

Générateurs de nombres aléatoires en parallèle

Jérôme Lelong

Ensimag

Année 2013-2014

- 1 Pourquoi utiliser des nombres aléatoires ?
- 2 Les générateurs de nombres aléatoires
 - Généralités et premier exemple
 - Des générateurs plus évolués
- 3 Et en parallèle ?
 - Les contraintes
 - Quelles solutions ?
- 4 Comment tester un générateur

Utilisation de tirages aléatoires en calcul parallèle I

- Chaque processeur calcule un résultat à part entière. Pas de “réduction” des différents résultats à la fin.
Exemple : Valorisation d'un portefeuille de produits financiers Les calculs sont indépendants : une valorisation se déroule sur un seul processeur, pas de communication entre les processeurs.
 ~> Toutes les valorisations peuvent à priori être faites avec les mêmes nombres aléatoires
- Un seul calcul de type Monte-Carlo est réalisé sur plusieurs processeurs avec “réduction” des différents résultats intermédiaires.
Exemple : Valorisation d'une seule option.
 ~> Les tirages utilisés par les différents processeurs doivent être “statistiquement indépendants”.

Utilisation de tirages aléatoires en calcul parallèle II

- Chaque processeur effectue un calcul de type Monte-Carlo qui intervient dans un calcul plus global.
Exemple : Résolution d'une EDP par une méthode probabiliste.
 ~> Les tirages utilisés par les différents processeurs doivent être “statistiquement indépendants”. Au sein d'un même processeur les tirages doivent être identiquement distribués.

- ▶ Les nombres aléatoires utilisés par chaque processeur doivent être indépendants entre eux et avoir de bonnes propriétés statistiques.
- ▶ Les suites utilisées par les différents processeurs doivent être indépendantes entre elles.
- ▶ La génération des nombres aléatoires ne doit pas induire de communication supplémentaire entre les processeurs.

- 1 Pourquoi utiliser des nombres aléatoires ?
- 2 Les générateurs de nombres aléatoires
 - Généralités et premier exemple
 - Des générateurs plus évolués
- 3 Et en parallèle ?
 - Les contraintes
 - Quelles solutions ?
- 4 Comment tester un générateur

Généralités sur les générateurs de nombres aléatoires

Ce sont des suites déterministes qui ont de bonnes propriétés statistiques.
i.e.

- ▶ propriété d'indépendance
- ▶ propriété d'uniformité
- ▶ grande période
- ▶ faible empreinte mémoire (moins un critère aujourd'hui)
- ▶ rapide

Un générateur de nombres aléatoires est une **machine à états** à nombre d'états finis. On peut formellement décrire un générateur aléatoire par

- ▶ un état initial s_0 appelé **graine** (**seed** en anglais),
- ▶ une fonction de transition sur les états possible telle que $s_{k+1} = S(s_k)$,
- ▶ une fonction V définie sur l'espace des états et telle que $V(s_k)$ est le "nombre aléatoire" produit à l'étape k .

la congruence linéaire (LCG)

- ▶ $S(x) = (Ax + B) \bmod M$
- ▶ Les états sont les entiers de 0 à $M - 1$.
- ▶ Les fonctions S et V sont les mêmes.

Quelques valeurs possibles pour A, B, M .

	A	B	M
Numerical Recipes	1664525	10113904223	2^{32}
glibc	1103515245	12345	2^{32}
Microsoft Visual	214013	2531011	2^{32}
Apple	16807	0	$2^{31} - 1$

- ▶ Si on relance un code 2 fois avec la même graine, alors les mêmes nombres aléatoires seront utilisés.
- ▶ On ne fixe la graine qu'une fois dans un code séquentiel.
- ▶ En parallèle

```
#pragma omp parallel
PnlRng *rng = pnl_rng_create(PNL_RNG_MERSENNE);
pnl_rng_sseed (rng, 4172);
#pragma omp for
for (i=0; i<N; i++)
    calcul utilisant rng
```

Dans cet exemple, les mêmes tirages sont utilisés par toutes les threads. Le minimum est d'utiliser des graines différents dans chaque thread.

- ▶ Pour un triplet (A, B, M) fixé, s'il existe n, p tels que $s_p = s_n$, alors $\forall k \geq 0, s_{p+k} = s_{n+k}$.
- ▶ Il existe $n \leq M$, tel que $s_n = s_0$. La période du générateur est le plus petit tel n .
Voici une condition pour qu'un générateur soit de période maximale.

Proposition (Art of Computer Programming, D. Knuth)

Un générateur LCG est de période maximale si

- ▶ M et B sont premiers entre eux
- ▶ tout nombre premier divisant M divise aussi $A - 1$.
- ▶ si 4 divise $A - 1$, alors 4 divise M .

Cas particulier : $B = 0$ et M est un nombre premier.

Améliorations possibles :

- ▶ Travailler avec des entiers 64bits. \rightsquigarrow Ce n'est pas encore vraiment portable. Même sur les architectures 64 bits, les entiers sont codés sur 32 bits, seules les adresses utilisent 64 bits.
- ▶ Augmenter la taille de l'espace d'états en augmentant le degré de récursivité. On parle alors parfois de générateurs multiplicatifs récursifs (MRG) d'ordre k . Dans ce cas, la périodicité de la suite n'apparaît qu'il existe n, p , tels que

$$s_{n-k} = s_{p-k}, \dots, s_n = s_p.$$

Ces générateurs sont une extension directe des générateurs à congruence linéaire et s'écrivent sous la forme

$$s_{n+1} = a_1 s_n + a_2 s_{n-1} + \dots + a_k s_{n-k} \bmod M$$

où k est l'ordre du générateur (i.e. l'ordre de la récurrence).

- ▶ La période maximale théorique de ces générateurs est M^k au lieu de M pour les LCG d'ordre 1.
- ▶ Quelques résultats sur la période

Proposition (Art of Computer Programming, D. Knuth)

Si M est un nombre premier, il existe des multiplicateurs a_1, a_2, \dots, a_k , tels que la suite S_n soit de période $M^k - 1$.

Proposition (Art of Computer Programming, D. Knuth)

Les multiplicateurs a_1, a_2, \dots, a_k satisfont la propriété précédente si le polynôme $X^k - a_1 X^{k-1} - \dots - a_k$ est un polynôme primitif.

Comment déterminer de telles séquences pratiquement ?

- ▶ nombre de séquences de multiplicateurs différentes : M^k , on ne peut pas toutes les tester.
- ▶ Le nombre de séquences donnant des générateurs de période maximale est $\phi(M^k - 1)/k$ où $\phi(m)$ est le nombre d'entiers entre 0 et m premiers avec m .
- ▶ Quelques tests au hasard permettent de trouver rapidement une bonne suite à condition de savoir tester efficacement si un polynôme est primitive.
 \rightsquigarrow facile pour $k = 1, 2, 3$ (Alanen et Knuth).

- ▶ Soit p la période d'un générateur MRG, si l'on considère approximativement plus de \sqrt{p} tirages alors la suite ne passe plus les tests statistiques qui garantissent les bonnes propriétés d'un générateur.
 \rightsquigarrow Besoin d'une période nettement plus longue
- ▶ Pour diminuer le temps de calcul, on choisit souvent tous les a_i nuls sauf 2.
 \rightsquigarrow Une condition nécessaire pour avoir un bon générateur est que $\sum_{i=1}^k a_i^2$ soit grande.

- ▶ L'Ecuyer dans "Good parameters and implementations for combined multiplicative recursive random number generators" suggère de considérer 2 générateurs MRG avec des paramètres différents et de les combiner

$$s_{j,n+1} = a_{j,1}s_{j,n} + a_{j,2}s_{j,n-1} + \dots + a_{j,k}s_{j,n-k} \bmod M_j$$

$$z_n = \left(\sum_{j=1}^J \delta_j s_{j,n} \right) \bmod M_1$$

Proposition (pour $J = 2$)

- ▶ Les propriétés statistiques de z_n ne sont jamais moins bonnes que celles de $s_{1,n}$ ou $s_{2,n}$.
- ▶ Si $s_{1,n}$ et $s_{2,n}$ sont de période maximale et si M_1 et M_2 sont premiers entre eux, alors z_n est de période $M_1 \times M_2$.

```
#define m1      4294967087.0
#define a12     1403580.0
#define a13n    810728.0
{
    long k;
    double p1, p2, u;

    p1 = a12 * g->Cg[1] - a13n * g->Cg[0];
    k = p1 / m1;
    p1 -= k * m1;
    if (p1 < 0.0) p1 += m1;
    g->Cg[0] = g->Cg[1];
    g->Cg[1] = g->Cg[2];
    g->Cg[2] = p1;

    return p1;
}
```

$$x_{k+n} = x_{k+m} + (x_k^u | x_{k+1}^l)A, \quad \forall k \geq 0$$

Génère des entiers non signés sur w bits, i.e. entre 0 et $2^w - 1$.

- n est un entier, c'est le degré de la récurrence, $1 \leq m \leq n$
- $(x_k^u | x_{k+1}^l)$ signifie que l'on concatène les $w - r$ bits de poids fort de x_k avec les r bits de poids faible de x_{k+1} où $0 \leq r \leq w - 1$.
- A est une matrice de taille $w \times w$ de 0 et 1.

$$\begin{pmatrix} & 1 & & & \\ & & 1 & & \\ & & & \ddots & \\ & & & & 1 \\ a_0 & a_1 & \dots & \dots & a_{w-1} \end{pmatrix}$$

xA est calculé par décalage de bits

- la période vaut $2^p - 1$ avec $p = nw - r$.

- L'implémentation en C (`mt19937.c`) a les caractéristiques suivantes :
 - très grande période : $2^{19937} - 1$
 - empreinte mémoire : 624 mots de 32 bits
 - 4 fois plus rapide que `rand()`.

- 1 Pourquoi utiliser des nombres aléatoires ?
- 2 Les générateurs de nombres aléatoires
 - Généralités et premier exemple
 - Des générateurs plus évolués
- 3 Et en parallèle ?
 - Les contraintes
 - Quelles solutions ?
- 4 Comment tester un générateur

- 1 Reproductibilité : comme en séquentielle, on souhaite pouvoir rejouer le même scénario, au moins en conservant le même nombre de processeurs.
- 2 Indépendance entre les processus : les suites produites par chacun des processus doivent être indépendantes entre elles.
- 3 Sur chacun des processeurs, les tirages doivent être statistiquement i.i.d.
- 4 La communication entre les processeurs doit être minimale, idéalement on souhaiterait qu'une fois la phase d'initialisation passée les générateurs ne communiquent plus entre eux.

- ❶ **division en cycles** : diviser un cycle du générateur en sous-segments et chaque flux du générateur utilise son propre sous-segment.
 \rightsquigarrow les flux sont indépendants mais la période est réduite.
- ❷ **paramétrisation** : chaque flux utilise des paramètres différents (graine, multiplicateurs, module) dans le générateur de manière à ce que les suites générées soient indépendantes.

- ❶ Avec un unique générateur, affecter au processeur k , la sous suite $k + n \times p$ où p est le nombre de processeurs.
 - ▶ Exige de savoir faire des sauts de taille p à chaque appel de manière aussi rapide qu'un pas de 1. C'est possible pour les récurrences linéaires du type $x_{n+1} = (Ax_n + B) \bmod M$ de calculer les paramètres Ak, Bk tels que

$$x_{n+k} = (Ak x_n + Bk) \bmod M$$
 - ▶ Les nombres aléatoires utilisés par chacun des processeurs changent dès que l'on modifie le nombre de processeurs.
 - ▶ Les éléments d'un sous-bloc sont régulièrement espacés dans la suite de départ (tous les p).
 \rightsquigarrow les sous-suites régulièrement espacées présentent des corrélations très fortes ([Matteis et Pagnutti, 84])

- ❷ Avec un unique générateur, affecter au processeur k , le bloc d'indices $(k - 1) \times N/p, \dots, k \times N/p$.
 - ▶ Exige de savoir avancer d'un bloc de taille N/p à l'initialisation de manière rapide.
 - ▶ Souvent, on ne sait avancer efficacement que d'une taille de la forme 2^i (voir [L'Ecuyer et Côté, 91]).
 - ▶ La période est réduite.
 - ▶ La répartition des nombres aléatoires sur les différents processeurs dépend de p .
 - ▶ Ce procédé peut faire apparaître des corrélations importantes entre les différentes suites car des corrélations à longue échelle de la suite initiale se traduisent pas des corrélations à courte échelle.

- ▶ Dans le cas d'un générateur à très très grande période (comme Mersenne Twister, $N = 2^{19937}$), les corrélations longue échelle sont très faibles et c'est une bonne solution.
 Même avec 10^5 processeurs, la période est encore $2^{19920} \approx 10^{5981}$.
- ▶ Articles de 2008 : [Haramoto et al., 08] et [Haramoto, 08].
- ▶ Difficiles à implémenter sans recourir à des bibliothèques spécialisées : calculer la puissance " $n^{\text{ème}}$ " de la matrice de récurrence avec n très grand.

- ❶ Changer la graine pour chaque processus
 - ▶ Solution simple à mettre en place et souvent utilisée, **mais sans fondement théorique**.
 - ▶ Deux graines différentes n'assurent pas à priori l'indépendance.
- ❷ Avoir des flux indépendants du même générateur par exemple en utilisant des jeux de paramètres assurant l'indépendance des flux.
 - ▶ Comment déterminer des jeux de paramètres qui assurent l'indépendance.
 - ▶ Phase d'initialisation relative longue.

- ▶ Librairie C++ avec interface Fortran dédiée à la génération de nombres aléatoires en parallèle.
- ▶ Elle contient les générateurs LCG, CMRG, LFG (Lagged Fibonacci), MLFG (Multiplicative Lagged Fibonacci)
- ▶ Les générateurs sont paramétrés par un entier au travers d'une fonction f .
 - ↪ facile de reproduire un même run.
 - ↪ chaque processeur peut créer lui-même sa propre instance du générateur.
- ▶ Possibilité de stocker les générateurs (leur état + d'autres informations) dans un fichier binaire dans un format indépendant machine.

$$s_{n+1} = (As_n + B) \bmod M, \quad \text{avec } M \text{ premier}$$

Proposition

Deux générateurs LCG sont indépendants si les polynômes caractéristiques de leur récurrence sont premiers entre eux.

On choisit de paramétrer par A .

- ▶ Une condition pour obtenir une période maximale est que A soit un élément primitif modulo M , i.e. $\{A^i \bmod m; 1 \leq i \leq m-1\} = \{1 \leq i \leq M-1\}$.
- ▶ Si a_1 et a_2 sont des éléments primitifs modulo M , alors $a_1 = a_2^i \bmod M$ pour un i premier avec $\phi(M) = M-1$, puisque M est premier.
- ▶ Il suffit de connaître les entiers premiers avec $M-1$.

La paramétrisation par B ne conduit pas à de bons générateurs.

Dynamic Creator Mersenne Twister

- ▶ Pour un triplet (w, m, n) fixé, on peut trouver des jeux de vecteur a tels que les générateurs associés soient indépendants.
Pour $p = 521$, $m = 9$, $n = 17$, $r = 23$ et $w = 32$, il y a $2^{16} = 65536$ valeurs possibles pour A et chaque générateur associé est de période maximale $2^p - 1$ (c'est le générateur PNL_RNG_DCMT).
- ▶ Dans le vecteur a est encodé un identifiant id assurant l'indépendance pour 2 valeurs différentes de id ↪ d'où le nombre réduit de générateurs.
- ▶ Combien de temps faut-il pour trouver un a assurant l'indépendance ?
 - ↪ temps CPU moyen : 0.32
 - ↪ probabilité que le temps de création > 1 sec. ≈ 0.006 .
- ▶ Sur les très gros clusters, pas assez de générateurs différents disponibles.

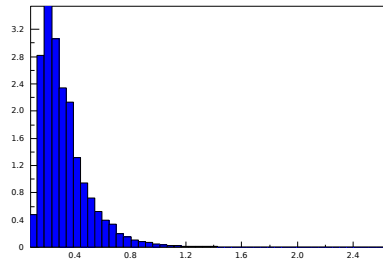


FIGURE: temps CPU nécessaire à la création d'un générateur

- ▶ `PnIRng*pnl_rng_dcmt_create_id(int id, ulong seed)`
Crée un générateur de type `PNL_RNG_DCMT` d'identifiant `id`. Deux générateurs avec des identifiants différents sont indépendants. Le générateur créé doit être initialisé avec `pnl_rng_sseed`.
- ▶ `PnIRng**pnl_rng_dcmt_create_array_id(int start_id, int max_id, ulong seed, int *count)`
Crée un tableau de générateurs de type `PNL_RNG_DCMT` et d'identifiants entre `start_id` and `max_id`. Ces générateurs sont indépendants et doivent être initialisés avec `pnl_rng_sseed`.
- ▶ `void pnl_rng_free(PnIRng **)`
Libère a `PnIRng`.
- ▶ `void pnl_rng_sseed(PnIRng *rng, unsigned long int s)`
Fixe la graine du générateur `rng` en utilisant `s`. Si `s=0`, une graine par défaut est utilisée.

Squelette d'un programme parallèle de type Monte-Carlo

```
Créer un générateur en fonction du rang
Calculer une moyenne Monte Carlo avec le générateur g[rank]
Envoyer le résultat au maître
Si processus maître
    Attendre la réponse de chaque processeur esclave
    et en faire la moyenne
Sinon
FinSi
```

Conclusion (partielle) sur les générateurs aléatoires en parallèle

- ▶ Si l'on a besoin de peu de générateurs indépendants :
 - ▶ La paramétrisation de A pour le générateur Mersenne Twister est une bonne solution
 - ▶ On peut utiliser les méthodes `MPI_Pack` et `MPI_Unpack` pour stocker les générateurs dans des fichiers binaires sous un format indépendant machine et relire les générateurs.
- ▶ Si l'on a besoin de beaucoup de générateurs indépendants :
 - ▶ Le découpage en cycle du générateur Mersenne Twister originale de période $2^{19937} - 1$ est une bonne solution. Même découpée en 100,000 sous-séquence, chacune est encore de période $\approx 2^{19920}$ et les corrélations entre les sous-séquences sont très faibles
 - ▶ La paramétrisation d'un LCG peut être une solution si chaque générateur n'effectue que peu de tirages.

- 1 Pourquoi utiliser des nombres aléatoires ?
- 2 Les générateurs de nombres aléatoires
 - Généralités et premier exemple
 - Des générateurs plus évolués
- 3 Et en parallèle ?
 - Les contraintes
 - Quelles solutions ?
- 4 Comment tester un générateur

Il existe de nombreux programmes de tests

- ▶ [Diehard] : ensemble de tests statistiques très complets
- ▶ Dans la librairie SPRNG, on retrouve ces tests et d'autres encore.

Test du χ^2 I

- 1 Pour tester l'adéquation à une loi donnée (normalement discrète).
On découpe l'intervalle $[0, 1]$ en m sous-intervalles réguliers et on teste la répartition uniforme sur chaque sous-intervalle grâce à la statistique

$$\zeta_n = nm \sum_{i=1}^m (N_i/n - 1/m)^2$$

qui suit une loi du $\chi^2(m-1)$, où N_i est le nombre de tirages dans l'intervalle i et n est la taille de l'échantillon.

↪ On calcule $\mathbb{P}(\chi^2(m-1) > \zeta_n)$

Test du χ^2 II

- 2 Pour tester l'indépendance de 2 variables aléatoires.
Soit $(X_i, Y_i)_i$ une suite de variables aléatoires indépendantes à valeurs dans $[0, 1] \times [0, 1]$. On veut tester l'indépendance de (X_i) et (Y_i) . La statistique suivante

$$\zeta_n = n \sum_{i=1}^m \sum_{j=1}^m \frac{(N_{i,j} - \frac{N_{i,\cdot} N_{\cdot,j}}{n})^2}{N_{i,\cdot} N_{\cdot,j}}$$

suit une loi du $\chi^2((m-1)^2)$, avec

$$N_{i,j} = \#\{(X_k, Y_k) \in I_i \times I_j\}; \quad N_{i,\cdot} = \#\{Y_k \in I_i\}; \quad N_{\cdot,j} = \#\{Y_k \in I_j\}$$


où I_i est le i -ème intervalle de $[0, 1]$.


↪ On calcule $\mathbb{P}(\chi^2((m-1)^2) > \zeta_n)$


- ▶ test de Kolmogorov Smirnov : permet de tester l'adéquation entre les données et une loi supposée et utilisant la fonction de répartition empirique (normalement pour les lois continues).
- ▶ La loi forte des grands nombres et le TCL sont-ils vérifiés ?
- ▶ Test spectral : teste la distribution jointe de t éléments consécutifs $x_n, x_{n+1}, \dots, x_{n+t-1}$. Les mauvais générateurs font apparaître des figures géométriques très caractéristiques.


Dynamic creation of pseudorandom number generators.


Monte Carlo and Quasi-Monte Carlo Methods 1998, Springer, 2000, pp 56–69.


-  H. Haramoto, M. Matsumoto, T. Nishimura, F. Panneton, and P. L'Ecuyer. Efficient jump ahead for f2-linear random number generators. INFORMS J. on Computing, 20 :385–390, July 2008.


-  H. Haramoto, M. Matsumoto, and P. L'Ecuyer. A fast jump ahead algorithm for linear recurrences in a polynomial space. Proceedings of the 5th international conference on Sequences and Their Applications, SETA '08, pages 290–298, Berlin, Heidelberg, 2008. Springer-Verlag.

-  G. Marsaglia. Diehard. <ftp://stat.fsu.edu/pub/diehard>.

-  D. E. Knuth. The Art of Computer Programming, Vol. 2 : Seminumerical Algorithms, Second edition. Addison-Wesley, Reading, Massachusetts, 1981.

-  P. L'Ecuyer. Efficient and portable combined random number generators. Comm. of the ACM, 31 :742-774, 1988.

-  L'Ecuyer and Côté (1991). Implementing a Random Number Package with Splitting Facilities. ACM Transactions on Mathematics Software, 17(1) :98-111, 1991.

-  A. De Matteis and A. Pagnutti. Long-range correlations in linear and non-linear random number generators. Parallel Computing, 1 :175-180, 1984.

-  M. Matsumoto and T. Nishimura.