Nombres aléatoires parallèles, simulation, et recherche reproductible

Article de vulgarisation

Martin Benito-Rodriguez
Université Clermont Auvergne
Clermont-Ferrand, France

***Abstract*—Cette article est une vulgarisation de l'article de David Hill [1]. La génération de nombres pseudo-aléatoires dans un contexte parallèle doit être de bonne qualité dans le domaine des simulations stochastiques parallèles. Il faut s'assurer à ce que les résultats soit reproductibles pour faire de la recherche, mais pourtant on peut avoir des résultats très différents si l'on ne connait pas les bonnes méthodes. Après une simulation stochastique parallèle, nous pouvons vérifier les résultats avec une contrepartie séquentielle.**

Mots-clés; générateurs; nombre pseudo-aléatoire; parallélisme; simulation; stochastique; reproductibilité; flux indépendants ; Calcul Haute Performance

I. INTRODUCTION

Dans les sciences expérimentales, la reproductibilité est au coeur de la recherche, il en est de même pour les simulations en informatique.

Pour tester, vérifier et trouver des erreurs et s'assurer de la validité des expériences et des ordinateurs, nous devons répliquer les résultats, c'est à dire, être capable de trouver les mêmes résultats avec les mêmes algorithmes et les mêmes données d'entrées.

Pour faire cela, il est nécessaire de répliquer les expériences, partager les informations tel le code, les données, utiliser la relecture par les pairs. Mais cette tache est rendue difficile car peu de scientifiques partagent leurs données et leurs codes, la réplication peut aussi etre compliqué d'un point de vue technique avec par exemple, les calculs utilisant des nombres flottants, les différences d'architectures, les différents compilateurs ou les différents langages.

Cependant, on voit de plus en plus en plus que même si tous les paramètres listés ci dessus sont respectés, la reproductibilité n'est pas assurée. Et le besoin de reproductibilité est de plus en plus important ces dernières années. Dans le domaines de la simulation stochastiques parallèles, c'est surtout le manque d'education académique qui est responsable du manque de reproductibilité. Il nous faut comprendre comment lancer des simulations stochastiques parallèles reproductibles de bonne qualitées sans que les flux parallèles s'influencent.

Depuis 1998 et l'article "Don't Trust Parallel Monte Carlo !" de Peter Hellekalek [1], de nombreuses avancées ont été faites. Ces avancées sont notamment dues aux exigences de la sûreté nucléaire qui nécessite des générateurs de nombres aléatoires avec une bonne distribution. D'autres domaines tel la médecine nucléaire ont besoins pour leurs simulations de plus de 10^20 nombres pseudo-aléatoires pour faire des calculs sur plusieurs milliers de processeurs. Les nombres aléatoires doivent êtres générés par différents flux indépendants, ce qui nécessite d'utiliser des techniques de parallélisme.

II. FLUX STOCHASTIQUES INDÉPENDANTS POUR LES SIMULATIONS PARALLÈLES

Nous allons voir ici deux méthodes pour avoir des flux de nombres aléatoires avec le parallélisme.

A. Partitionner un seul flux de nombre aléatoires:

La méthode la plus basique pour partitionner un générateur de nombres pseudo-aléatoires est d'utiliser un serveur centrale qui gère l'unique générateur. Les méthodes basées sur les automates booléens cellulaires peuvent générer des nombres pseudo-aléatoires dans un contexte parallèle, mais elles ne sont pas utilisées dans le domaine du calcul haute performace.

La méthode du *sequence splitting*/*blocking*/*regular spacing* consiste à diviser la séquence en N blocs contigus.

La méthode de la séquence indéxé ou *random spacing* initialise le générateur avec différents paramètres aléatoires.

La dernière méthode est la méthode *leap frog* ou des nombres aléatoires sont distribués entre les différents processus comme lorsque l'on distribue des cartes à des joueurs.

B. Produire plusieurs flux indépendants

Pour avoir nos flux indépendants , on va avoir besoin de générateurs parallèles indépendants qui appartiennent à la même famille, c'est ce qu'on appelle la *paramétrisation*.

Pour les générateurs de la famille Mersenne Twisters (MT) [2], l'outil pour créer le plus de générateurs indépendants est *Dynamic Creator* (DC) [3].

TinyMT et MTGP (utilisé pour GP-GPU) partage l'approche de DC.

On peut aussi utilliser le Multiplicative Lagged Fibonacci Generators (MLFG) [4].

Chacun des avantages et désavantages de ses techniques sont expliqués dans l'article [5].

III. Tests et compatibilité

Une chose importante à savoir est qu'on ne peut pas prouver mathématiquement l'indépendance de flux lancées en parallèles. Mais il y a quand même des pistes de recherche pour éviter les corrélations entre différents flux parallèles [6] [7].

Les générateurs pour le calcul parallèle doivent être testés avec une collection de tests nommée BigCrush Test U01 [8]. Les générateurs qui passent les tests avec succès sont dit « crush-resistant » [9], mais ils ne sont pas pour autant parfait.

On peut aussi utiliser les conseils de Srinivasan et Mascagni [10], l'article repose sur des propositions de Mascagni pour la génération de nombre pseudo-aléatoires parallèle [11].

Il existe encore d'autres techniques listés dans cet article [12].

Les algorithmes suivant pour la distribution de nombres aléatoires dans un environnement parallèle sont intéressant pour les architectures multicoeurs.

-MRG32k3a

-MLFG6331_64

-MTGP/TinyMT

-Phylox/Threefry

Pour les simulations stochastiques, il vaut mieux utiliser plusieurs types de générateurs pour déterminer la variabilité stochastique. Ceux présentés ici appartiennent à des familles différentes et ont des structures différentes.

IV. Création de simulations stochastiques parallèle reproductibles

Pour la simulation stochastique parallèle, il est difficile d'avoir des résultats de qualités dans des contextes différents. Pour reformuler: il est difficile d'obtenir les mêmes résultats en changeant des paramètres comme le nombre de processeurs. Pour conserver une bonne qualité, on peut suivre plusieurs pistes: une approche orientée objet [13][14], l'utilisation de générateurs modernes et statistiquement robustes, l'utilisation de techniques de parallélisation adaptées au générateur, une méthode de conception parallèle à partir de la conception du programme séquentiel. Les détails pour comprendre ce dernier point sont les suivants:

-créer un programme séquentiel avec en entrée des données de référence, mais avec des processus stochastiques indépendants qui ont leur propre flux de nombre pseudo aléatoires.

-les flux stochastiques indépendants doivent êtres créer avec une des méthodes montrée plus haut: plusieurs générateurs parallèles indépendants ou un générateur qui créer plusieurs flux indépendants. Il faut penser à sauvegarder les paramètres des flux pour retrouver la meme séquence (pour s'assurer de la reproductibilité).

-distribution du calcul entre tous les processus

-vérification que l'exécution parallèle avec un nombre différent de processus donne le même résultat que l'exécution séquentielle si l'initialisation et les paramètres sont identiques. Vérifier la procédure de réduction avec sa contrepartie séquentielle. Avant le déploiement sur une grande échelle, la répétabilité doit être vérifiée avec une exécution séquentielle significative.

Cette technique permet d'avoir des simulations stochastiques parallèles reproductibles et une comparaison avec une implémentation séquentielle. Même si ce système ne résout pas tous le problème de la reproductibilité, cela reste une avancé interessante.

Cependant, les architectures GP-GPU ne permettent pas forcément une reproductibilité numérique. D'après Intel la reproductibilité bit-to-bit est impossible avec un Intel Xeon Phi [15]. Il faudrait aussi être d'autant plus prudent dans un contexte hybride, avec par exemple différents compilateurs ou processeurs.

Pour avoir des résultats parfaitement identiques d'une expériences à l'autre avec les mêmes entrées et paramètres dans un environnement parallèle, il faudrait pouvoir contrôler l'ordre de toute les opérations. Mais si nous voulons avoir de la reproductibilité de bon niveau aujourd'hui, nous devons adopter des méthodes telles que celles cités dans l'article, continuer à faire de la recherche et communiquer entre informaticiens pour avoir des bonnes pratiques.

References

[1] P. Hellekalek, “Don’t Trust Parallel Monte Carlo!” Proc. Parallel and Distributed Simulation Conf., 1998, pp. 82–89.

[2] M. Matsumoto and T. Nishimura, “Mersenne Twister: A 623-Dimensionnally Equidistributed Uniform Pseudorandom Number Generator,” ACM Trans. Modeling and Computer Simulations, vol. 8, no. 1, 1998, pp. 3–30

[3] M. Matsumoto and T. Nishimura, “Dynamic Creation of Pseudorandom Number Generators,” Monte Carlo and Quasi-Monte Carlo Methods, H. Niederreiter and J. Spanier, eds., Springer, 1998, pp. 56–69

[4] M. Mascagni and A. Srinivasan, “Parameterizing Parallel Multiplicative Lagged-Fibonacci Generators,” Parallel Computing, vol. 30, 2004, pp. 899–916

[5] D.R.C. Hill et al., “Distribution of Random Streams for Simulation Practitioners,” Concurrency and Computation: Practice and Experience, vol. 25, no. 10, 2013, pp. 1427–1442.

[6] A.D. De Matteis and S. Pagnutti, “Parallelization of Random Number Generators and Long-Range Correlations,” Numerische Mathematik, vol. 53, no. 5, 1988, pp. 595–608.

[7] A.D. De Matteis and S. Pagnutti, “Controlling Correlations in Parallel Monte Carlo,” Parallel Computing, vol. 21, no. 1, 1995, pp. 73–84.

[8] P. L’Ecuyer and R. Simard, “TestU01: A C Library for Empirical Testing of Random Number Generators,” ACM Trans. Mathematical Software, vol. 33, no. 4, 2007; www.iro.umontreal.ca/~lecuyer/myftp/.../testu01.pdf

[9] J.K. Salmon et al., “Parallel Random Numbers: As Easy as 1, 2, 3,” Proc. Int’l Conf. High Performance Computing, Networking, Storage and Analysis, 2011, pp. 11–12.

[10] A. Srinivasan, M. Mascagni, and D. Ceperley, “Testing Parallel Random Number Generators,” Parallel Computing, vol. 29, no. 1, 2003, pp. 69–94.

[11] M. Mascagni, D. Ceperley, and A. Srinivasan, “SPRNG: A Scalable Library for Pseudorandom Number Generation,” ACM Trans. Mathematical Software, vol. 26, no. 3, 2000, 436–461.

[12] P.D. Coddington and S.-H. Ko, “Random Number Generator for Parallel Computers,” Proc. 12th Int’l Supercomputing Conf., 1998, pp. 282–288.

[13] D.R.C. Hill et al., “Distribution of Random Streams for Simulation Practitioners,” Concurrency and Computation: Practice and Experience, vol. 25, no. 10, 2013, pp. 1427–1442

[14] O.J. Dahl and K. Nygaard, “Simula: An ALGOLbased Simulation Language,” Comm. ACM, vol. 9, 1966, pp. 671–678.

[15] M. Taufer et al., “Improving Numerical Reproducibility and Stability in Large-Scale Numerical Simulations on GPUs,” IEEE Int’l Symp. Parallel and Distributed Processing, 2010, pp. 1–9.