersenne Twister (Antunes et al., 2023), appliquée sur 4096 flux stochastiques indépendants, nous avons observé que ce dernier échoue au moins une fois à tous les tests de la batterie Big Crush (sur les 4096 flux), tandis qu'il est principalement connu pour échouer uniquement aux deux tests de « Linear Complexity », qui constituent presque marqueur pour repérer ce générateur (qui est connu pour ses faiblesses en cryptographie). PCG et Philox sont supposés réussir tous les tests de Big Crush tout en offrant de bonnes performances. Toutefois, nos résultats préliminaires suggèrent que ces générateurs pourraient ne pas être aussi fiables que prévu, ce que signale les travaux en ligne de Vigna (homepage). Une étude approfondie permettrait de clarifier ces résultats et de vérifier la robustesse de ces générateurs dans différents contextes.

V.2.2.4. Etude de la reproductiblité numérique et des performances entre différents langages de programmation

Enfin, une étude amorcée durant cette thèse, mais non achevée faute de temps, porte sur les différences de reproductibilité numérique et de performance lors de l'implémentation des mêmes algorithmes dans différents langages de programmation. Nous avons aussi commencé en parallèle avec ces études sur la performance, une étude sur la consommation énergétique (même si les deux sont liées).

VI. Bibliographie

Abbasi, K. (2020). Covid-19 : Politicisation, “corruption,” and suppression of science. *British Medical Journal Publishing Group*. 2 p.

Abraham, S., Paul, A.K., Khan, R.I.S., Butt, A.R. (2020). On the use of containers in high performance computing environments. *International Conference on Cloud Computing (CLOUD)*, 284–293.

Acharya, A., Fanguède, J., Paolino, M., Raho, D. (2018). A performance benchmarking analysis of hypervisors containers and unikernels on ARMv8 and x86 CPUs. *European Conference on Networks and Communications (EuCNC)*, 282–9.

Alcott, B. (2005). Jevons’ paradox. *Ecological economics*, 54(1), 9-21.

Aleta, A., Martin-Corral, D., Pastore y Piontti, A., Ajelli, M., Litvinova, M., Chinazzi, M., Dean, N. E., Halloran, M. E., Longini Jr, I. M., & Merler, S. (2020). Modelling the impact of testing, contact tracing and household quarantine on second waves of COVID-19. *Nature Human Behaviour*, 4(9), 964-971.

Allaire, J. (2012). RStudio: integrated development environment for R. *The R User Conference, user!*, p. 14

Altekar, G., Stoica, I. (2009). ODR: Output-deterministic replay for multicore debugging. *ACM SIGOPS symposium on Operating systems principles*, 193–206.

Altintas, I., Berkley, C., Jaeger, E., Jones, M., Ludascher, B., Mock, S. (2004). Kepler: an extensible system for design and execution of scientific workflows. *International Conference on Scientific and Statistical Database Management*, 423–424.

Amdahl, G.M. (1967). Validity of the single processor approach to achieving large scale computing capabilities. *spring joint computer conference*, 483-485.

Amstutz, P., Crusoe, M.R., Tijanić, N., Chapman, B., Chilton, J., Heuer, M., Kartashov, A., Leehr, D., Ménager, H., Nedeljkovich, M. (2016). Common workflow language, *v1.0 eScholarship, University of California*, 22 p.

Amstutz, P., Mikheev, M., Crusoe, M.R., Tijanić, N., Lampa, S. (2024). Existing Workflow systems. , 25 Janvier 2024. [En ligne]. Disponible à l’adresse : https ://s.apache.org/existing-workflow-systems [Consulté le 19 Mars 2024].

Antunes B., Hill, D.R.C. (2024). Reproducibility, Replicability and Repeatability: A survey of reproducible research with a focus on high performance computing, *Computer Science Review*, 53, 100655, 28 p.

Antunes, B., Hill, D.R.C. (2024). Reproducibility, energy efficiency and performance of pseudorandom number generators in machine learning: a comparative study of python, numpy, tensorflow, and pytorch implementations. *arXiv preprint arXiv:2401.17345*, 20 p.

Antunes, B., Mazel, C., Lacomme, P., Hill, D.R.C. (2024). Test de reproductibilité de l'algorithme de Grover sur simulateur et sur machines quantiques IBM. *Roadef*, 4 p.

Antunes B., Hill D.R.C. (2024). Recherche Reproductible : Comment les outils informatiques et le calcul scientifique impactent bien des disciplines, Atramentra-Stylit-Hachette, ISBN 978-952-390-749-2, 142 p.

Antunes, B., Mazel, C., Hill, D.R.C. (2024). Performance and reproducibility assessment of quantum dissipative dynamics framework: a comparative study of Fortran compilers, MKL, and FFTW. *Hal, archive ouverte*, 17 p.

Antunes, B., Mazel, C., Hill, D. R. C. (2023). Identifying quality mersenne twister streams for parallel stochastic simulations. *Winter Simulation Conference (WSC)*, 2801-2812.

Antunes, B., Hill, D.R.C. (2023). Evaluating Simultaneous Multi-threading and Affinity Performance for Reproducible Parallel Stochastic Simulation. *Research Reports on Computer Science*, 91-110.

Antunes, B., Hill, D.R.C. (2022). Utilisation de l'intelligence artificielle distribuée pour des simulations reproductibles de l'épidémie de Covid19. *Giseh*, 8 p.

ARB. (2020). "Artifact Review and Badging Version 1.1, August 24, 2020", Consulté le 22 Janvier 2024, URL : https://www.acm.org/publications/policies/artifact-review-and-badging-current

Aupy, G., Benoit, A., Hérault, T., Robert, Y., Vivien, F., Zaidouni, D. (2013). On the combination of silent error detection and checkpointing. *Pacific Rim International Symposium on Dependable Computing*, 11–20.

Axhausen, K. W. (2021). Modelling COVID19 using an agent-based-model of travel. *Institute of Space and Earth Information Science (ISEIS) seminar.*

Aylett-Bullock, J., Cuesta-Lazaro, C., Quera-Bofarull, A., Icaza-Lizaola, M., Sedgewick, A., Truong, H., Curran, A., Elliott, E., Caulfield, T., Fong, K. (2021). June : Open-source individual-based epidemiology simulation. *Royal Society open science*, 8(7), 210506.

Baier, T., Neuwirth, E. (2007). Excel : : Com. *Computational statistics*, 22(1), 91–108.

Bajpai, V., Bonaventure, O., Claffy, K., Karrenberg, D. (2019). Encouraging reproducibility in scientific research of the internet. *Dagstuhl reports*, 8(10).

Bajpai, V., Brunstrom, A., Feldmann, A., Kellerer, W., Pras, A., Schulzrinne, H., Smaragdakis, G., Wählisch, M., Wehrle, K. (2019). The Dagstuhl beginners guide to reproducibility for experimental networking research. *ACM SIGCOMM Computer Communication Review*, 49(1), 24–30.

Bajpai, V., Kühlewind, M., Ott, J., Schönwälder, J., Sperotto, A., Trammell, B. (2017). Challenges with reproducibility. *Reproducibility Workshop*, 1–4.

Baker, M. (2016). Reproducibility crisis. *Nature*, 533(26), 353–66.

Bakita, J., Ahmed, S., Osborne, S.H., Tang, S., Chen, J., Smith, F.D., Anderson, J.H. (2021). Simultaneous multithreading in mixed-criticality real-time systems. *Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 278-291.

Bansal, S., Grenfell, B. T., Meyers, L. A. (2007). When individual behaviour matters : Homogeneous and network models in epidemiology. *Journal of the Royal Society Interface*, 4(16), 879-891.

Barba, L.A. (2018). Terminologies for reproducible research. *arXiv preprint arXiv* :1802.03311, 18 p.

Bauch, C. T. (2021). Estimating the COVID-19 R number : A bargain with the devil? *The Lancet infectious diseases*, 21(2), 151-153.

Baumann, R.C. (2005). Radiation-induced soft errors in advanced semiconductor technologies. *IEEE Transactions on Device and materials reliability*, 5(3), 305–316.

Baumer, B., Cetinkaya-Rundel, M., Bray, A., Loi, L., Horton, N.J. (2014). R Markdown : Integrating a reproducible analysis tool into introductory statistics. *arXiv preprint arXiv* :1402.1894, 30 p.

Begley, C.G., Ellis, L.M. (2012). Raise standards for preclinical cancer research. *Nature*, 483(7391), 531–533.

Benedicic, L., Cruz, F.A., Madonna, A., Mariotti, K. (2019). Sarus: Highly scalable docker containers for hpc systems. *High Performance Computing: ISC High Performance*, 46–60.

Benson, A.R., Schmit, S., Schreiber, R. (2015). Silent error detection in numerical time-stepping schemes. *The International Journal of High Performance Computing Applications*, 29(4), 403–421.

Bernal, M. A., Bordage, M. C., Brown J.M.C., Davídková, M., Delage, E., El Bitar, Z., Enger, S.A., Francis, Z., Guatelli, S., Ivanchenko, V.N., Karamitros, M., Kyriakou, I., Maigne, L., Meylan, S., Murakami, K., Okada, S., Payno, H., Perrot, Y., Petrovic, I., Pham, Q.T., Ristic-Fira, A., Sasaki, T., Štěpán, V., Tran, H.N., Villagrasa, C., Incerti, S. (2015). Track Structure Modeling in Liquid Water: A Review of the Geant4-DNA Very Low Energy Extension of the Geant4 Monte Carlo Simulation Toolkit. *Physica Medica,* 31(8), 861-874.

Beserra, D., Oliveira, F., Araujo, J., Fernandes, F., Araújo, A., Endo, P., Maciel, P., Moreno, E.D. (2015). Performance evaluation of hypervisors for hpc applications. *International Conference on Systems, Man, and Cybernetics,* 846–851.

Bhattacharjee, K., Das, S. (2022). A Search for Good Pseudo-Random Number Generators. Survey and Empirical Studies. *Computer Science Review,* 45, p. 100471

Bisgambiglia, P. A., Hill, D.R.C. (2022). Reproductibilité numérique: enjeux de crédibilité pour les expériences de simulation. *BULLETIN de la société informatique de France*, 1024, 137-144.

Blackman, D., Vigna, S. (2021). Scrambled linear pseudorandom number generators. *ACM Transactions on Mathematical Software (TOMS)*, 47(4), 1–32.

Blanchard, P., Higham, N.J., Mary, T. (2020). A class of fast and accurate summation algorithms. *SIAM journal on scientific computing*, 42(3), A1541–A1557.

Boettiger, C. (2015). An introduction to Docker for reproducible research. *ACM SIGOPS Operating Systems Review*, 49(1), 71–79.

Bononi, L., Bracuto, M., D'Angelo, G., Donatiello, L. (2006). Exploring the Effects of Hyper-threading on Parallel Simulation. *International Symposium on Distributed Simulation and Real-Time Applications*, 257-260.

Booch, G., Jacobson, I., Rumbaugh, J. (1996). The unified modeling language. *Unix Review*, 14(13), 30 p.

Bordage, C., Jeannot, E. (2018). Process Affinity, Metrics and Impact on Performance: An Empirical Study. *International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 523-532.

Borthakur, D. (2007). The hadoop distributed file system : Architecture and design. *Hadoop Project Website*, 14 p.

Bottou, L. (2003). Stochastic learning. *Summer School on Machine Learning*, 146-168.

Bottou, L. (2010). Large-scale machine learning with stochastic gradient descent. *International Conference on Computational Statistics*, 177-186.

Boyer, A.F., Haen, C., Stagni, F., Hill, D.R.C. (2022). Porting DIRAC Benchmark to Python3: impact of the discrepancies and solutions. *International Conference on High Energy physics*, 4 p.

Boyer, A.F. (2022a). Contributions to Computing needs in High Energy Physics Offline Activities: Towards an efficient exploitation of heterogeneous, distributed and shared Computing Resources. Thèse de doctorat. CERN – Université Clermont-Auvergne.

Boyer, A., Haen, C., Stagni F., Hill, D.R.C. (2022b). DIRAC Site Director: Improving Pilot-Job Provisioning on Grid Resources. *Future Generation Computer Systems,* 133, 23-38.

Boyer, A.F., Haen, C., Stagni, F., Hill, D.R.C. (2022c). Pilot-job provisioning on grid resources: Collecting analysis and performance evaluation data. *Data in Brief*, 42, p. 108104.

Brauer, F. (2005). The Kermack–McKendrick epidemic model revisited. *Mathematical biosciences*, 198(2), 119-131.

Bronevetsky, G., de Supinski, B. (2008). Soft error vulnerability of iterative linear algebra methods. *Annual international conference on Supercomputing*. 155–164.

Broquedis, F., Clert-ortega J., Moreaud S., Furmento, N., Goglin, B., Mercier, G., Thibault, S., Namyst, R. (2010). hwloc: A generic framework for managing hardware affinities in HPC applications. *Euromicro Conference on Parallel, Distributed and Network-based Processing*, 180-186.

Brown, R. G., D. Eddelbuettel, and D. Bauer. (2013). Dieharder: A Random Number Test Suite. Open Source Software Library, Under Development. https://webhome.phy.duke.edu/~rgb/General/dieharder.php, accès le 05 août 2023.

Buckheit, J.B., Donoho, D.L. (1995). Wavelab and reproducible research. *Springer New York*, 27 p.

Budanur, S., Mueller, F., Gamblin, T. (2011). Memory trace compression and replay for spmd systems using extended prsds ? *ACM SIGMETRICS Performance Evaluation Review*, 38(4), 30–36.

Bulpin, R.J., Pratt, I.A. (2004). Multiprogramming Performance of the Pentium 4 with Simultaneous Multi-threading. *Annual Workshop on Duplicating, Deconstruction and Debunking (WDDD)*, 53-62.

Button, K.S., Munafò, M.R. (2017). Powering reproducible research. *Psychological science under scrutiny : Recent challenges and proposed solutions*, 22–33.

Casalicchio, E., Perciballi, V. (2017). Measuring docker performance: What a mess!!!. *ACM/SPEC on International Conference on Performance Engineering Companion*. 11–16.

Casas, M., de Supinski, B.R., Bronevetsky, G., Schulz, M. (2012). Fault resilience of the algebraic multi-grid solver. *ACM international conference on Supercomputing*. 91–100.

Castro, B. M., de Melo, Y. de A., Dos Santos, N. F., da Costa Barcellos, A. L., Choren, R., Salles, R. M. (2021). Multi-agent simulation model for the evaluation of COVID-19 transmission. *Computers in Biology and Medicine*, 136, 104645.

Cazenave, T., Jouandeau, N. (2009). Parallel nested monte-carlo search. *International Symposium on Parallel & Distributed Processing*, 1-6.

Celebioglu O, Saify A, Leng T, Hsieh J, Mashayekhi V, Rooholamini R. (2004). The performance impact of computational efficiency on HPC clusters with hyper-threading technology. *International Parallel and Distributed Processing Symposium*, 250

Chae, M., Lee, H., Lee, K. (2019). A performance comparison of linux containers and virtual machines using Docker and KVM. *Cluster Computing*, 22(1), 1765–1775.

Chang, S.L., Harding, N., Zachreson, C., Cliff, O. M., Prokopenko, M. (2020). Modelling transmission and control of the COVID-19 pandemic in Australia. *Nature communications*, 11(1), 5710.

Chao, D. L., Halloran, M. E., Obenchain, V. J., Longini Jr, I. M. (2010). FluTE, a publicly available stochastic influenza epidemic simulation model. *PLoS computational biology*, 6(1), e1000656.

Chapp, D., Sato, K., Ahn, D.H., Taufer, M. (2018). Record-and-replay techniques for HPC systems: A survey. *Supercomputing Frontiers and Innovations*, 5(1), 11–30.

Charpilloz, C., Arteaga, A., Fuhrer, O., Harrop, C., & Thind, M. (2017). Reproducible climate and weather simulations : An application to the cosmo model. *Platform for Advanced Scientific Computing (PASC) Conference*.

Chen, Y., Shi, Q., Li, X. CSSMT: Compiler Based Software Simultaneous Multithreading (SMT). (2018). *Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP),* 60-67.

Chirigati, F., Rampin, R., Shasha, D., Freire, J. (2016). Reprozip: Computational reproducibility with ease. *International conference on management of data*. 2085–2088.

Chitlur, N., Srinivasa, G., Hahn, S., Gupta, P.K., Reddy, D., Koufaty, D., Brett, P., Prabhakaran, A., Zhao, L., Ijih, N. (2012). QuickIA: Exploring heterogeneous architectures on real prototypes. *International Symposium on High-Performance Comp Architecture*, 1–8.

Chohra, C., Langlois, P., Parello, D. (2016). Reproducible, accurately rounded and efficient BLAS. *European Conference on Parallel Processing*, 609–620.

Chung, M.T., Quang-Hung, N., Nguyen, M.-T., Thoai, N. (2016). Using docker in high performance computing applications. *International Conference on Communications and Electronics (ICCE)*, 52–57.

Claerbout, J.F., Karrenbach, M. (1992). Electronic documents give reproducible research a new meaning. *Society of Exploration Geophysicists*, 601–604.

Clayson, P.E., Carbine, K.A., Baldwin, S.A., Larson, M.J. (2019). Methodological reporting behavior, sample sizes, and statistical power in studies of event-related potentials: Barriers to reproducibility and replicability. *Psychophysiology*, 56(11), e13437.

Clerk, A., Devoret, M., Girvin, S., Marquardt, F., Schoelkopf, R.J. (2010) Introduction to quantum noise, measurement, and amplification. *Reviews of Modern Physics*, 82(2), 1155.

Click, T.H., Lui A., Kaminski G.A. (2011). Quality of Random Number Generators Significantly Affects Results of Monte Carlo Simulations for Organic and Biological Systems. *Journal of Computational Chemistry,* 32(3), 513-524.

Cluzel, T., Mazel C., Hill D.R.C. (2019). Quantum Computing: a Short Introduction. LIMOS UMR CNRS 6158, Research Report RR-2019-03778746, Université Clermont Auvergne, 10 p.

Cohen-Boulakia, S., Belhajjame, K., Collin, O., Chopard, J., Froidevaux, C., Gaignard, A., Hinsen, K., Larmande, P., Le Bras, Y., Lemoine, F. (2017). Scientific workflows for computational reproducibility in the life sciences: Status, challenges and opportunities. *Future Generation Computer Systems*, 75, 284–298.

Collaboration, Open Science. (2012). An open, large-scale, collaborative effort to estimate the reproducibility of psychological science. *Perspectives on Psychological Science*, 7(6), 657–660.

Collberg, C., Proebsting, T. (2016). Repeatability in Computer Systems Research. *Communications of the ACM*. 59(3), 62-69.

Collberg, C., Proebsting, T., Warren, A.M. (2015). Repeatability and benefaction in computer systems research. *University of Arizona TR*, 14(4).

Collins, F.S., Tabak, L.A. (2014). Policy: NIH plans to enhance reproducibility. *Nature*, 505(7485), 612–613.

Congo, F.Y.P., Traore, M.K., Hill, D.R.C. (2018). Computation operations caching for numerical repeatability, *Summer Simulation Multi-Conference,* 370-381.

Courtès, L., Wurmus, R. (2015). Reproducible and user-controlled software environments in HPC with Guix. *Euro-Par International Workshops*. 579–591.

Daemen, J., Rijmen V. (2001). The Advanced Encryption Standard. *Dr. Dobb's Journal: Software Tools for the Professional Programmer,* 26(3), 137-139.

Dao, V.T., Nguyen H.Q., Maigne L., Breton V., Hill D.R.C. (2014). Numerical Reproducibility, Portability and Performance of Modern Pseudo Random Number Generators: Preliminary

Study for Parallel Stochastic Simulations Using Hybrid Xeon Phi Computing Processors. *European Simulation and modelling conference*, 80-87.

Dasgupta, S., Humble, T.S. (2021a). Reproducibility in quantum computing. *Computer Society Annual Symposium on VLSI (ISVLSI)*, 458–461.

Dasgupta, S., Humble, T.S. (2021b). Stability of noisy quantum computing devices. *arXiv preprint arXiv :2105.09472*, 8 p.

Dasgupta, S., Humble, T.S. (2022). Characterizing the reproducibility of noisy quantum circuits. *Entropy*, 24(2), 244.

David, H., Gorbatov, E., Hanebutte, U. R., Khanna, R., Le, C. (2010). RAPL: Memory power estimation and capping. *International symposium on Low power electronics and design*, 189-194.

Davids, A., Rand, G. du, Georg, C.-P., Koziol, T., Schasfoort, J. (2020). SABCoM: A spatial agent-based COVID-19 model. *medRxiv*, 2020.07.30.20164855, 56 p.

Davison, A. (2012). Automated capture of experiment context for easier reproducibility in computational research. *Computing in Science & Engineering*, 14(4), 48–56.

Deelman, E., Blythe, J., Gil, Y., Kesselman, C., Mehta, G., Patil, S., Su, M.-H., Vahi, K., Livny, M. (2004). Pegasus: Mapping scientific workflows onto the grid. *Grid Computing: Second European AcrossGrids Conference, AxGrids*, 11–20.

Delescluse, M., Franconville, R., Joucla, S., Lieury, T., Pouzat, C. (2012). Making neurophysiological data analysis reproducible: Why and how?. *Journal of Physiology*, 106(3–4), 159–170.

Demmel, J., Nguyen, H.D. (2013a). Fast reproducible floating-point summation. *Symposium on Computer Arithmetic*, 163–172.

Demmel, J., Nguyen, H.D. (2013b). Numerical reproducibility and accuracy at exascale. *Symposium on Computer Arithmetic*, 235–237.

Demmel, J., Nguyen, H.D. (2014). Parallel reproducible summation. *Transactions on Computers*, 64(7), 2060–2070.

Desquilbet, L., Granger, S., Hejblum, B., Legrand, A., Pernot, P., Rougier, N.P., de Castro Guerra, E., Courbin-Coulaud, M., Duvaux, L., Gravier, P. (2019). Vers une recherche reproductible : Faire évoluer ses pratiques. *Unité régionale de formation à l'information scientifique et technique de Bordeaux*., 161 p.

Desvillechabrol, D., Legendre, R., Rioualen, C., Bouchier, C., Van Helden, J., Kennedy, S., Cokelaer, T. (2018). Sequanix: a dynamic graphical interface for Snakemake workflows. *Bioinformatics*, 34(11), 1934–1936.

Di Cosmo, R., Zacchiroli, S. (2017). Software heritage: Why and how to preserve software source code. *International Conference on Digital Preservation*. 1–10.

Di Tommaso, P., Chatzou, M., Baraja, P.P., Notredame, C. (2014). A novel tool for highly scalable computational pipelines. *Figshare*, 1 p. https://scholar.archive.org/work/yzg3p7mf5fc5ticwhqbsvh7uly/access/wayback/https://s3-eu-west-1.amazonaws.com/pfigshare-u-files/1810568/Nextflowposterver3.pdf

Dinh, P.M., Vincendon, M., Coppens, F., Suraud, E., Reinhard, P.G. (2022). Quantum Dissipative Dynamics (QDD) : A real-time real-space approach to far-off-equilibrium dynamics in finite electron systems. *Computer Physics Communications*, 270, p. 108155.

Dixit, H.D., Pendharkar, S., Beadon, M., Mason, C., Chakravarthy, T., Muthiah, B., Sankar, S. (2021). Silent data corruptions at scale. *arXiv preprint arXiv* :2102.11245, 8 p.

Dolstra, E., De Jonge, M., Visser, E. (2004). Nix: A Safe and Policy-Free System for Software Deployment. *LISA*. 79–92.

Donoho, D.L., Stodden, V. (2015). Reproducible research in the mathematical sciences. *Princeton University Press*, 916-925.

Đorđević, B., Timčenko, V., Pavlović, O., Davidović, N. (2021). Performance comparison of native host and hyper-based virtualization VirtualBox. *International Symposium Infoteh-Jahorina (Infoteh)*. 1–4.

Drummond C. (2009). Replicability is not reproducibility: nor is it good science. *Evaluation Methods for Machine Learning Workshop*, 4 p.

Drummond, C. (2018). Reproducible research : a minority opinion. *Journal of Experimental & Theoretical Artificial Intelligence*, 30(1), 1–11.

Drummond, C. (2019). Is the drive for reproducible science having a detrimental effect on what is published ?. *Learned Publishing*, 32(1), 63–69.

Du, P., Luszczek, P., Dongarra, J. (2012). High performance dense linear system solver with resilience to multiple soft errors. *Procedia Computer Science*, 9, 216–225.

Eggers, S.J., Emer, J.S., Levy, H.M., Lo, J.L., Stamm, R.L., Tullsen, D.M. (1997). Simultaneous multithreading: A platform for next-generation processors. *IEEE micro,* 17(5), 12-19.

Eklund, A., Nichols, T.E., Knutsson, H. (2016). Cluster failure: Why fMRI inferences for spatial extent have inflated false-positive rates. *National academy of sciences*, 113(28), 7900–7905.

Elliott, J., Mueller, F., Stoyanov, F., Webster, C. (2013). Quantifying the impact of single bit flips on floating point arithmetic. Rapport, *North Carolina State University. Dept. of Computer Science*, 13 p.

Errington, T.M., Mathur, M., Soderberg, C.K., Denis, A., Perfito, N., Iorns, E., Nosek, B.A. (2021). Investigating the replicability of preclinical cancer biology. *Elife*, 10, e71601.

Fanelli, D. (2010). Do pressures to publish increase scientists' bias ? An empirical support from US States Data. *PloS one*, 5(4), e10271.

Fanelli, D. (2018). Is science really facing a reproducibility crisis, and do we need it to ?. *National Academy of Sciences*, 115(11), 2628–2631.

Fatès, N., Chevrier, V. (2010). How important are updating schemes in multi-agent systems ? An illustration on a multi-turmite model. *International Conference on Autonomous Agents and Multiagent Systems*, 533-540.

Felter, W., Ferreira, A., Rajamony, R., Rubio, J. (2015). An updated performance comparison of virtual machines and linux containers. *International symposium on performance analysis of systems and software (ISPASS)*, 171–172.

Ferguson, N. M., Laydon, D., Nedjati-Gilani, G., Imai, N., Ainslie, K., Baguelin, M., Bhatia, S., Boonyasiri, A., Cucunubá, Z., Cuomo-Dannenburg, G. (2020). Impact of non-pharmaceutical interventions (NPIs) to reduce COVID-19 mortality and healthcare demand. *Imperial College COVID-19 Response Team*, 20(10.25561), 77482.

Févotte, F., Lathuilière, B. (2016). VERROU : a CESTAC evaluation without recompilation. *Séminaire de Conception Architecturale Numérique (SCAN),* 12 p.

Feynmann, R. (1982). Simulating Physics with Computers, Physics and Computation. *International Journal of Theoretical Physics,* 21, 467–488.

Fiala, D., Mueller, F., Engelmann, C., Riesen, R., Ferreira, K., Brightwell, R. (2012). Detection and correction of silent data corruption for large-scale high-performance computing. *International Conference on High Performance Computing, Networking, Storage and Analysis*, 1–12.

Figgatt, C., Maslov, D., Landsman, K.A., Linke, N.M., Debnath, S., Monroe, C. (2017). Complete 3-Qubit Grover search on a programmable quantum computer. *Nat Commun,* 8, 1918, 9 p.

Filippi, J.-B., Morandini, F., Balbi, J. H., Hill, D.R.C. (2010). Discrete event front-tracking simulation of a physical fire-spread model. *Simulation*, 86(10), 629-646.

Foong, A., Fung, J., Newell, D. (2004). An In-Depth Analysis of the Impact of Processor Affinity on Network Performance. *International Conference on Networks (ICON)*, 244-250.

Forstmeier, W., Wagenmakers, E.-J., Parker, T.H. (2017). Detecting and avoiding likely false-positive findings–a practical guide. *Biological Reviews*, 92(4), 1941–1968.

Franceschini, R., Bisgambiglia, P.-A., Hill, D.R.C. (2018). Reproducibility study of a PDEVS model application to fire spreading. *Computer Simulation Conference*, 1-11.

Frigo, M., Johnson, S. G. (2005). The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2), 216-231.

Gajardo, A., Moreira, A., Goles, E. (2002). Complexity of Langton's ant. *Discrete Applied Mathematics,* 117(1-3), 41-50.

Gambron, P., Thorne, S. (2020). Comparison of several FFT libraries in C/C++. *STFC*, 32 p. https://epubs.stfc.ac.uk/work/45434573

Gampe, J., Zinn, S., Willekens, F., van der Gaag, N., de Beer, J., Himmelspach, J., Uhrmacher, A. (2009). The microsimulation tool of the MicMac project. *General Conference of the International Microsimulation Association*, 17 p.

Gantikow, H., Walter, S., Reich, C. (2020). Rootless containers with Podman for HPC. *International Conference on High Performance Computing*, 343–354.

Gaudou, B., Huynh, N. Q., Philippon, D., Brugière, A., Chapuis, K., Taillandier, P., Larmande, P., Drogoul, A. (2020). Comokit : A modeling kit to understand, analyze, and compare the impacts of mitigation policies against the covid-19 epidemic at the scale of a city. *Frontiers in public health*, 8, p. 563247.

Gelman, A., Stern, H. (2006). The difference between "significant" and "not significant" is not itself statistically significant. *The American Statistician*, 60(4), 328–331.

Gerhardt, L., Bhimji, W., Canon, S., Fasel, M., Jacobsen, D., Mustafa, M., Porter, J., Tsulaia, V. (2017). Shifter: Containers for hpc. *Journal of physics : Conference series*, 082021.

Giardine, B., Riemer, C., Hardison, R.C., Burhans, R., Elnitski, L., Shah, P., Zhang, Y., Blankenberg, D., Albert, I., Taylor, J. (2005). Galaxy: a platform for interactive large-scale genome analysis. *Genome research*, 15(10), 1451–1455.

Gilbert, L., Tseng, J., Newman, R., Iqbal, I., Pepper, R., Celebioglu, O., Hsieh, J., Cobban, M. (2005). Performance implications of virtualization and hyper-threading on high energy physics applications in a grid environment. *International parallel and distributed processing symposium*. 10 p.

Gioachin, F., Zheng, G., Kalé, L.V. (2010). Robust non-intrusive record-replay with processor extraction. *Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*. 9–19.

Goldberg, D. (1991). What every computer scientist should know about floating-point arithmetic. *ACM computing surveys (CSUR),* 23(1), 5–48.

Goldberg, R.P. (1974). Survey of virtual machine research. *Computer*, 7(6), 34–45.

Gomez, J., Prieto, J., Leon, E., Rodriguez, A. (2020). INFEKTA: a general agent-based model for transmission of infectious diseases : Studying the COVID-19 propagation in Bogotá-Colombia. *MedRxiv*, 2020.04.06.20056119, 15 p.

Goodman, S.N., Fanelli, D., Ioannidis, J. P. (2016). What does research reproducibility mean?. *Science translational medicine*, 8(341), 341ps12-341ps12.

Goralski, M. A., Tan, T. K. (2020). Artificial intelligence and sustainable development. *The International Journal of Management Education*, 18(1), 100330.

Grais, R.F., Hugh Ellis, J., Glass, G.E. (2003). Assessing the impact of airline travel on the geographic spread of pandemic influenza. *European journal of epidemiology*, 18, 1065-1072.

Guermouche, A., Ropars, T., Brunet, E., Snir, M., Cappello, F. (2011). Uncoordinated checkpointing without domino effect for send-deterministic MPI applications. *International Parallel & Distributed Processing Symposium*, 989–1000.

Gundersen, O.E. (2021). The fundamental principles of reproducibility. *Philosophical Transactions of the Royal Society A*, 379(2197), 20200210.

Gundersen, O.E., Coakley, K., Kirkpatrick, C., Gil, Y. (2022). Sources of irreproducibility in machine learning: A review. *arXiv preprint,* arXiv:2204.07610, 9 p.

Gundersen, O.E., Kjensmo, S. (2018). State of the art : Reproducibility in artificial intelligence. *AAAI Conference on Artificial Intelligence*. 32(1), 1644-1651.

Guo, P.J., Engler, D. (2011). CDE: Using System Call Interposition to Automatically Create Portable Software Packages. *USENIX Annual Technical Conference (USENIX ATC 11)*.

Halchenko, Y.O., Hanke, M. (2015). Four aspects to make science open "by design" and not as an after-thought. *GigaScience*, 4(1), s13742-015-0072–7.

Hale, J.S., Li, L., Richardson, C.N., Wells, G.N. (2017). Containers for portable, productive, and performant scientific computing. *Computing in Science & Engineering*, 19(6), 40–50.

Haramoto, H., Matsumoto, M., Nishimura, T., Panneton, F., L'Ecuyer, P. (2008). Efficient Jump Ahead For F_2-Linear Random Number Generators. *INFORMS Journal on Computing,* 20, 385–390.

Hellekalek, P. (1998). Don't trust parallel Monte Carlo ! *ACM SIGSIM Simulation Digest*, 28(1), 82–89.

Henderson, P., Islam, R., Bachman, P., Pineau, J., Precup, D., Meger, D. (2018). Deep reinforcement learning that matters. *AAAI conference on artificial intelligence,* 32(1), 3207–3214.

Herndon, T., Ash, M., Pollin, R. (2014). Does high public debt consistently stifle economic growth ? A critique of Reinhart and Rogoff. *Cambridge journal of economics*, 38(2), 257–279.

Hill, D.R.C., Antunes, B., Bertrand, A., Nguifo, E. M., Yon, L., Nautré-Domanski, J., Violaine, A. (2024). Machine learning and reproducibility impact of random numbers. 11 p.

Hill, D.R.C., Antunes, B., Cluzel, T., Mazel, C. (2023). A few words about quantum computing, epistemology, repeatability and reproducibility. *EU/ME Meeting - Quantum SchoolEmerging Optimization Methods: From Metaheuristics To Quantum Approaches*. 4 p.

Hill, D.R.C., Antunes, B. (2022). Reproductibilité et modèles Covid-un modèle multi-agents. *Journées DEVS Francophones-Convergences entre la Théorie de la Modélisation et de la Simulation et les Systèmes multi-agents*, 35-53.

Hill, D.R.C. (2022). Reproducibility of Simulations and High Performance Computing. *European Simulation and Modelling Conference*, 5-9.

Hill, D.R.C. (2019). Repeatability, reproducibility, computer science and high performance computing : Stochastic simulations can be reproducible too…. *International Conference on High Performance Computing & Simulation (HPCS)*, 322-323.

Hill, D.R.C. (2017). Numerical reproducibility of parallel and distributed stochastic simulation using high-performance computing. *Computational Frameworks*, 95-109.

Hill, D.R.C., Mazel, C., Breton, V. (2017). Reproductibilité et répétabilité numérique. Constats, conseils et bonnes pratiques pour le cas des simulations stochastiques parallèles et distribuées. *Revue des Sciences et Technologies de l'Information-Série TSI : Technique et Science Informatiques*, 36(3–6), 243–272.

Hill D.R.C. (2015). Parallel random numbers, simulation, and reproducible research. *Computing in Science & Engineering*. 17(4), 66-71.

Hill, D.R.C., Passerat-palmbach, J., Mazel, C., Traore, M.K. (2013). Distribution of Random Streams for Simulation Practitioners. *Concurrency and Computation: Practice and Experience*, 25(10), 1427-1442.

Hill, D.R.C., Barraud, R., Crozat, B., Touraille, L., Muzy, A., Leccia, F., Mérimée, V., Granval, C. (2009). Decision support system for a regional spreading of a/H1N1 influenza virus. *European Simulation and Modelling Conference*, 261-268.

Hill, D.R.C., Muzy, A., Barraud, R., Crozat, B., Madary, J., Leccia, F. (2008). Design of a spatial and stochastic simulator for bird flu spreading in Corsica. *International Simulation Multi-Conference. Grand Modeling Challenge*, 16-19.

Hinch, R., Probert, W. J., Nurtay, A., Kendall, M., Wymant, C., Hall, M., Lythgoe, K., Bulas Cruz, A., Zhao, L., Stewart, A. (2021). OpenABM-Covid19—An agent-based model for non-pharmaceutical interventions against COVID-19 including contact tracing. *PLoS computational biology*, 17(7), e1009146.

Hinsen, K. (2021). La reproductibilité des calculs coûteux. *Reproductibilité de la Recherche*. 11–14.

Hinsen, K. (2020b). Staged computation: The technique you did not know you were using. *Computing in Science & Engineering*, 22(4), 99–103.

Hinsen, K. (2020a). [Rp] Structural flexibility in proteins - impact of the crystal environment. *ReScience C*, 6(1), 9 p.

Hinsen, K. (2018). Reusable vs. re-editable code. *Computing in Science and Engineering*, 20(3), 78–83.

Hinsen, K. (2017). Enjeux et défis de la recherche reproductible. *Journée MaDICS-ReproVirtuFlow*. https://indico.mathrice.fr/event/165/contributions/299/attachments/336/395/Aramis2019_HINSEN.pdf [Consulté le 17 septembre 2024].

Hinsen, K. (2014). Reproducibility, replicability, and the two layers of computational science. [En ligne]. Disponible à l'adresse : https ://khinsen.wordpress.com/2014/08/27/reproducibility-replicability-and-the-two-layers-of-computational-science/ [Consulté le 22 Août 2023].

Hinsen, K. (2013). Software development for reproducible research. *Computing in Science & Engineering*, 15(04), 60–63.

Hoemmen, M.F., Heroux, M.A. (2011). Fault-tolerant iterative methods. Rapport, *Sandia National Lab.(SNL-NM)*. 12 p.

Hogg, T., Portnov, D. (2000). Quantum optimization. *Information Sciences*, 128(3–4), 181–197.

Holmes, J. (2020). Software engineers calling to retract papers based on this code. [En ligne]. Disponible à l'adresse : https ://github.com/mrc-ide/covid-sim/issues/165 [Consulté le 1 Janvier 2024].

Hoo, K.C., Ee, O.S., Yin, T.S. (2022). Effect of Simultaneous Multithreading Towards the Performance of Cloud Workloads. *Embedded World*. 4 p.

Howe, B. (2012). Virtual appliances, cloud computing, and reproducible research. *Computing in Science & Engineering*, 14(4), 36–41.

Hower, D.R., Hill, M.D. (2008). Rerun: Exploiting episodes for lightweight memory race recording. *ACM SIGARCH computer architecture news*, 36(3), 265–276.

Hu, G., Zhang, Y., Chen, W. (2019). Exploring the performance of singularity for high performance computing scenarios. *International Conference on High Performance Computing and Communications*, 2587–2593.

Huang, G., Sun, Y., Liu, Z., Sedra, D., Weinberger, K. Q. (2016). Deep networks with stochastic depth. *Computer Vision–ECCV European Conference*, 646-661.

Huang, K.-H., Abraham, J.A. (1984). Algorithm-based fault tolerance for matrix operations. *IEEE transactions on computers*, 100(6), 518–528.

Hufnagel, L., Brockmann, D., Geisel, T. (2004). Forecast and control of epidemics in a globalized world. *National academy of sciences*, 101(42), 15124-15129.

Huk, M., Shin, K., Kuboyama, T., Hashimoto, T. (2021). Random number generators in training of contextual neural networks. *Asian Conference on Intelligent Information and Database Systems*, 717-730.

Hunold, S. (2015). A survey on reproducibility in parallel computing. *arXiv preprint* arXiv :1511.04217, 15 p.

Hwu, W.W., Patt, N.Y. (1987). Checkpoint Repair for High-Performance Out-of-Order Execution Machines. *IEEE Transactions on Computers*, 100(12), 1496-1514.

Ide, Y., Yamasaki, N. (2020). A Learning-based Fetch Thread Gating Mechanism for A Simultaneous Multithreading Processor. *International Symposium on Computing and Networking (CANDAR),* 1-10.

Innocenti, E., Silvani, X., Muzy, A., Hill, D.R.C. (2009). A software framework for fine grain parallelization of cellular models with OpenMP: Application to fire spread. *Environmental Modelling & Software*, 24(7), 819-831.

Ioannidis, J.P. (2022). Correction: Why most published research findings are false. *PLoS Medicine*, 19(8), e1004085.

Ioannidis, J.P. (2015). How to make more published research true. *Revista Cubana de Información en Ciencias de la Salud (ACIMED)*, 26(2), 187–200.

Ioannidis, J.P. (2005). Why most published research findings are false. *PLoS medicine*, 2(8), e124.

Iqbal, S.A., Wallach, J.D., Khoury, M.J., Schully, S.D., Ioannidis, J.P. (2016). Reproducible research practices and transparency across the biomedical literature. *PLoS biology*, 14(1), e1002333.

Ivie, P., Thain, D. (2018). Reproducibility in scientific computing. *ACM Computing Surveys (CSUR),* 51(3), 1–36.

Jacobsen, D.M., Canon, R.S. (2015). Contain this, unleashing docker for hpc. *Proceedings of the Cray User Group*, 33–49.

Janz, N. (2016). Bringing the gold standard into the classroom: replication in university teaching. *International Studies Perspectives*, 17(4), 392–407.

Jézéquel, F., Lamotte, J.-L., Saïd, I. (2015). Estimation of numerical reproducibility on CPU and GPU. *Federated Conference on Computer Science and Information Systems (FedCSIS*), 675–680.

Jin, X., Zhou Y., Huang, B., Yu, Z., Zhan, X., Wang, H., Bao, Y. (2019). Qosmt: supporting precise performance control for simultaneous multithreading architecture. *International Conference on Supercomputing*, 206-216.

Johnson, A.L., Johnson, B.C. (1997). Literate programming using noweb. *Linux Journal*, 42, 64–69.

Kapoor, S., Narayanan, A. (2023). Leakage and the reproducibility crisis in machine-learning-based science. *Patterns*, 4(9).

Kazempour, V., Fedorova, A., Alagheband, P. (2008). Performance Implications of Cache Affinity on Multicore Processors. *European Conference on Parallel Processing*, 151-161.

Keller Tesser, R., Borin, E. (2023). Containers in HPC: a survey. *The Journal of Supercomputing*, 79(5), 5759–5827.

Kermack, W.O., McKendrick, A.G. (1932). Contributions to the mathematical theory of epidemics. II.—The problem of endemicity. *Proceedings of the Royal Society of London. Series A, containing papers of a mathematical and physical character*, 138(834), 55-83.

Kerr, N.L. (1998). HARKing: Hypothesizing after the results are known. *Personality and social psychology review*, 2(3), 196–217.

Khan, K.N., Hirki, M., Niemi, T., Nurminen, J.K., Ou, Z. (2018). RAPL in Action : Experiences in Using RAPL for Power measurements. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS),* 3(2), 1-26.

Kim, K., Kim, J., Yu, J., Seo, J., Lee, J., Choi, K. (2016). Dynamic energy-accuracy trade-off using stochastic computing in deep neural networks. *Annual Design Automation Conference*, 1-6.

Kingma, D.P., Welling, M. (2013). Auto-encoding variational bayes. *arXiv preprint* arXiv:1312.6114.
https://www.ee.bgu.ac.il/~rrtammy/DNN/StudentPresentations/2018/AUTOEN~2.PDF

Kitzes, J., Turek, D., Deniz, F. (2018). The practice of reproducible research: case studies and lessons from the data-intensive sciences. *Univ of California Press*. 368 p.

Kivity, A., Kamay, Y., Laor, D., Lublin, U., Liguori, A. (2007). kvm : the Linux virtual machine monitor. *Linux symposium,* 225–230.

Klôh, V.P., Silva, G.D., Ferro, M., Araújo, E., de Melo, C.B., de Andrade Lima, J.R.P., Martins, E.R. (2020). The virus and socioeconomic inequality : An agent-based model to simulate and assess the impact of interventions to reduce the spread of COVID-19 in Rio de Janeiro, Brazil. *Brazilian Journal of Health Review*, 3(2), 3647-3673.

Kluyver, T., Ragan-Kelley, B., Pérez, F., Granger, B.E., Bussonnier, M., Frederic, J., Kelley, K., Hamrick, J.B., Grout, J., Corlay, S. (2016). Jupyter Notebooks-a publishing format for reproducible computational workflows. *Elpub*, 87–90.

Knuth, D.E. (1999). The Art of Computer Programming. Numerical Algorithms (Vol. 2). *Addison Wesley/Pearson Education*. 688 p.

Knuth, D.E., Levy, S. (1994). The CWEB system of structured documentation: version 3.0 *Addison-Wesley Longman Publishing Co., Inc*.

Knuth, D.E. (1984). Literate programming. *The computer journal*, 27(2), 97–111.

Knuth, D.E. (1973). The art of computer programming (Vol. 3). *Addison-Wesley Reading*.

Kohanoff, J., Reinhard, P. G., Stella, L., Suraud, E. (2024). Guidebook to Real Time Electron Dynamics: Irradiation Dynamics from Molecules to Nanoclusters. *CRC Press*, 262 p.

Koivu, A., Kakko, J.-P., Mäntyniemi, S., Sairanen, M. (2022). Quality of randomness and node dropout regularization for fitting neural networks. *Expert Systems with Applications*, 207, 117938.

Koopman, J.S., Lynch, J.W. (1999). Individual causal models and population system models in epidemiology. *American journal of public health*, 89(8), 1170-1174.

Köster, J., Rahmann, S. (2012). Snakemake—a scalable bioinformatics workflow engine. *Bioinformatics*, 28(19), 2520–2522.

Kovacevic, J. (2007). How to encourage and publish reproducible research. *International Conference on Acoustics, Speech and Signal Processing-ICASSP*, 1273-1276.

Kurkowski, S., Camp, T., Colagrosso, M. (2005). MANET simulation studies: the incredibles. *ACM SIGMOBILE Mobile Computing and Communications Review*, 9(4), 50–61.

Kurtzer, G.M., Sochat, V., Bauer, M.W. (2017). Singularity: Scientific containers for mobility of compute. *PloS one*, 12(5), e0177459.

L'ecuyer, P. (1999). Good parameters and implementations for combined multiple recursive random number generators. *Operations Research*, 47(1), 159–164.

L'ecuyer, P., Simard, R. (2007). TestU01 : AC library for empirical testing of random number generators. *ACM Transactions on Mathematical Software (TOMS),* 33(4), 1-40.

L'Ecuyer, P., Munger, D., Oreshkin B., Simard, R. (2017). Random Numbers for Parallel Computers: Requirements and Methods, with Emphasis on Gpus. *Mathematics and Computers in Simulation*. 135, 3-17.

Lange, M. (2022). Toward accurate and fast summation. *ACM Transactions on Mathematical Software (TOMS)*, 48(3), 1–39.

Langezaal, E.R., Belleman, J., Veenboer, T., Noorthoek, J. (2023). [Re] Label-Free Explainability for Unsupervised Models. *ReScience C*, 9(2), 17 p.

Laperrière, V. (2006). SIMPEST: Un modèele multiagents pour simuler l'expression spatiale à grande eéchelle de la peste malgache. *RTP MODYS, Réseau thématique pluridisciplinaire Modélisation et Dynamiques Spatiales, Département SHS-CNRS*, 60-65.

Lazaro, D., El Bitar, Z., Breton, V., Hill, D.R.C, Buvat, I. (2005). Fully 3D Monte Carlo Reconstruction in SPECT: A Feasibility Study. *Physics in Medicine & Biology*, 50(16), 3739.

Le, E., Paz, D. (2017). Performance analysis of applications using singularity container on sdsc comet. *Practice and Experience in Advanced Research Computing on Sustainability, Success and Impact,* 1–4.

Lee, B., Lee, M., Mogk, J., Goldstein, R., Bibliowicz, J., Brudy, F., Tessier, A. (2021). Designing a multi-agent occupant simulation system to support facility planning and analysis for covid-19. *ACM Designing Interactive Systems Conference*, 15-30.

Lee, D., Wester, B., Veeraraghavan, K., Narayanasamy, S., Chen, P.M., Flinn, J. (2010). Respec: efficient online multiprocessor replayvia speculation and external determinism. *ACM Sigplan Notices*, 45(3), 77–90.

Legrand, A., Velho, P. (2023). [Re] Velho and Legrand (2009) - Accuracy Study and Improvement of Network Simulation in the SimGrid Framework. *ReScience C*, 6(1), 19 p.

Leng, T., Ali, R., Hsieh, J., Mashayekhi, V., Rooholamini, R. (2002). An Empirical Study of Hyper-threading in High-Performance Computing Clusters. *Linux HPC Revolution*. 12p.

Lenth, R.V. (2012). StatWeave users' manual. En ligne: https://homepage.stat.uiowa.edu/~rlenth/StatWeave/StatWeave-manual.pdf, accès le 30/07/2024. 19 p.

Lenth, R.V., Højsgaard, S. (2007). SASweave: Literate programming using SAS. *Journal of Statistical Software*, 19, 1–20.

Li, L. (2012). Testing several types of random number generator. *The Florida State University*, 98 p.

Li, R., Liu, L., Yang, G., Zhang, C., Wang, B. (2016). Bitwise identical compiling setup : Prospective for reproducibility and reliability of Earth system modeling. *Geoscientific Model Development*, 9(2), 731-748.

Lionel, S. (2013). Improving Numerical Reproducibility in C/C++/Fortran. *SuperComputing*. http://sc13.supercomputing.org/sites/default/files/WorkshopsArchive/pdfs/wp129s1.pdf

Liu, J., Pacitti, E., Valduriez, P., Mattoso, M. (2015). A survey of data-intensive scientific workflow management. *Journal of Grid Computing*, 13, 457–493.

Liu, Y., Liu, S., Wang, Y., Lombardi, F., Han, J. (2020). A survey of stochastic computing neural networks for machine learning applications. *Transactions on Neural Networks and Learning Systems*, 32(7), 2809-2824.

Liu, Y., Wang, Y., Lombardi, F., Han, J. (2018). An energy-efficient online-learning stochastic computational deep belief network. *Journal on Emerging and Selected Topics in Circuits and Systems*, 8(3), 454-465.

Grover, L.K. (1996). A fast quantum mechanical algorithm for database search. *ACM symposium on Theory of computing*, 212-219.

Ludwig, T. (2019). Bitwise reproducibility with exascale machines. [En ligne]. Disponible à l'adresse : https ://www.cs.fsu.edu/~nre/nre-2019/presentations/03-Ludwig.2019-06-20-ISC-Reproducibility.pdf [Consulté le 1 Janvier 2024].

Lutz, D.R., Hinds, C.N. (2017). High-Precision Anchored Accumulators for Reproducible Floating-Point Summation. *Computer Arithmetic (ARITH),* 98-105.

Maigne, L., Hill, D.R.C., Calvat P., Breton, V., Reuillon, R., Lazaro, D., Legre, Y., Donnarieix, D. (2004). Parallelization of Monte Carlo Simulations and Submission to a Grid Environment. *Parallel processing letters,* 14(2), 177-196.

Maneerat, S., Daudé, E. (2016). A spatial agent-based simulation model of the dengue vector Aedes aegypti to explore its population dynamics in urban areas. *Ecological Modelling*, 333, 66-78.

Manninen, T., Havela, R., Linne, M.-L. (2017). Reproducibility and comparability of computational models for astrocyte calcium excitability. *Frontiers in neuroinformatics*, 11, 18 p.

Marr, D.T., Binns, F., Hill, D.L, Hinton, G., Koufaty, D.A., Miller, J.A., Upton, M. (2002). Hyper-Threading Technology Architecture and Microarchitecture. *Intel Technology Journal*. 6(1), 12 p.

Marsaglia, G. (1996). DIEHARD: A Battery of Tests of Randomness. URL: https://web.archive.org/web/20160125103112/http://stat.fsu.edu/pub/diehard/

Marwick, B. (2015). How computers broke science—And what we can do to fix it. *The Conversation*. https://theconversation.com/how-computers-broke-science-and-what-we-can-do-to-fix-it-49938

Mashtizadeh, A.J., Garfinkel, T., Terei, D., Mazieres, D., Rosenblum, M. (2017). Towards practical default-on multi-core record/replay. *ACM SIGPLAN Notices*, 52(4), 693–708.

Mathiot, J.-F., Gerbaud, L., Breton, V. (2021). Highlighting the impact of social relationships on the propagation of respiratory viruses using percolation theory. *Scientific Reports*, 11(1), 24326.

Matsumoto, M., Nishimura, T. (1998). Mersenne twister : A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8(1), 3-30.

Matsumoto, M., Nishimura, T. (2000). Dynamic Creation of Pseudorandom Number Generators. *Monte Carlo and Quasi-Monte Carlo Methods Conference*, 56–69.

Matsumoto, M., Wada, I., Kuramoto, A., Ashihara, H. (2007). Common Defects in Initialization of Pseudorandom Number Generators. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 17(4), 15-es.

Matthews, J.N., Hu, W., Hapuarachchi, M., Deshane, T., Dimatos, D., Hamilton, G., McCabe, M., Owens, J. (2007). Quantifying the performance isolation properties of virtualization systems. *Workshop on Experimental computer science*, 6-es.

Mauerer, W., Scherzinger, S. (2022). 1-2-3 reproducibility for quantum software experiments. *International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 1247–1248.

Memeti, S., Li, L., Pllana, S., Kołodziej, J., Kessler, C. (2017). Benchmarking OpenCL, OpenACC, OpenMP, and CUDA: programming productivity, performance, and energy consumption. *Workshop on Adaptive Resource Management and Scheduling for Cloud Computing*, 1-6.

Meneses, E., Mendes, C.L., Kalé, L.V. (2010). Team-based message logging: Preliminary results. *International Conference on Cluster, Cloud and Grid Computing*, 697–702.

Merkel, D. (2014). Docker: lightweight linux containers for consistent development and deployment. *Linux j*, 239(2), 5 p.

Mesnard, O., Barba, L.A. (2017). Reproducible and replicable computational fluid dynamics: it's harder than you think. *Computing in Science & Engineering*. 19(4), 44-55.

Nielsen, M.A., Chuang, I.L. (2010). Quantum computation and quantum information. *Cambridge university press*. 665 p.

Miller, G. (2006). A scientist's nightmare: software problem leads to five retractions. *American Association for the Advancement of Science*, 314, 1856-1857.

Ministère de l'enseignement supérieur et de la recherche (2021). Second National Plan for Open Science. [En ligne]. Disponible à l'adresse : https ://www.ouvrirlascience.fr/second-national-plan-for-open-science/ [Consulté le 1 Janvier 2024].

Mittal, S., Vetter, J.S. (2015). A survey of techniques for modeling and improving reliability of computing systems. *IEEE Transactions on Parallel and Distributed Systems*, 27(4), 1226–1238.

Mniszewski, S.M., Del Valle, S.Y., Stroud, P.D., Riese, J.M., Sydoriak, S.J. (2008). EpiSimS simulation of a multi-component strategy for pandemic influenza. *Spring simulation multiconference*, 556-563.

Munafò, M.R., Nosek, B.A., Bishop, D.V., Button, K.S., Chambers, C.D., Percie du Sert, N., Simonsohn, U., Wagenmakers, E.-J., Ware, J.J., Ioannidis, J. (2017). A manifesto for reproducible science. *Nature human behaviour*, 1(1), 1–9.

Mytkowicz, T., Diwan, A., Hauswirth, M., Sweeney, P.F. (2009). Producing wrong data without doing anything obviously wrong!. *ACM Sigplan Notices*, 44(3), 265–276.

Nagarajan, P., Warnell, G., Stone, P. (2019). The Impact of Nondeterminism on Reproducibility in Deep Reinforcement Learning. *Workshop on Reproducible AI*. 10 p.

NASEM. (2019). Reproducibility and Replicability in Science, National Academies of Science Engineering and Medicine. https://nap.nationalacademies.org/catalog/25303/reproducibility-and-replicability-in-science

Nature Editorial. (2024). Trust but verify. *Nature materitals*, 23, 1 p.

Neal, R.M. (2012). Bayesian learning for neural networks. *Springer Science & Business Media*.

Neelakantan, A., Vilnis, L., Le, Q.V., Sutskever, I., Kaiser, L., Kurach, K., Martens, J. (2015). Adding gradient noise improves learning for very deep networks. *arXiv preprint* arXiv:1511.06807, 11 p.

Newman, M.E. (2003). The structure and function of complex networks. *SIAM review*, 45(2), 167-256.

Nguyen, L.M., Liu, J., Scheinberg, K., Takáč, M. (2017). SARAH: A novel method for machine learning problems using stochastic recursive gradient. *International conference on machine learning*, 2613-2621.

Nikolić, M., Jović, A., Jakić, J., Slavnić, V., Balaž, A. (2014). An analysis of FFTW and FFTE performance. *High-Performance Computing Infrastructure for South East Europe's Research Communities*, 163-170.

Normand, E. (1996). Single event upset at ground level. *IEEE transactions on Nuclear Science*, 43(6), 2742–2750.

Noureddine, A. (2022). Powerjoular and joularjx : Multi-platform software power monitoring tools. *International Conference on Intelligent Environments (IE),* 1-4.

Nüst, D., Sochat, V., Marwick, B., Eglen, S.J., Head, T., Hirst, T., Evans, B.D. (2020). Ten simple rules for writing Dockerfiles for reproducible data science. *Public Library of Science San Francisco*, 16, 11, p. e1008316.

Nuzzo, R. (2014). Scientific method: statistical errors. *Nature*, 506(7487), 151-152.

O'neill, M.E. (2014). PCG: A family of simple fast space-efficient statistically good algorithms for random number generation. Research report (was not published in ACM Transactions on Mathematical Software). https://www.pcg-random.org/paper.html

Ogasawara, E., Dias, J., Silva, V., Chirigati, F., De Oliveira, D., Porto, F., Valduriez, P., Mattoso, M. (2013). Chiron: a parallel engine for algebraic scientific workflows. *Concurrency and Computation: Practice and Experience*, 25(16), 2327–2341.

Oinn, T., Addis, M., Ferris, J., Marvin, D., Senger, M., Greenwood, M., Carver, T., Glover, K., Pocock, M.R., Wipat, A. (2004). Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17), 3045–3054.

Omae, Y., Toyotani, J., Hara, K., Gon, Y., Takahashi, H. (2021). Effectiveness of the COVID-19 contact-confirming application (COCOA) based on multi-agent simulation. *Journal of Advanced Computational Intelligence and Intelligent Informatics*, 25(6), 931-943.

Onose, E. (2023). How to Solve Reproducibility in Machine Learning, 13 November 2023. [En ligne]. Disponible à l'adresse: https://neptune.ai/blog/how-to-solve-reproducibility-in-ml [Consulté le 1 January 2024].

Open Science Collaboration (2015). Estimating the reproducibility of psychological science. *Science*, 349(6251), aac4716.

Osborne, S.H., Ahmed, S., Nandi, S., Anderson, J.H. (2020). Exploiting simultaneous multithreading in priority-driven hard real-time systems. *International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 1-10.

Osborne, S.H., Bakita, J.J., Anderson, J.H. (2019). Simultaneous multithreading applied to real time. *Dagstuhl artifacts series*. 5(1), 76 p.

Padala, P., Zhu, X., Wang, Z., Singhal, S., Shin, K.G. (2007). Performance evaluation of virtualization technologies for server consolidation. *HP Labs Tec. Report*, 137, 13 p.

Panneton, F., L'ecuyer, P., Matsumoto, M. (2006). Improved long-period generators based on linear recurrences modulo 2. *ACM Transactions on Mathematical Software (TOMS)*, 32(1), 1-16.

Park, S., Zhou, Y., Xiong, W., Yin, Z., Kaushik, R., Lee, K.H., Lu, S. (2009). Pres: probabilistic replay with execution sketching on multiprocessors. *ACM SIGOPS symposium on Operating systems principles*, 177–192.

Passerat-palmbach, J., Caux, J., Siregar, P., Mazel, C., Hill, D.R.C. (2011). Warp-Level Parallelism: Enabling Multiple Replications in Parallel on GPU. *European Simulation Multiconference*, 76-83.

Passerat-Palmbach, J., Mazel, C., Hill, D.R.C. (2011). Pseudo-Random Number Generation on GP-GPU. *ACM/IEEE/SCS Workshop on Principles of Advanced and Distributed Simulation (PADS 2011),* 146-153.

Patil, H., Pereira, C., Stallcup, M., Lueck, G., Cownie, J. (2010). Pinplay: a framework for deterministic replay and reproducible analysis of parallel programs. *ACM international symposium on Code generation and optimization*. 2–11.

Pereira, R., Couto, M., Ribeiro, F., Rua, R., Cunha, J., Fernandes, J.P., Saraiva, J. (2017). Energy efficiency across programming languages : How do energy, time, and memory relate? *ACM SIGPLAN international conference on software language engineering*, 256-267.

Perianayagam, S., Andrews, G.R., Hartman, J.H. (2010). Rex: a toolset for reproducing software experiments. *International Conference on Bioinformatics and Biomedicine (BIBM)*, 613–617.

Pernet, C.R., Wilcox, R., Rousselet, G.A. (2013). Robust correlation analyses: false positive and power validation using a new open source matlab toolbox. *Frontiers in psychology*, 3(606), 1–18.

Piller, C., Servick, K. (2020). Two elite medical journals retract coronavirus papers over data integrity questions. *Science*. [En ligne]. Disponible à l'adresse : https ://www.science.org/content/article/two-elite-medical-journals-retract-coronavirus-papers-over-data-integrity-questions.

Plesser, H.E. (2018). Reproducibility vs. replicability: a brief history of a confused terminology. *Frontiers in neuroinformatics*. 11, 4 p.

PLOS one Materials and Software Sharing. [En ligne]. Disponible à l'adresse : https ://journals.plos.org/plosone/s/materials-and-software-sharing [Consulté le 1 Janvier 2024].

Polyhedron benchmarks. (2024). Consulté 19 avril 2024, à l'adresse https://fortran.uk/fortran-compiler-comparisons/polyhedron-benchmarks-linux64-on-intel/

Popper, K. (2005). The logic of scientific discovery. *Routledge* (re-edition).

Potdar, A.M., Narayan, D.G., Kengond, S., Mulla, M.M. (2020). Performance evaluation of docker container and virtual machine. *Procedia Computer Science*, 171, 1419–1428.

Pouzat, C. (2021). Un panorama de la recherche reproductible. *Bulletin de la ROADEF*, 7 p.

Pouzat, C. (2022). Pourquoi devrions-nous arrêter d'embêter les gens avec la « recherche reproductible » et autres « bonnes pratiques » ?. *Statistique et Société*, 10(1), 53–57.

Pouzat, C., Legrand, A., Hinsen, K. (2018). Recherche Reproductible : Principes Methodologiques pour une Science Transparente. [En ligne]. Disponible à l'adresse : https ://www.fun-mooc.fr/fr/cours/recherche-reproductible-principes-methodologiques-pour-une-science-transparente/ [Consulté le 2 Janvier 2024].

Pradal, C., Fournier, C., Valduriez, P., Cohen-Boulakia, S. (2015). OpenAlea: scientific workflows combining data analysis and simulation. *International Conference on Scientific and Statistical Database Management*, 1–6.

Priedhorsky, R., Randles, T. (2017). Charliecloud: Unprivileged containers for user-defined software stacks in hpc. *International conference for high performance computing, networking, storage and analysis*. 1–10.

Rad, B.B., Bhatti, H.J., Ahmadi, M. (2017). An introduction to docker and analysis of its performance. *International Journal of Computer Science and Network Security (IJCSNS),* 17(3), 228.

Ragan-Kelley, B., Willing, C., Akici, F., Lippa, D., Niederhut, D., Pacer, M. (2018). Binder 2.0-Reproducible, interactive, sharable environments for science at scale. *Python in science conference*, 113–120.

Randall, D., Welser, C. (2018). The Irreproducibility Crisis of Modern Science: Causes, Consequences, and the Road to Reform. *ERIC*.

Madduri, R., Chard, K., d'Arcy, M., Jung, S.C., Rodriguez, A., Sulakhe, D., Deutsch, E., Funk, C., Heavner, B., Richards, M., Shannon, P., Glusman, G., Price, N., Kesselman, C., Foster, I. (2019). Reproducible big data science: A case study in continuous FAIRness. *PloS one*, 14(4), e0213013.

Ren, S., Li, C., Tan, L., Xiao, Z. (2015). Samsara : Efficient deterministic replay with hardware virtualization extensions. *Asia-Pacific workshop on systems*. 1–7.

Reuillon, R., Hill, D.R.C., El Bitar, Z., Breton, V. (2008) Rigorous Distribution of Stochastic Simulations Using the DistMe Toolkit, *Transactions on Nuclear Science*, 55(1), 595-603.

Reuillon, R. (2008). Testing 65536 Parallel Pseudo-Random Number Streams. *EGEE Grid User Forum, poster session*. 1 p.

Revol, N., Théveny, P. (2014). Numerical reproducibility and parallel computations : Issues for interval algorithms. *IEEE Transactions on Computers*, 63(8), 1915–1924.

Rijmen, V., Daemen, J. (2001). Advanced encryption standard. *Federal information processing standards publications, national institute of standards and technology*, 19, 7 p.

Rosenquist, R.T., Story, S. (2013). Using the intel math kernel library (intel MKL) and intel compilers to obtain run-to-run numerical reproducible results. *ACM/IEEE Supercomputing Conference*. 6 p.
https://sc13.supercomputing.org/sites/default/files/WorkshopsArchive/pdfs/wp143s1.pdf

Rosenthal, E., León, E.A., Moody, A.T. (2013). Mitigating system noise with simultaneous multi-threading. *SC13, poster session*. 2 p.

Rougier, N.P., Hinsen, K., Alexandre, F., Arildsen, T., Barba, L.A., Benureau, F.C., Brown, C.T., De Buyl, P., Caglayan, O., Davison, A.P. (2017). Sustainable computational science : the ReScience initiative. *PeerJ Computer Science*, 3, e142.

Royal Society (1660). History of the Royal Society. [En ligne]. Disponible à l'adresse : https ://royalsociety.org/about-us/history/ [Consulté le 1 Janvier 2024].

Ruiz, C., Jeanvoine, E., Nussbaum, L. (2015). Performance evaluation of containers for HPC. *Euro-Par: Parallel Processing Workshops*, 813–824.

Ruiz, C., Richard, O., Emeras, J. (2014). Reproducible software appliances for experimentation. *Testbeds and Research Infrastructure: Development of Networks and Communities*, 33–42.

Rukhin, A., Soto, J., Nechvatal, J., Smid, M., Barker, E., Leigh, S., Levenson, M., Vangel, M., Banks, D., Heckert, A. (2001). A statistical test suite for random and pseudorandom number generators for cryptographic applications. *US Department of Commerce, Technology Administration, National Institute of Strandards of Technology*.

Rump, S.M., Ogita, T., Oishi, S. (2010). Fast high precision summation. *Nonlinear Theory and Its Applications*, 1(1), 2–24.

Rutherford, E., Royds, T. (1908). LXVIII. The action of the radium emanation upon water. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 16(95), 812–818.

Saini, S., Jin, H., Hood, R., Barker, D., Mehrotra, P., Biswas, R. (2011). The Impact of Hyper-threading on Processor Resource Utilization in Production Applications. *Conference on High Performance Computing*, 1-10.

Saito, M., Matsumoto, M. (2006). SIMD-oriented fast Mersenne Twister : A 128-bit pseudorandom number generator. *Monte Carlo and Quasi-Monte Carlo Methods*, 607-622.

Salje, H., Tran Kiem, C., Lefrancq, N., Courtejoie, N., Bosetti, P., Paireau, J., Andronico, A., Hozé, N., Richet, J., Dubost, C.-L. (2020). Estimating the burden of SARS-CoV-2 in France. *Science*, 369(6500), 208-211.

Salmon, J.K., Moraes, M.A., Dror, R.O., Shaw, D.E. (2011). Parallel random numbers : as easy as 1, 2, 3. *International conference for high performance computing, networking, storage and analysis*, 1–12.

Schroeder, B., Pinheiro, E., Weber, W.-D. (2009). DRAM errors in the wild: a large-scale field study. *ACM SIGMETRICS Performance Evaluation Review*, 37(1), 193–204.

Schulte, E., Davison, D., Dye, T., Dominik, C. (2012). A multi-language computing environment for literate programming and reproducible research. *Journal of Statistical Software*, 46, 1–24.

Schwab, M., Karrenbach, N., Claerbout, J. (2000). Making scientific computations reproducible. *Computing in Science & Engineering*, 2(6), 61-67.

Senthil Kumaran, S. (2017). Practical LXC and LXD: linux containers for virtualization and orchestratio. *New York: Apress.*

Shaydulin, R., Marwaha, K., Wurtz, J., Lotshaw, P.C. (2021). QAOAKit: A toolkit for reproducible study, application, and verification of the QAOA. *International Workshop on Quantum Computing Software (QCS)*, 64–71.

Shorten, C., Khoshgoftaar, T.M. (2019). A survey on image data augmentation for deep learning. *Journal of big data*, 6(1), 1-48.

Krishnamurthi, S., Vtek, J. (2015). The real software crisis: Repeatability as a core value. *Communications of the ACM*, 58(3), 34-36.

Sommerville, I. (2001). Software documentation. *Software engineering*, 2, 143–154.

Sorrell, S. (2009). Jevons' Paradox revisited : The evidence for backfire from improved energy efficiency. *Energy policy*, 37(4), 1456-1469.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., Salakhutdinov, R. (2014). Dropout : A simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1), 1929-1958.

Stallman, R.M. (1981). EMACS the extensible, customizable self-documenting display editor. *ACM SIGPLAN SIGOA symposium on Text manipulation*. 147–156.

Stanisic, L., Legrand, A., Danjean, V. (2015). An effective git and org-mode based workflow for reproducible research. *ACM SIGOPS Operating Systems Review*, 49(1), 61–70.

Stodden, V. (2011). Trust your science? Open your data and code. *Amstat news*, 21-22.

Stodden, V., Borwein, J., Bailey, D.H. (2013). Setting the default to reproducible. computational science research. *SIAM News*, 46(5), 4–6.

Stodden, V., Krafczyk, M.S., Bhaskar, A. (2018). Enabling the verification of computational results : An empirical evaluation of computational reproducibility. *International Workshop on Practical Reproducible Evaluation of Computer Systems*. 1–5.

Stodden, V., Leisch, F., Peng, R.D. (2014). Implementing reproducible research. *CRC Press*.

Strong, R.W., Alvarez, G. (2019). Using simulation and resampling to improve the statistical power and reproducibility of psychological research. *OSF preprints*, 27 p.

Sutton, R.S., Barto, A.G. (2018). Reinforcement learning : An introduction. *MIT press*.

Szalay A.S., Bell G., Huang H.H., Terzis A., White A. (2010) Low-power amdahl-balanced blades for data intensive computing. *ACM SIGOPS Operating Systems Review*. 44(1), 71-75.

Tadić, B., Melnik, R. (2020). Modeling latent infection transmissions through biosocial stochastic dynamics. *PloS one*, 15(10), e0241163.

Taufer, M., Padron, O., Saponaro, P., Patel, S. (2010). Improving numerical reproducibility and stability in large-scale numerical simulations on GPUs. *International Symposium on Parallel & Distributed Processing (IPDPS)*, 1–9.

Taylor, I., Shields, M., Wang, I., Harrison, A. (2007). The triana workflow environment: Architecture and applications. *Workflows for e-science: Scientific workflows for grids*, 320–339.

Ten Hagen, K.G. (2016). Novel or reproducible: That is the question. *Glycobiology*, 26(5), 429–429.

Tikir, M.M., Carrington, L., Strohmaier, E., Snavely, A. (2007). A Genetic Algorithms Approach to Modeling the Performance of Memory-Bound Computations. *ACM/IEEE Conference on Supercomputing,* 1-12.

Topalidou, M., Leblois, A., Boraud, T., Rougier, N.P. (2015). A long journey into reproducible computational neuroscience. *Frontiers in computational neuroscience*, 9, 30, 2 p.

Torrez, A., Randles, T., Priedhorsky, R. (2019). HPC container runtimes have minimal or no performance impact. *International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*, 37–42.

Traore, M., Hill, D.R.C. (2001). The use of random number generation for stochastic distributed simulation: application to ecological modeling. *European Simulation Symposium*, 555-559.

Tsamardinos, I., Greasidou, E., Borboudakis, G. (2018). Bootstrapping the out-of-sample predictions for efficient and accurate cross-validation. *Machine learning*, 107, 1895-1922.

Tuck N., Tullsen M.D. (2003). Initial Observations of the Simultaneous Multithreading Pentium 4 Processor. *International Conference on Parallel Architectures and Compilation Techniques*, 26-34.

Tuomisto, J.T., Yrjölä, J., Kolehmainen, M., Bonsdorff, J., Pekkanen, J., Tikkanen, T. (2020). An agent-based epidemic model REINA for COVID-19 to identify destructive policies. *MedRxiv*, 2020.04.09.20047498.

Utterback, R., Agrawal, K., Lee, I.-T.A., Kulkarni, M. (2017). Processor-oblivious record and replay. *ACM SIGPLAN Notices*, 52(8), 145–161.

Van Dyk, D.A., Meng, X.-L. (2001). The art of data augmentation. *Journal of Computational and Graphical Statistics*, 10(1), 1-50.

Vandewalle, P., Kovacevic, J., Vetterli, M. (2009). Reproducible research in signal processing. *IEEE Signal Processing Magazine*, 26(3), 37–47.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., Polosukhin, I. (2017). Attention is all you need. *Advances in neural information processing systems*, 11 p.

Vyklyuk, Y., Manylich, M., Škoda, M., Radovanović, M.M., Petrović, M.D. (2021). Modeling and analysis of different scenarios for the spread of COVID-19 by using the modified multi-agent systems–Evidence from the selected countries. *Results in Physics*, 20, 103662.

Wang, C., Dryden, N., Cappello, F., Snir, M. (2018). Neural network based silent error detector. *International Conference on Cluster Computing (CLUSTER)*, 168–178.

Wang, L., Xu, T., Stoecker, T., Stoecker, H., Jiang, Y., Zhou, K. (2021). Machine learning spatio-temporal epidemiological model to evaluate Germany-county-level COVID-19 risk. *Machine Learning: Science and Technology*, 2(3), 035031.

Wilkinson, M.D., Dumontier, M., Aalbersberg, I.J., Appleton, G., Axton, M., Baak, A., Blomberg, N., Boiten, J.-W., da Silva Santos, L.B., Bourne, P.E. (2016). The FAIR Guiding Principles for scientific data management and stewardship. *Scientific data*, 3(1), 1–9.

Wilson, G., Aruliah, D.A., Brown, C.T., Chue Hong, N.P., Davis, M., Guy, R.T., Haddock, S.H., Huff, K.D., Mitchell, I.M., Plumbley, M.D. (2014). Best practices for scientific computing. *PLoS biology*, 12(1), e1001745.

Wolfram, S. (2018). Complex systems theory 1. *Emerging syntheses in science, CRC press,* 183-190.

Wolfram, S. (1986). Random Sequence Generation by Cellular Automata. *Advances in Applied Mathematics,* 7(2), 123–169.

Wu, X., Mueller, F. (2013). Elastic and scalable tracing and accurate replay of non-deterministic events. *International conference on supercomputing*. 59–68.

Wu, X., Vijayakumar, K., Mueller, F., Ma, X., Roth, P.C. (2011). Probabilistic communication and i/o tracing with deterministic replay at scale. *International Conference on Parallel Processing*, 196–205.

Wulf, W.A., McKee, S. (1995). Hitting the memory wall: Implications of the obvious. *ACM SIGARCH Computer Architecture News,* 23(1), 20-24.

Xavier, M.G., Neves, M.V., Rossi, F.D., Ferreto, T.C., Lange, T., De Rose, C.A. (2013). Performance evaluation of container-based virtualization for high performance computing environments. *Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, 233–240.

Junyuan, X., Girshick, R., Farhadi, A. (2016). Unsupervised deep embedding for clustering analysis. *International conference on machine learning*. 478-487.

Xue, R., Liu, X., Wu, M., Guo, Z., Chen, W., Zheng, W., Zhang, Z., Voelker, G. (2009). MPIWiz : Subgroup reproducible replay of MPI applications. *ACM SIGPLAN symposium on Principles and practice of parallel programming*, 251–260.

Yong, C., Lee, G.-W., Huh, E.N. (2018). Proposal of container-based HPC structures and performance analysis. *Journal of Information Processing Systems*, 14(6), 1398–1404.

Younge, A.J., Pedretti, K., Grant, R.E., Brightwell, R. (2017). A tale of two systems: Using containers to deploy HPC applications on supercomputers and clouds. *International Conference on Cloud Computing Technology and Science (CloudCom)*, 74–81.

Young-S, L. E., Muruganandam, P., Adhikari, S. K., Lončar, V., Vudragović, D., Balaž, A. (2017). OpenMP GNU and Intel Fortran programs for solving the time-dependent Gross–Pitaevskii equation. *Computer Physics Communications*, 220, 503-506.

Zhang, Y., Laurenzano, M.A., Mars, J., Tang, L. (2014). Smite: Precise QoS prediction on real-system SMT processors to improve utilization in warehouse scale computers. *International Symposium on Microarchitecture*, 406-418.

Zhao, Y., Hategan, M., Clifford, B., Foster, I., Von Laszewski, G., Nefedova, V., Raicu, I., Stef-Praun, T., Wilde, M. (2007). Swift: Fast, reliable, loosely coupled parallel computation. *Congress on Services (Services),* 199–206.

Zhou, N., Zhou, H., Hoppe, D. (2022). Containerization for High Performance Computing Systems: Survey and Prospects. *Transactions on Software Engineering*, 49(4), 2722–2740.

Zhuang, D., Zhang, X., Song, S., Hooker, S. (2022). Randomness in neural network training: Characterizing the impact of tooling. *Machine Learning and Systems*, 4, 316-336.

Zolfagharifard, E., Boland, H. (2020). Coding that led to lockdown "totally unreliable" and "a buggy mess", say experts. [En ligne]. Disponible à l'adresse : https ://www.telegraph.co.uk/technology/2020/05/16/coding-led-lockdown-totally-unreliable-buggy-mess-say-experts/ [Consulté le 1 Janvier 2024].

Zwirn, H. P. (2000). Limites de la connaissance (Les). *Odile Jacob*. 384 p.

Zwirn, H. P. (2006). Systèmes complexes (Les) : Mathématiques et biologie. *Odile Jacob*. 224 p.